Source Code Analysis, Management, and Visualization for PROLOG

Dissertation zur Erlangung des
naturwissenschaftlichen Doktorgrades
der Julius-Maximilians-Universität Würzburg

vorgelegt von

Marbod Hopfner

aus Iphofen

Würzburg, 2008

Eingereicht am 27.11.2008
bei der Fakultät für Mathematik und Informatik

1. Gutachter: Prof. Dr. Dietmar Seipel
2. Gutachter: Prof. Dr. Wolff von Gudenberg

Tag der mündlichen Prüfung: 25.05.2009

Preface

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software. The discipline of software engineering encompasses knowledge, tools, and methods for defining software requirements, and performing software design, computer programming, user interface design, software testing, and software maintenance tasks. Such software can contain millions of lines of source code, making it comparable in complexity to the most complex modern machines. For example, a modern airliner has several millions of physical parts, while its software can run to 4 million lines of code. Software engineering also draws on knowledge from fields such as computer engineering, computer science, management, mathematics, project management, quality management, software ergonomics, and systems engineering.

The design, implementation and extension of computer programs is a very complex process. For keeping an overview of large programs it is necessary to *organize* and *structure* the source code. For small programs, having about hundred lines of source code, it is often sufficient to use a simple editor. But if a program grows, *Integrated Development Environments (IDEs)* are needed for supporting programmers. IDEs help both newcomers and expert programmers in implementing, organizing, structuring and extending their programs. Using an IDE for *analyzing* source code, *calculating metrics*, *refactoring* or *visualizing* methods is useful. Call graphs can be used for human understanding of programs, or as a basis for further analyses, such as tracking the flow of values between procedures [69].

The declarative language PROLOG is very appropriate for analyzing and refactoring source code. Using its symbolic computation and meta-programming techniques, we can nicely analyze source code in a structural representation such as XML. We have designed the tool SCAV (Source Code Analysis and Visualization) consisting of the libraries RAR (Reasoning about Rules), VISUR (Visualization of Rules), and the software repository PROSORE. These libraries support

- software metrics and source code statistics,
- program slicing and refactoring, and
- visualization of code structures.

This dissertation is organized in five parts: Part I, the *Background*, introduces source code analysis and visualization. Chapter 1 explains some well known software development processes. We sketch basic software engineering methods, and we explain software metrics, design patterns, and software repositories. In Chapter 2, we introduce the declarative programming language PROLOG, we list some existing Integrated Development Environments (IDE) for PROLOG, and we present some typical PROLOG refactorings.

Part II deals with the *management and analysis* of source code. Chapter 3 explains the structure of the used software repository PROSORE in detail. We use XML as the basic data structure for complex data, such as programs or graphs. Chapter 4 introduces the basic RAR methods and contains some examples of source code analysis in PROLOG, such as source code statistics, and predicate spectra. We give some examples how to change the basic predicates, in order to visualize and analyze languages other than PROLOG, and we present case studies for PHP, and JAVA. Chapter 5 describes slicing of PROLOG source code.

Part III deals with the *visualization* of source code. We have implemented a graph library based on the Graph eXchange Language GXL. Our graph library, which we explain in Chapter 6, contains methods to validate and to transform GXL graphs, to layout graphs, and to classify nodes and edges. Chapter 7 describes the GUI of SCAV. The case studies in Chapter 7 give some examples of visualizations.

Part IV, the *Conclusion*, summarizes the essential parts of this work and gives an outlook on possible extensions by further methods for source code analysis, visualization and refactoring.

In Part V, the *Appendix*, we describe some configurations, XPCE programming techniques and useful tools, which we have implemented.

Acknowledgment

During my doctorate many people crossed my path. I am grateful to all these people who helped me in writing this thesis. First of all, I thank a lot my doctor father Prof. Dr. Dietmar Seipel from the University of Würzburg. Our regular meetings for developing new ideas gave me interesting insights, and his guidance improved my work a lot. I also thank Prof. Dr. Wolff von Gudenberg, who agreed to be my second reviewer. His suggestions and reviewing of the thesis helped me, too.

I would also like to thank Prof. Dr. Ulrich Güntzer from the University of Tübingen for the years in his research group and my colleagues at the University of Tübingen, Andreas Ludwig and Bernd Heumesser, with whom I had a great working atmosphere.

I thank all proofreaders, especially my cousin, Prof. Dr. Reinhold Behringer, and Dr. Jürgen Engbring. Both gave me critical and useful feedback. Furthermore, I thank Andreas Klein and Andreas Vetter for their friendly help with software and operating system problems.

Finally, I thank my parents, Elena and Wielant, and the very special Susanne for their support and understanding – especially in the last months during the completion of this thesis. Without them, this work would not have been possible.

To Susi and my parents

Contents

Preface iii

I Background 1

1 Software Engineering, Development and Programming Tools 5
- 1.1 Software Development . . . 5
- 1.2 Software Metrics . . . 14
- 1.3 Design Patterns . . . 15
- 1.4 Call Dependencies and Program Slicing . . . 16
- 1.5 Software Repositories . . . 17

2 Software Engineering and PROLOG 19
- 2.1 Integrated Development Environments for PROLOG . . . 20
- 2.2 PROLOG Refactorings . . . 25

II Management and Analysis of Source Code 29

3 The Software Repository PROSORE 35
- 3.1 Representation of Source Code in PROLOGML . . . 37
  - 3.1.1 Source Code . . . 38
  - 3.1.2 Hierarchy Tree and Exported Predicates . . . 44
  - 3.1.3 Management of PROLOGML using PL4XML . . . 51
- 3.2 The PROSORE Database . . . 56
  - 3.2.1 The Structure . . . 56
  - 3.2.2 Basic Access Methods . . . 62
  - 3.2.3 Updates . . . 69
  - 3.2.4 Call Dependencies . . . 72
  - 3.2.5 The Hierarchy . . . 75
  - 3.2.6 Predicate Groups and Meta-Call Predicates . . . 77

4 Source Code Analysis **83**
4.1 Call Dependencies . . . 84
4.1.1 Calls between Predicates, Files and Packages . . . 85
4.1.2 Strongly Connected Components . . . 92
4.1.3 Dependencies in PROLOG, JAVA or PHP . . . 93
4.2 Anomalies, Software Metrics, and Design Patterns . . . 97
4.2.1 Undefined Predicates and Dead Code . . . 99
4.2.2 Predicate Properties . . . 109
4.2.3 File Statistics . . . 112
4.2.4 PROLOG Design Patterns . . . 114
4.3 Spectra . . . 117
4.3.1 Predicate Spectra . . . 117
4.3.2 Package Spectra . . . 126

5 Slicing **129**
5.1 Basic Concepts . . . 130
5.2 Special PROLOG Constructs . . . 136
5.2.1 Directives . . . 137
5.2.2 Consults . . . 138
5.2.3 Global Variables . . . 138
5.2.4 Test Predicates . . . 139

III Visualization of Source Code **141**

6 A Graph Library Based on GXL **145**
6.1 Layout of Extended GXL Documents . . . 145
6.1.1 The GXL Document Extensions for Layout . . . 146
6.1.2 Layout Methods . . . 156
6.2 Basic Methods for GXL Documents . . . 162
6.2.1 Correctness of GXL Documents . . . 166
6.2.2 Classifying Nodes and Edges . . . 167
6.2.3 Conversion between Ugraphs and GXL Documents . . . 169
6.3 Transformation Methods for GXL Documents . . . 174
6.3.1 Completing and Correcting GXL Documents . . . 176
6.3.2 Deleting Nodes and Edges . . . 178
6.3.3 Highlighting Nodes and Edges . . . 181
6.3.4 Merging Nodes . . . 187
6.4 The Picture Class for GXL Graphs . . . 194
6.4.1 Visualization of GXL Graphs and Ugraph . . . 195
6.4.2 Reconstruction of Graphs from Pictures . . . 200

7 The Visualization Tool VISUR **203**
7.1 The System VISUR 203
7.1.1 The Menu Bar 204
7.1.2 The Hierarchy Browser 207
7.1.3 The Visualization Picture 208
7.1.4 Plug-ins – Extending the GUI 210
7.2 Case Studies 211
7.2.1 Cross Reference Graphs for Packages 212
7.2.2 Visualizing Rule Based Knowledge 220
7.2.3 Visualizing Other Graphs 221

IV Conclusions **227**

V Appendix **231**

A XPCE Programming Techniques **233**
A.1 Applying Methods to Objects of a Picture 233
A.2 Generating Mouse Click Events 235
A.3 Generating Pull-down and Pop-up Menus 237
A.4 The Hierarchy Browser 241
A.5 The Alias Browser for the File History 249

B The XML-Based Configuration of the Picture Class for GXL Graphs **257**
B.1 The Default Configuration for Graph Pictures 257
B.2 The DTD of the XML Configuration Document 258

C Writing Plug-ins **267**
C.1 Ancestor Relations Plug-in 267
C.1.1 The Ancestor Facts 267
C.1.2 Plug-ins for SCAV 269
C.1.3 Generating the Hierarchy Tree 273
C.2 JAVA Plug-in 276

D Examples, Default Settings and Configurations **279**
D.1 XML Document in Field Notation 279
D.2 XML Configuration Special Predicates 280
D.3 Default GXL Settings 281
D.4 Default Visualization Configuration 282

List of Figures and Method Index **289**

Bibliography **295**

Part I

Background

Software engineering is the application of a systematic, disciplined, quantifiable approach to the *development*, *operation*, and *maintenance* of software. It encompasses techniques and procedures, often regulated by a *software development process*, with the purpose of improving the reliability and maintainability of software systems. The effort is necessitated by the potential complexity of systems, which may contain millions of lines of source code.

Up to now, software engineering techniques are primarily applied within large projects implemented in C, C++, C#, JAVA, PERL, FORTRAN, DELPHI, etc. For these languages, there exists many tools supporting the development of software. In the last few years, more and more people deal with software development in logic programming languages, such as PROLOG [72]. There exist a lot of PROLOG implementations, e.g., Amzi! PROLOG, B-PROLOG, GNU PROLOG, IF/PROLOG, Quintus PROLOG, SICStus PROLOG, Visual PROLOG, SWI-PROLOG.

Chapter 1

Software Engineering, Development and Programming Tools

Software engineers use a wide variety of technologies, e.g., compilers, code repositories, text editors. They also use a wide variety of practices to carry out and coordinate their efforts: pair programming, code reviews and daily stand up meetings. The International Conference on Software Engineering (ICSE) is the biggest and oldest conference devoted to software engineering. This annual conference discusses improvements in research, education, and practice [80].

Software development is usually done in teams. Each developers is assigned a specific task and is responsible for a certain functionality or module of the software. For these modules to work together properly, it is essential that interfaces are defined and are designed very well. Furthermore, it may be happen that developers need to swap responsibility. This requires that the code is well documented and is written according to general guidelines, allowing a new developer to quickly become familiar with the code functionality. This requires an adherence to those guidelines, without individuals using *their own style*, which often cannot be comprehended by their colleagues.

In Section 1.1, we introduce some software development tools, processes, and software engineering methods. Section 1.2 gives some examples of well known software metrics, Section 1.3 explains design patterns, Section 1.4 introduces call dependencies and program slicing and Section 1.5 explains common software repositories.

1.1 Software Development

Software Development today is usually a process divided into two parts: the creation and editing of files which contain the *source code*, and the compiling and testing/debugging of the executable. As software system grew more complex, the software developer (or the software development team) is required to manage an enormous amount of lines of source code (cf. Figure 1.1). For example, the current kernel of Linux has about 9 Million, and Windows XP about 40 Million lines of source code [74, 75].

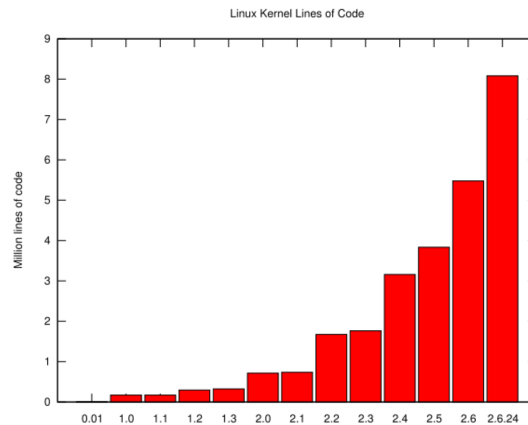


Figure 1.1: Increasing Number of Lines of Source Code of the Linux Kernel

Software, which consists of an immense number of lines of source code is difficult to comprehend and to maintain without using corresponding tools. E.g., some well-known problems of software development are *duplicated source code*, *undefined methods* or methods, which are not used anymore (*dead code*). These problems especially appear during team software authoring. In order to put things right, *refactoring* source code is helpful. It is important to apply refactoring methods at regular intervals, in order to improve the design of large software projects, and to keep an overview of the whole source code. Also, refactoring source code improves the design of a software project, which initially is well designed, but which is incrementally extended without thinking about the architecture of the software.

Generally, in order to obtain a better structured source code, we first detect the problems of badly structured and implemented source code. Here, special software tools, which contain software engineering metrics, and visualize the structure of the source code are useful. Then, after detecting software design faults, we refactor the source code.

Integrated Development Environments

It is necessary to have two types of tools: on the one hand, we need tools, which detect badly structured source code, design faults, and necessary refactorings and on the other hand, we need tools, which support refactorings. This may be simple tools like a syntax highlighting editor, or more complex tools, which visualize the source code and which even allow for interactive refactorings of source code.

To understand difficult coherences, it is useful to *see a simple visualization of complex structures*. It is important that the visualization is scaleable in order not to overload the view with too detailed information. Too many visualized nodes and edges are not helpful. It is useful, if the source code can be divided into several components, whereby each component is visualized in an own picture.

For the management of complex source code and a large number of lines of source code, special tools have been developed. In the following, we list some well known software development environments.

- *Microsoft Visual Studio* is the main Integrated Development Environment (IDE) from Microsoft. It can be used to develop console and GUI applications along with windows forms applications, web sites, web applications, and web services in both native code as well as managed code for all platforms supported by Microsoft Windows, Windows Mobile, the .NET Framework, the .NET Compact Framework and Microsoft Silverlight. Visual Studio includes a code editor supporting IntelliSense as well as code refactoring.
- *Borland DELPHI* is an IDE created by Borland, and now owned by Borland's subsidiary, CodeGear. DELPHI 2007 supports the DELPHI programming language and C++ for the 32 bit Microsoft Windows platform, and DELPHI and C# for the Microsoft .NET platform. Borland Kylix is the equivalent to DELPHI for the Linux platform. DELPHI is mainly used for the development of desktop and enterprise database applications, but it is a general-purpose software development tool suitable for most software projects.
- *Borland C++ Builder* is a popular rapid application development (RAD) environment produced by the CodeGear subsidiary of Borland for C++. C++ Builder combines the Visual Component Library and IDE as found in DELPHI with a C++ compiler.
- *Eclipse* is an open-source IDE written primarily in JAVA. The initial codebase originated from VisualAge. In its default form it is meant for JAVA developers, consisting of the JAVA Development Tools (JDT). Users can extend its capabilities by installing plug-ins written for the Eclipse software framework, such as development toolkits for other programming languages, and can write and they contribute their own plug-in modules. Language packs provide translations into over a dozen natural languages.
- *Xcode* is Apple's suite of tools for developing software on Mac OS X.

A *well-engineered design* of source code helps to understand and extend the functionality of methods, too.

Software Development Processes

A software development process is a structure imposed on the development of a software product. Synonyms include software life cycle and software process. There are several models for software development processes, each describing approaches to a variety of tasks or activities that take place during the process. A model is then used to measure what a development organization or project team actually does during software development. This information is analyzed to identify weaknesses and to drive improvement.

Waterfall Process

The *waterfall process* is the best-known and oldest process. As the origin of the term waterfall process often an article published in 1970 by Winston Royce is cited, although Royce did not use the term "waterfall" in this article [89]. In the waterfall process the developers (roughly) follow these steps:

- state requirements,
- analyze requirements,
- design an approach to solving the problem,
- develop the architecture of a software framework for that solution,
- implement code,
- test (perhaps unit tests then system tests),
- deploy, and
- post implementation.

After each step, the process proceeds to the next step, just as builders don't revise the foundation of a house after the framing has been erected. There is a misconception that the process has no provision for correcting errors in early steps (for example, in the requirements). The problems in waterfall do not arise from immature engineering practices, particularly in requirements analysis and requirements management. Often the supposed stages are part of a joint review between customer and supplier, the supplier can, in fact, develop at risk and evolve the design but must sell off the design at a key milestone called *Critical Design Review*. This shifts engineering burdens from engineers to customers who may have other skills. [81]

V-Model

The *V-model* is a software development model which can be considered to be the extension of the waterfall model [84]. Instead of moving down in a linear way, the process steps are bent upwards after the coding phase, to form the typical V shape (cf. Figure 1.2). The V-model demonstrates the relationships between each phase of the development life cycle and its associated phase of testing. The 'V' can also stand for the terms *Verification* and *Validation*.

The V-model is more helpful and profitable to companies as it reduces the time for whole development of a new product and can also be used to some complex maintenance projects.

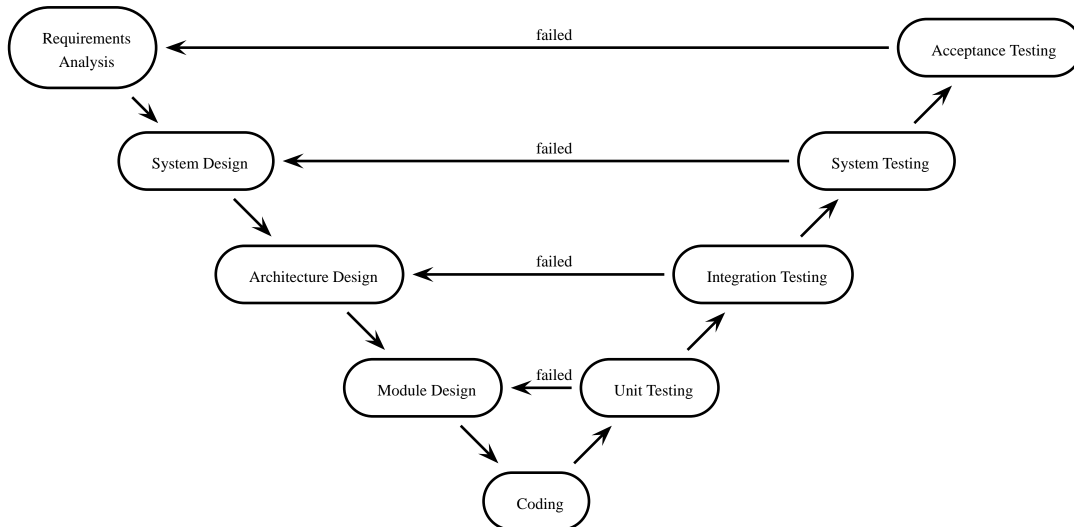


Figure 1.2: V-Model

Requirements Analysis In this phase, the requirements of the proposed system are collected by analyzing the needs of the users. Usually, the users are interviewed and a document called the *user requirements document* is generated. The user requirements document will typically describe the system's functional, physical, interface, performance, data, security requirements etc. as expected by the user. The software will not be designed or built, yet. The user *acceptance tests* are designed in this phase.

System Design System engineers analyze the proposed system by studying the *user requirements document*. They figure out possibilities and techniques by which the user requirements can be implemented. If any of the requirements are not feasible, the user is informed of the issue. A solution is found and the *user requirement document* is edited accordingly. The *software specification document*, which serves as a blueprint for the development phase, is generated. This document contains the general system organization, menu structures, data structures etc. The *documents for system testing* is prepared in this phase.

Architecture Design In this phase, the typical list of modules, brief functionality of each module, their interface relationships, dependencies, database tables, architecture diagrams, technology details etc., are realized. The *integration testing design* is carried out in this phase.

Module Design Now, the designed system is broken up into smaller units or modules and each of them is explained so that the programmer can start coding directly. The *unit test design* is developed at this stage.

Unit Testing Unit testing implies the first stage of a dynamic testing process. According to software development expert Barry Boehm [2], a fault discovered and corrected in the unit testing phase is more than a hundred times cheaper than a fault detected after delivery to the customer. It involves the analysis of the written code with the intention of eliminating errors. Testing is usually white box. It is done using the *unit test design* prepared during the *module design* phase.

Integration Testing In integration testing the separate modules will be tested together to expose faults in the interfaces and in the interaction between integrated components. Testing is usually black box as the code is not directly checked for errors. It is done using the *integration test design* prepared during the *architecture design* phase.

System Testing System testing will compare the system specifications against the actual system. The system test design is derived from the system design documents and is used in this phase. Sometimes system testing is automated using testing tools. Once all the modules are integrated, several errors may arise.

User Acceptance Testing The purpose of acceptance testing is to verify the system and the changes according to the original needs. We test whether a system satisfies its acceptance criteria or not. The customer can accept the system or not, and we can test the software in the "real world" by the intended audience.

Iterative and Incremental Development

Iterative and incremental development is a cyclical software development process developed in response to the weaknesses of the waterfall model. It prescribes the construction of initially small but ever larger portions of a software project to help all those involved to uncover important issues early before problems or faulty assumptions can lead to disaster. Iterative processes are preferred by commercial developers, because they allow a potential of reaching the design goals of customers who do not know how to define what they want.

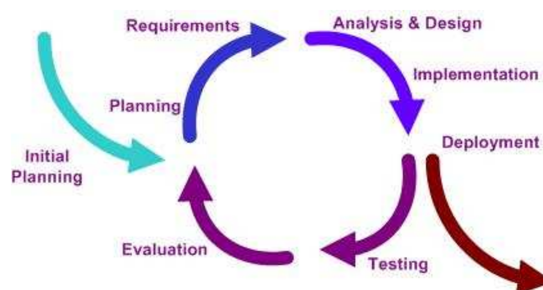


Figure 1.3: Iterative Development Model

The basic idea behind iterative enhancement is to develop a software system incrementally, allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system (cf. Figure 1.3). Learning comes from both the development and the use of the system. At each iteration, design modifications are made and new functional capabilities are added [81, 85]. There exists several types of iterative and incremental development.

Agile Software Development is an iterative and incremental (evolutionary) approach to software development processes. It is a conceptual framework for software engineering that promotes development iterations throughout the life cycle of the project. Agile processes use feedback, rather than planning, as their primary control mechanism. The feedback is driven by regular tests and releases of the evolving software. Some of the principles of *agile software development* are [86]:

- customer satisfaction by rapid, continuous delivery of useful software,
- working software is delivered frequently (weeks rather than months),
- working software is the principal measure of progress,
- even late changes in requirements are welcome,
- close, daily cooperation between business people and developers,
- face-to-face conversation is the best form of communication,
- projects are built around motivated individuals, who should be trusted,
- continuous attention to technical excellence and good design,
- simplicity,
- self-organizing teams,
- regular adaptation to changing circumstances.

E.g., Joachim Baumeister [4] introduces an agile process model for developing diagnostic knowledge systems.

Agile Software Development uses a process model that is inspired by the popular methodology eXtreme programming known in software engineering research. The development process is structured by the following agile phases: Definition of the system metaphor, the planning game, the implementation phase, and the integration phase.

The *system metaphor* defines a common system of names for the knowledge system project. With the system metaphor, the semantics of, e.g., a diagnosis, a question, a question set, and a case are defined. The remaining phases are repeatedly traversed in a cyclic manner. The *planning game* typically considers the short-term scope and requirements of the running project. It is designed to provide an early and concrete feedback, a flexible

schedule of the development process, and it lasts as long as the system lasts. The *implementation phase* consists of a test and a code implementation: in principle, the coding of new knowledge or the restructuring of existing knowledge is always preceded by the coding of appropriate test knowledge. The *integration phase* guarantees an always running system by continuously integrating the new and modified knowledge into a production version of the knowledge system. The production version of the system is always validated using additional integration tests, extensive tests that are usually time-consuming, and that are covering the expected behavior of the knowledge system as a whole [4, 5, 6].

Test Driven Development is a software development technique consisting of short iterations where new test cases covering the desired improvement or new functionality are written first, then the production code necessary to pass the tests is implemented, and finally the software is refactored to accommodate changes. The availability of tests before actual development ensures rapid feedback after any change. Practitioners emphasize that test-driven development is a method of designing software, not merely a method of testing [87].

Rapid application development is a software development process developed initially by James Martin in 1991 [34]. The methodology involves iterative development, and the construction of prototypes. Traditionally the rapid application development approach involves compromises in usability, features, and/or execution speed. It is described as a process through which the development cycle of an application is expedited. *Rapid Application Development* thus enables quality products to be developed faster, saving valuable resources [88]. One problem with previous methodologies was that applications took so long to build that requirements had changed before the system was complete, resulting in inadequate or even unusable systems.

Software Engineering Methods

The process of gathering and analyzing an application's requirements and incorporating them into a program design is a complex one, and the industry currently supports many methodologies that define formal procedures (e.g., waterfall process, v-model, agile software development process). These methodologies are complemented by models.

Unified Modeling Language (UML) In the field of software engineering, UML is a standardized visual specification language for object modeling, developed by the *Object Management Group* (OMG). Modeling is the design of software applications before coding. It is an essential part of large software projects, and helpful to medium and even small projects as well. UML helps to specify, visualize, and document models of software systems, including their structure and design. It is a general-purpose modeling language that includes a graphical notation used to create an abstract model of a system, referred to as

a UML model. UML has allowed software developers to concentrate more on design and architecture [56].

Model-Driven Engineering (MDE), Model-Driven Development (MDD) MDE refers to the systematic use of models as primary engineering artifacts throughout the engineering life cycle. MDE can be applied to software, system, and data engineering. As it pertains to software development, MDE, and MDD refer to a range of development approaches that are based on the use of software modeling as a primary form of expression. Sometimes models are constructed to a certain level of detail, and then code is written by hand in a separate step. Sometimes complete models are built including executable actions. Code can be generated from the models, ranging from system skeletons to complete, deployable products. With the introduction of UML, MDD has become very popular today with a wide body of practitioners and supporting tools. More advanced types of MDD have expanded to permit industry standards which allow for consistent application and results. The continued evolution of MDD has added an increased focus on architecture and automation.

MDD technologies with a greater focus on architecture and corresponding automation yield higher levels of abstraction in software development. This abstraction promotes simpler models with a greater focus on problem space. Combined with executable semantics this elevates the total level of automation possible.

The OMG has developed a set of standards called *Model Driven Architecture* (MDA), building a foundation for this advanced architecture-focused approach [82].

Some examples, with which among others software engineering methods deal, are [39]:

- Too much code. Hundreds of thousands, or millions of lines of source code for an *application*. This causes that the source code is difficult to understand and it is not possible to get a whole overview and understanding of the source code.
- No one really understands exactly what the code is doing, sometimes even, if it is only short source code.
- There is insufficient documentation. In fact, since it is so unusual to have documentation, it is ignored when it does exist.
- We have little chance of ever understanding all of the interactions between different parts of a large system.
- The people who do have a chance of understanding the code, very often don't understand the application.
- People do not reuse existing software, either because they don't trust it or because it is not general enough.

Therefore, it is important to use tools, which help to understand and handle the source code in an easy way.

1.2 Software Metrics

A software metric is a measure of some property of a piece of software or its specification. Since quantitative methods have proved so powerful in other sciences, computer science practitioners and theoreticians have worked hard to bring similar approaches to software development [83]. Tom DeMarco stated,

> "You can’t control what you can’t measure [9]."

Software metrics help to assess the quality and testability of code. Many different aspects of source code can be measured. In the following, we briefly list some software metrics used in software engineering [25, 83].

- Variable Level Metrics
  - *Functions Reading* is the number of functions in the source files which read the variable.
  - *Functions Setting* is the number of functions in the source files which set the variable.
  - *Functions Using* is the number of functions in the source files which read or set the variable.
- Function Level Metrics
  - *Cyclomatic complexity*, which was developed by Thomas McCabe, is used to measure the complexity of a program. It directly measures the number of linearly independent paths through a program’s source code. Cyclomatic complexity is computed using a graph that describes the control flow of the program. The nodes of the graph correspond to the commands of a program. A directed edge connects two nodes if the second command might be executed immediately after the first command.
  - *Code coverage* is a measure used in software testing. It describes the degree to which the source code of a program has been tested.
  - *Time* spent executing the function (profile data).
  - *Lines in Function* is the number of lines in the function.
  - *Global Variables Used* is the number of global variables set or read by the function.
- File Level Metrics
  - *Lines of Source Code* measures the size of a software program by counting the number of lines in the text of the program’s source code. LoC is typically used to predict the effort that will be required to develop a program, as well as to estimate programming productivity or effort once the software is produced.

  - *Lines of Comments* is the number of lines of comments in the file.
  - *Variables in File* is the number of variables and parameters defined in the file.

- Class Level Metrics
  - *Cohesion* is a measure of how strongly-related and focused the various responsibilities of a software module are. Cohesion is an ordinal type of measurement and is usually expressed as *high cohesion* or *low cohesion*. Modules with *high cohesion* tend to be preferable because *high cohesion* is associated with several desirable traits of software including robustness, reliability, reusability, and understandability whereas *low cohesion* is associated with undesirable traits such as being difficult to maintain, difficult to test, difficult to reuse, and even difficult to understand. *Cohesion* is often contrasted with *coupling*, a different concept. Nonetheless *high cohesion* often correlates with loose coupling, and vice versa.
  - The *number of concrete and abstract classes and interfaces* in the package is an indicator of the extensibility of the package.
  - *Coupling* or *dependency* is the degree to which each program module relies on each one of the other modules.
  - *Member Classes* is the number of nested classes (member classes) of the class.

Software metrics help to decide, if source code should be refactored or not. They can also be used to verify the progress and the improvements of software development.

1.3 Design Patterns

In software engineering, a *design pattern* is a general, repeatable solution to a commonly occurring problem in software design. Design patterns cover knowledge of software experts, which has been built up along many years of software engineering, e.g., problems such as adapting the interface of one object to that of another object or notifying an object of a change in another object's state. It is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. The idea is to use sophisticated design ideas to solve problems that often waste time solving over and over again in the programming.

Solutions are presented in generalized diagrams of data and logic structures. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Algorithms are not thought of as design patterns, since they solve computational problems rather than design problems.

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to

prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns [78]. In Chapter 4, we show some examples how we retrieve design patterns in PROLOG source code.

1.4 Call Dependencies and Program Slicing

Debugging or refactoring software is a complex process. In general, it is easier to understand complex facts, if they are visualized in graphs or figures instead of the descriptions in words, formulas or mathematical definitions. Therefore, it is preferable to *visualize* facts whenever it seems to be helpful to facilitate the comprehension of the user. We generate diverse visualizations in order to show dependencies, e.g., between predicates, files (cf. Figure 1.4) or packages.

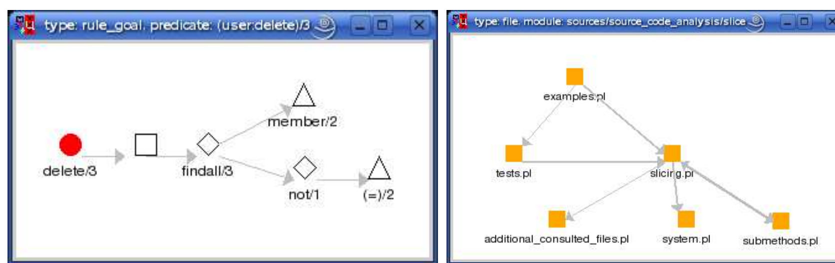


Figure 1.4: Call Dependencies between Predicates and Files

All these visualizations are based on the predicate *call dependencies*. Predicate call dependencies can be used for visualizing the structure of a program and for *program slicing*.

If we want to debug a large program, it is useful to consider only a part of the program. Therefore, it helps to extract a program slice of the source code. The term *program slicing* denotes the analysis of a computer program with the aim to find out what instructions of a program influences or are influenced by a *certain* instruction of a particular topic. We differ between several types of slicing [79]:

- *Static slicing* is the analysis of a program which is independent of a certain startup configuration.
- *Dynamic slicing* is the analysis of a program depending on a specific startup configuration.
- *Approximate dynamic slicing* is nearly the same as *static slicing*. But instead of slicing the whole program, only those parts of the program, which are executed using a special startup configuration are sliced.

Slicing source code is useful, e.g., in order to debug software or in order to apply refactoring methods on parts of a large software system. We try to separate only full executable

parts of the source code. We can even pass sliced parts to other persons, which only need/want a subsystem of a large system. If someone wants to import additional functionalities to his own program, without importing the whole system, we extract a program slice. Considering only parts of the system is easier to understand and integrate into another system. This minimizes the effort of importing PROLOG methods.

In order to calculate a slice, we have to determine all dependencies of the necessary methods. Therefore, we need among other things, the knowledge about the predicate call dependencies. We generate the transitive closure of the corresponding methods. How to retrieve all dependencies for a program slice is described in detail in Chapter 5.

1.5 Software Repositories

A repository is a place where data is stored and maintained. It can be, e.g., a place where digital data is stored, a site where eprints are located, a place where multiple databases or files are located for distribution over a network, a computer location that is directly accessible to the user without having to travel across a network, or a place where anything is stored for probable reuse.

The following examples show, where repositories are useful: component repository management, data warehouse, data mining, digital repository, institutional repository, directory, national repository, revision control, repository Open Service Interface Definitions (OSID), repository for academic publishing, or software repository [76].

A *software repository* is a storage location from which software packages may be retrieved and installed on a computer. Many software publishers and other organizations maintain servers on the Internet for this purpose, either free of charge or for a subscription fee. Repositories may be solely for particular programs, such as CPAN for the PERL programming language, or for an entire operating system. Operators of such repositories typically provide a package management system, tools intended to search for, install and otherwise manipulate software packages from the repositories. For example, many Linux distributions use *Advanced Packaging Tool* or *yum* [77].

A special case of a repository is the *meta data repository*, which is a database, which is used to administrate meta data. These meta data provide a basis of highly interactive systems, including all necessary descriptions of the system and its environment. By means of such meta data repositories, systems can react flexible to changes without programming effort [76].

An XML *repository* is a repository for storing and querying collections of XML documents. XML can be used to represent the source code of various programming languages, too, such as PROLOG, PHP or JAVA. We use an XML repository in order to store the source code of the programming language, which we want to analyze. We parse the source code into an abstract syntax tree and we transform it into an XML representation. Then, we store the parsed XML representation of the source code into an XML repository. Using an XML repository has the advantage, that we can use the same methods of the repository for different programming languages.

There exists several possibilities to represent rules in XML, e.g., RULEML[40, 41], SWRL [57, 58] or PROLOGML (cf. Section 3.1.3). All these possibilities are modeling the PROLOG term structure in a similar way using different tags.

RULEML (*Rule Markup Language*) is a markup language developed to express both forward and backward rules in XML for deduction, rewriting, and further inferential-transformational tasks. It is defined by the *Rule Markup Initiative*, an open network of individuals and groups from both industry and academia that was formed to develop a canonical web language for rules using XML markup and transformations from and to other rule standards/systems.

SWRL (*Semantic Web Rule Language*) is a proposal for a semantic web rules-language, combining sublanguages of the OWL (*Web Ontology Language*, a markup language for publishing and sharing data using ontologies on the World Wide Web [36, 37]) with those of RULEML.

In order to manage, update and query the XML representation in PROLOG, we use the XML query and transforming language PL4XML (cf. Section 3.1.3).

Chapter 2

Software Engineering and PROLOG

PROLOG was one of the first logic programming languages, and remains among the most popular such languages today, with many free and commercial implementations available. It was first conceived by a group around Alain Colmerauer in Marseille, France, in the early 1970s, while the first compiler was written by David H. D. Warren in Edinburgh, Scotland. Having its roots in formal logic, and unlike many other programming languages, PROLOG is *declarative*: the program logic is expressed in terms of relations, and execution is triggered by running queries over these relations. Often, PROLOG is associated with artificial intelligence and computational linguistics. While initially aimed at natural language processing, the language has since then stretched far into other areas like theorem proving, expert systems, games, automated answering systems, ontologies and sophisticated control systems. It is particularly useful for database, symbolic mathematics, and language parsing applications. Modern PROLOG environments support the creation of graphical user interfaces, as well as administrative and networked applications.

For this work, we use SWI-PROLOG. SWI-PROLOG offers a comprehensive *free software* PROLOG *environment*, licensed under the *Lesser GNU Public License*. Together with its graphics toolkit XPCE, its development started in 1987 and has been driven by the needs for *real-world applications*. SWI-PROLOG is robust. and has excellent development facilities. It supports a graphical debugging environment and a range of libraries that allows to implement GUIs, use object-orientation, use modules, implement a multi-threaded HTTP server or client, TCP/IP sockets and many other functionalities. It scales well for large applications. SWI-PROLOG supports multi-threading, is well-maintained and is available for all major platforms. The big advantage of SWI-PROLOG is the friendly development environment: the graphical debugger is indispensable, and even the command-line interface offers some goodies like a help system, command completion and command history.

SWI-PROLOG is widely used in research and education as well as for commercial applications. But in the field of software engineering, PROLOG is only scarcely used [38, 61, 70].

2.1 Integrated Development Environments for PROLOG

IDEs support interactions between methods and users in an easy, graphical way. Since the logic programming community is comparatively small, only few tools exist for comfortably programming and for analyzing source code in PROLOG. SWI-PROLOG by Jan Wielemaker probably has the most comprehensive PROLOG IDE.

SWI-PROLOG

SWI-PROLOG contains some tools in order to debug and analyze software. Using this software we can search for failures in PROLOG source code using a graphical environment. The following describes some parts of SWI-PROLOG.

MANPCE is one part of the SWI-PROLOG IDE (cf. Figure 2.1), which opens

> "...the graphical gateway to the PCE environment and the PCE manual. It contains four pull-down menus and a field for (short) general feedback messages." [61, 63]

Using the IDE `manpce`, we open the SWI-PROLOG *help*, view demo programs, open diverse browsers and other useful tools, e.g., the PROLOG *emacs editor*, a *thread monitor*, a *graphical tracer*, an *event viewer* or the *Prolog Navigator*.

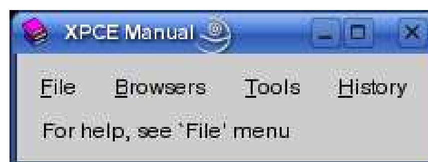


Figure 2.1: XPCE Manual

The PROLOG Navigator of SWI-PROLOG provides an explorer-like view of a directory holding PROLOG source code files. PROLOG source code files can be expanded, showing their contents. The screen-dump of the *Prolog Navigator* (cf. Figure 2.2) shows the hierarchy structure of the file system. Menus provide consulting and editing files, and access to the manual and debugger.

GXREF is a *cross-referencer* (cf. Figure 2.3), which operates on the currently loaded files. Therefore, it must be run after loading an application. After loading `gxref`, the dependencies window displays a graphical overview of dependencies between *files*. Unfortunately, it is not possible to sum up several files to packages. There is no possibility to scale dependency views.

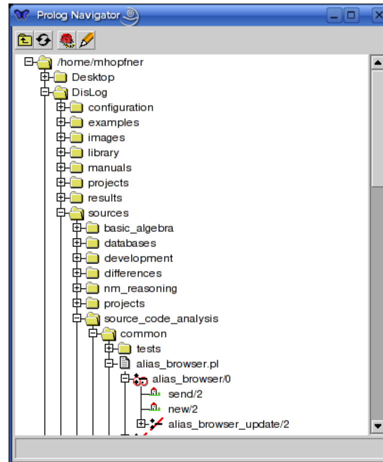


Figure 2.2: Prolog Navigator

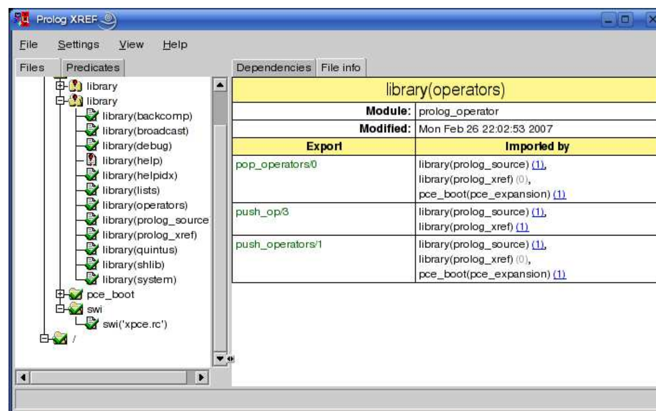


Figure 2.3: Cross referencer: gxref

Visual PROLOG

Another IDE for the MS operating systems is *Visual Prolog* (cf. Figure 2.4). One of the features of *Visual Prolog* is an IDE, which is fully object oriented, and which is based on logical programming with horn clauses.

> "Visual PROLOG is a logical programming language that counts PROLOG Development Center (PDC), and Turbo PROLOG as predecessors. The goal of Visual PROLOG is to support industrial strength programming of complex knowledge emphasized problems . . . " [60].

Some features of the *Visual Prolog* IDE are, e.g., logical programming concepts (backtracking, pattern matching), algebraic data types, multi-threading, unicode support, automatic memory management (garbage collection), a tree representation of modules, a

text editor, a dialog editor, a menu editor, which creates pull-down and pop-up menus, a help generator tool, extra library support, COM support, ISAPI support, multi-threading, ODBC, Sockets, pipes and many extra GUI controls.

Figure 2.4 shows the part of the Visual PROLOG IDE, which is used to generate GUIs. Corresponding to the well known developer suites of Borland, Microsoft, ... (cf. Section 1.1), there exist buttons for simple graphical elements, which can be put on a user defined GUI.

Figure 2.4: Visual Prolog Studio Shell

CIDER: The Curry Integrated Development EnviRonment

CIDER is a graphical programming and development environment for the multi-paradigm declarative language Curry. Curry is a universal programming language aiming at amalgamating the most important declarative programming paradigms, namely functional programming and logic programming. Moreover, it also covers the most important operational principles developed in the area of integrated functional logic languages: *residuation* and *narrowing*.

CIDER is intended to be a platform for integrating various tools for analyzing and debugging Curry programs. Currently, CIDER consists of

- a program editor with the usual functionality,
- various tools for analyzing properties of functions in Curry programs (types, overlapping definitions, complete definitions, non-determinism, operational completeness, dependencies etc),
- a tool for drawing dependency graphs,
- a graphical debugger, i.e., a visualization of the evaluation of expressions.

Figure 2.5 shows the main window after starting CIDER and loading a program. The main window in the middle is an editor window for editing the current program. On the left- and right-hand side, there is a list of the top-level functions in the current file and a list of the currently available analysis tools, respectively. After selecting a function and an analysis in the corresponding list boxes, the function is analyzed and the result is either shown in the bottom window (if it is a textual result) or, if it is a graph, it is visualized [15, 16].

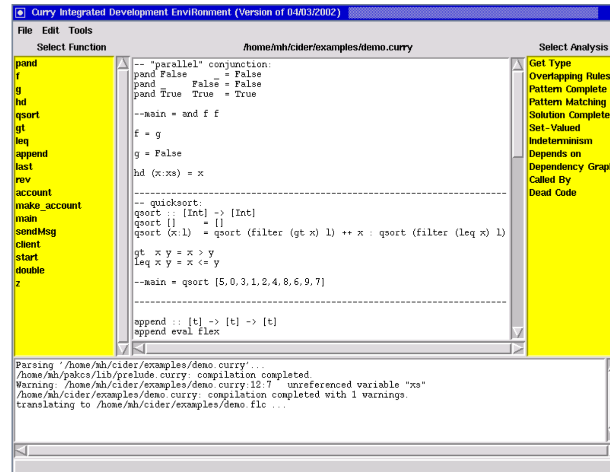


Figure 2.5: CIDER

ViPReSS: the PROLOG Refactoring Browser

ViPReSS is a PROLOG refactoring browser [45], which is designed by Tom Schrijvers and Alexander Serebrenik. It is construed for SICStus PROLOG and it integrates refactorings for PROLOG programs into the VIM editor. Some of the features of ViPReSS are: computing the call graph of a source file, replacing 'cut' with 'if-then-else', extracting predicates locally, finding dead, non-exported code, finding dead project code, and reordering 'if-then-else' branches.

Tools of the DISLOG DEVELOPERS' KIT (DDK)

The DISLOG DEVELOPERS' KIT (DDK) is developed by Dietmar Seipel. The functionality of the DDK ranges from reasoning in disjunctive deductive databases to applications such as the management and visualization of stock information [50, 52, 55].

It contains several tools in order to refactor PROLOG source code, too. We report about three tools, which have been implemented in diploma thesis in the past few years.

ECLIPSE Plug-in Dian Dochev has implemented an Eclipse plug-in (cf. Figure 2.6) for PROLOG in his diploma thesis [10, 17]. He has defined an interface between PROLOG and the JAVA environment ECLIPSE. Then, he has analyzed PROLOG source code, using a model driven architecture (MDA, cf. Section 1.1), in which he has defined some of the PROLOG refactoring methods mentioned in Section 2.2.

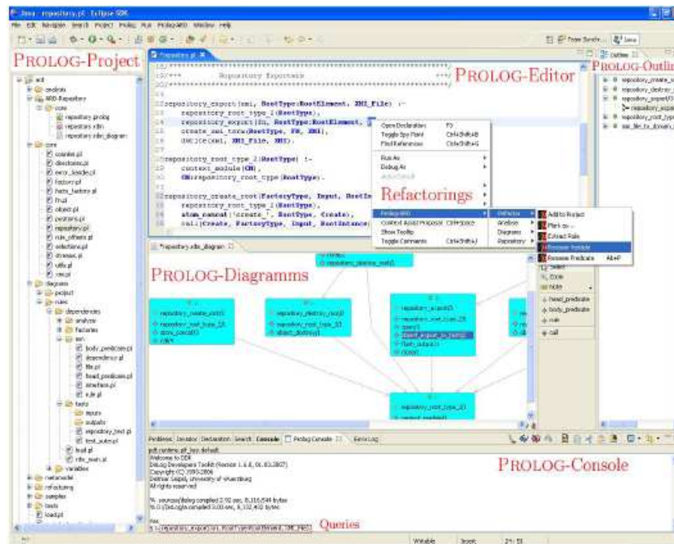


Figure 2.6: PROLOG Plug-in for Eclipse

VISUR/RAR Marbod Hopfner has implemented the tool VISUR/RAR in his diploma thesis [20]. This tool visualizes call dependencies between methods and manages rules in deductive databases. SCAV (Source Code Analysis and Visualization), which has been designed in this dissertation, is the continuation of VISUR/RAR. In comparison to the previous system, SCAV has been completely redesigned, enhanced and extended. E.g., calculation speed has been increased. RAR has been split into the software repository PROSORE and the library RAR. PROSORE stores the data in XML. PROLOGML has been defined in order to represent PROLOG in XML. We have added the graph exchange library GXL. Now, all important configurations are written in XML. Slicing, visualizations, reasoning and retrieval are largely extended and more configurable and further analysis methods and statistical information about the source code are available.

PROLOG Refactoring Browser Michael Müller has implemented a tool called PROLOG *refactoring browser* (cf. Figure 2.7) in his diploma thesis [35]. He has examined and implemented some of the refactorings mentioned in Section 2.2. Thereby, he has defined, e.g., *design patterns* in order to find and replace repeated code fragments in the source code by a common method. Using the refactoring browser, he marks methods containing redundant code and replaces these methods by an abstract method. Using

an abstract method in order to combine several methods decreases the number of clones in the source code. Another way to combine several methods is to use a method with additional parameters.

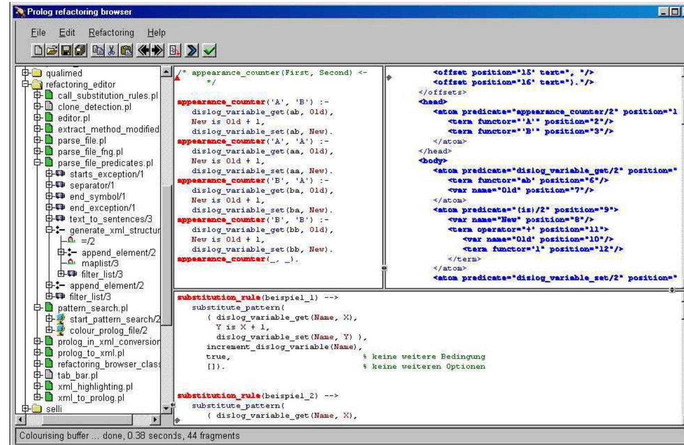


Figure 2.7: PROLOG Refactoring Browser

Michael Müller has defined an extended XML representation of PROLOG source code, too, by extending the DTD of PROLOGML. Using the extended PROLOGML representation, it is possible to represent the complete layout of PROLOG source code and to write the XML representation of PROLOG source code back to a PROLOG file. Thereby, even the comments are included.

In comparison to the theses of Dian Dochev and Michael Müller, which both make *Predicate Scope Refactorings* and *Clause Scope Refactorings*, we analyze in this thesis the *backbone* of a PROLOG system. We analyze the architecture of the whole system and make a graph based analysis for subsequent refactoring. These analyses are useful for *System Scope Refactorings* and *Module Scope Refactorings*. We make the macro-analyses, whereby Dian Dochev and Michael Müller make the micro-analyses.

2.2 PROLOG Refactorings

Refactoring source code means to modify the source code in order to make it better readable and to simplify its structure. At the same time, the functionality is *not* changed. We can understand refactoring as *cleaning up*. Automatic tests should exist and assure that the source code does not change its behavior. Martin Fowler, the leading refactoring personality, who is one of the first persons making refactorings, suggests about 100 possible refactorings in order to make source code more transparent. He created a list of good refactorings, which he describes in detail [13]. He defines refactoring in the following way:

> Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a *refactoring*) does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring [13].

Tom Schrijvers, Alexander Serebrenik, and Bart Demoen give a catalog of refactorings for PROLOG programs [45, 46, 47]. They say, refactoring is a *source-to-source program transformation* that changes program structure and organization, but not program functionality. The major aim of refactoring is to improve readability, maintainability and extensibility of the existing software. It is important to find the right parts of the source code and the best time to make refactorings. Both is decided by the programmer itself. The following items are according to [46] the most frequently mentioned problems with which software developers have to deal.

System Scope Refactorings The system scope encompasses the entire code base. The user does not transform a particular subpart; he affects the entire system as a whole. An example is dead code detection and removal. The deadness property can in general only be inferred with the entire system in scope. This scope is evidently appropriate for all programming paradigms. It might even apply to multi-language systems:

- extract common code into predicates,
- hide predicates (remove them from export lists),
- remove dead code,
- remove duplicate predicates,
- remove redundant arguments,
- rename functor.

Module Scope Refactorings The module scope considers a particular module. Usually a module is implementing a well-defined functionality and is typically arranged as one file. For example, a module can be renamed, split into separate modules or several modules can be merged together. This scope corresponds to refactorings operating on classes in JAVA or C++:

- merge modules,
- remove dead intra-module code,
- rename module,
- split module.

Predicate Scope Refactorings The predicate scope targets a single predicate. The counterparts of predicates in PROLOG are methods in JAVA or C++. For example, the moving of a predicate to a different module, affects the predicate directly. The code that depends on the predicate may need updating as well. But this is considered an implication of the refactoring of which either the user is alerted or the necessary transformations are performed implicitly:

- add argument,
- reorder arguments,
- move predicate,
- rename predicate.

Clause Scope Refactorings The clause scope affects a single clause in a predicate. Usually, this does not affect any code outside of the clause directly. An example is when a part of the clause body is replaced by a call to a new predicate that consists of the original goal. Code outside of the clause scope is affected in as much as that the new predicate is introduced and the necessary import declarations are added or removed. In object-oriented and functional programming languages, this scope affects methods and function bodies respectively:

- extract predicate locally,
- invert *if-then-else* (negate condition and reorder branches),
- replace *cut* by *if-then-else*,
- replace unification by (in)equality test.

Tool support for refactoring is highly desirable because checking the preconditions for a given refactoring often requires nontrivial program analyses, and applying the transformations may affect many locations in the program.

Part II

Management and Analysis of Source Code

This part deals with the management of PROLOG source code in a repository, and with the analysis, refactoring and slicing of the source code. Figure 1 shows the architecture of SCAV. The basic methods of PROSORE are used by RAR to reason about source code, and they are used by VISUR to generate visualizations.

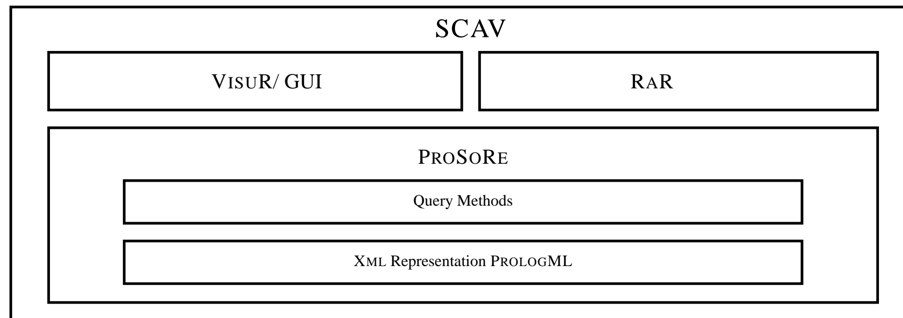


Figure 1: Architecture of SCAV

Call Dependencies In order to structure and comprehend the design of source code, we consider the hierarchy of the source code and the call spectra of packages. The call dependencies of predicates play an important role, because we can use them to decide to which package a predicate can be assigned. To obtain a transparent structure of the source code, we can group predicates of the same scope. Using the PROSORE database and the methods of PROSORE, we can retrieve call dependencies between predicates. For example, the following call determines the calls of the predicate `writeln/1`:

```
?- A = (user:writeln)/1,
   rar:calls(predicate:A, predicate:B).

B = (user: write)/1 ;
B = (user:nl)/0

Yes
```

We can detect the calls of meta-call predicates, such as `checklist/2`, `maplist/3`, or `findall/3`, calling further predicates, too.

```
?- A = (user:writeln_list)/1,
   rar:calls(predicate:A, predicate:B).

B = (user:checklist)/1 ;
B = (user: writeln)/1 ;

Yes
```

Call dependencies can help to determine undefined predicates, dead code or cross references between packages. Having the predicate call dependencies and knowing in which

file a predicate is defined, we are able to determine the cross references between files, modules, and, in general, packages. Figure 2 visualizes the call dependencies between the units of the DDK. Edges between units which represent only very few calls have been faded out.

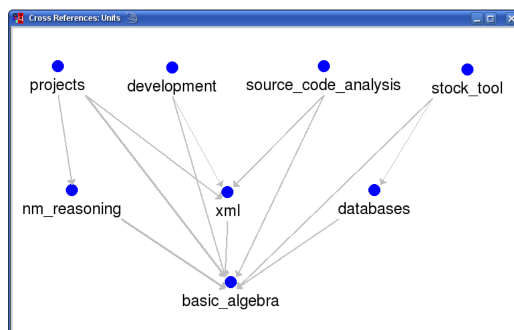


Figure 2: Cross Reference Graph

Call Spectra contain information about the calls of predicates or packages. We can either determine which packages a predicate is calling, or we can determine the calls between one package and the other packages. Two predicates having similar call spectra are considered to belong to the same package.

The spectra help us to organize the source code of a project. We try to refactor the source code into a hierarchical structure. Each package belongs to a certain scope, collecting all predicates of the same scope. We minimize calls between basic packages and higher level packages. Calls from higher level predicates to basic predicates are favored. If there exist undesired calls, we try to reorganize the corresponding predicates into other packages. It is not always possible to get a complete hierarchical structure, because there exist predicates, which are called and defined in each package, e.g., `dportray/2` or `test/2`.

Figure 3 shows the spectrum of the unit `basic_algebra`. Each bar shows the number of calls to another unit. The 4 units on level 1 and 2 are basic units, and the 4 units on level 3 are higher level units.

Slicing Using the predicate call dependencies, we can slice source code. The intention is to retrieve a fully working part of the source code, which is restricted to a certain part of the whole functionality. This makes it easier to refactor source code, to search for bugs in the source code, or to integrate a certain functionality into another project.

Figure 4 shows a report which summarizes the result of a slice. One column shows the number of predicates of a package, which are necessary for the slice; another column shows the number of predicates, which are chosen. As we always choose complete files, the number of chosen predicates always is greater or equal to the number of necessary predicates.

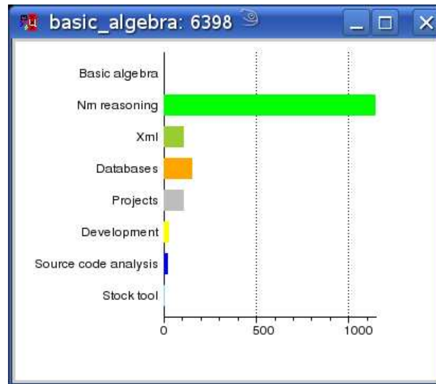


Figure 3: Calls of the Unit basic_algebra

Percentage Use of Predicates

Name				Statistic				
System	Sources	Unit	Module	LoC	Predicates	Choosen	Necessary	Percentage
DisLog				137098	9575	904	294	3,1
	sources			135023	9390	833	276	2,9
		basic_algebra		18929	1645	384	59	3,8
			basics	4897	435	259	43	9,9
			programs_dislog	2245	221	79	6	2,7
			utilities	3209	252	46	10	4
		nm_reasoning		14171	1254	20	7	0,6
			nmr_interfaces	3178	241	20	7	2,9

Figure 4: Percentage of Use of the Predicates in each Unit and Module of the Sliced Predicate slice/3

Software Repository To manage source code, we have developed the PROLOG software repository PROSORE. It consists of the PROSORE database which stores the source code in facts, a method to parse PROLOG source code, and methods to query and update the PROSORE database. Each fact of the PROSORE database represents a PROLOG rule in the XML language PROLOGML. In order to query and update PROLOGML, we use the powerful XML library PL4XML of the DDK [50, 52, 55].

So far, the analysis is mainly tailored to PROLOG. Nevertheless, we can use the PROSORE database to store the XML representation of other languages like JAML, and PHPML, too. Thereby, either the corresponding XML representation of the language has to be transformed into a suitable XML representation, which can be used together with the existing query methods, or the language specific XML representation is used and the basic query methods are adapted.

We present some case studies to demonstrate the usage of RAR. We introduce adapted query methods for JAVA source code, and we show, how to extract design information from JAVA which is represented in JAML [12] and how to extract design information from PHP which is represented in PHPML [14]. Moreover, we describe how to detect and refactor PROLOG design patterns. E.g., we search for the directly recursive predicates which can be replaced by the meta-call predicate `maplist/3` [53]. A statistics shows

the result of the directly recursive predicates found in the source code (cf. Figure 5).

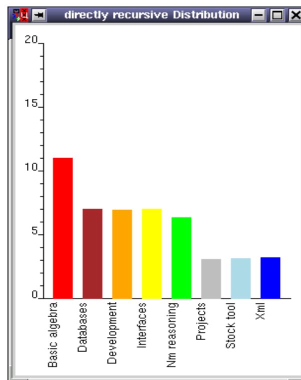


Figure 5: Rules with Directly Recursive Predicates (in %)

Chapter 3

The Software Repository PROSORE

We have developed the PROLOG tool SCAV (Source Code Analysis and Visualization), which is integrated in the DDK [50, 52, 55]. SCAV consists of the repository PROSORE, and the libraries VISUR (Visualization of Rules), RAR (Reasoning about Rules) and GXL (Graph eXchange Language). The XML processing is done using PL4XML (Programming Language for XML, or PROLOG for XML) [49, 51]. Each library of SCAV is responsible for a certain scope. SCAV provides a graphical user interface (GUI), for the repository PROSORE and the libraries VISUR, RAR, and GXL.

The software repository PROSORE uses an XML representation of the source code, which is saved using PROLOG facts. Together with a corresponding DML and a query language these facts constitute the database of the software repository PROSORE. The software repository PROSORE supports basic methods for reasoning about the source code. The tool is very flexible; it can also be applied to XML representations of other programming languages, e.g., of C++, JAVA, or PHP, which can be imported into the repository, if there exists an appropriate XML representation of the source code. E.g., we import JAVA source code using JAML [12], or PHP using PHPML [14].

The library PL4XML is a declarative XML query, transformation and update language, which is implemented in and fully interleaved with the logic programming environment of SWI-PROLOG. It is part of the DDK and it is implemented by Dietmar Seipel [49, 51]. The library RAR supports complex methods, for reasoning about source code and for generating statistical information about source code. The library VISUR provides methods for creating reports such as tables, graphs, polar diagrams, and histograms. We visualize call dependencies or cross references, predicate memberships, and statistics. The PROLOG library GXL makes the common Graph eXchange Language accessible to PROLOG. GXL is designed to offer the possibility of using diverse graph tools by importing or exporting graphs between these tools. Thus, each user can choose a tool with the best functionality for his purposes. VISUR computes various graphs in GXL format. This may be entity relationships, rule goal graphs, or cross reference graphs, which show the dependencies between several units, modules, or files. All of our transformation and visualization methods are based on GXL.

The combination of the repository PROSORE, and the libraries VISUR, RAR, GXL

and PL4Xml is called SCAV. SCAV can be used for code inspection, code analysis, design pattern detection, refactoring and visualization of source code. We retrieve information about the architecture of source code. Visualizing the dependencies between packages of a given hierarchy structure supports us in reorganizing the source code. Analyzing the given structure, we can detect cycles in cross references of packages and we can rearrange methods in the source code in order to retrieve a hierarchical structure of the packages, each package being of a certain scope.

In order to test source code, or to export the necessary parts of the source code, including a certain functionality, we isolate the necessary parts of the source code; in other words, we extract a *program slice* of the source code. This slice includes all necessary methods, global variables, and tests. Considering just a small part of the source code makes it sometimes easier to find bugs in the source code.

We calculate statistics, determine coherences between *units*, *modules*, *files*, and *predicates* and we visualize these information in tables, trees, graphs, histograms or polar diagrams.

The software repository ProSoRe consists of a transformation from Prolog source code to the Xml representation PrologML, the ProSoRe database, and multiple query methods (cf. Figure 3.1). Via PL4Xml of the Ddk, we access and query the Xml representation in the repository.

The Xml documents are defined using Dtds. We decided using Dtds instead of Xml schemas, because the defined Xml documents are not complex. In our case, using Dtds is completely sufficient in order to describe the structure of an Xml document.

The repository just contains a view of the source code, which we want to analyze. The parsed Prolog files do not belong to the repository. The content of the repository is not automatically updated, if a parsed file is changed in the file system. Only, if we execute the update process manually after a file has been changed, input files and repository are consistent, again.

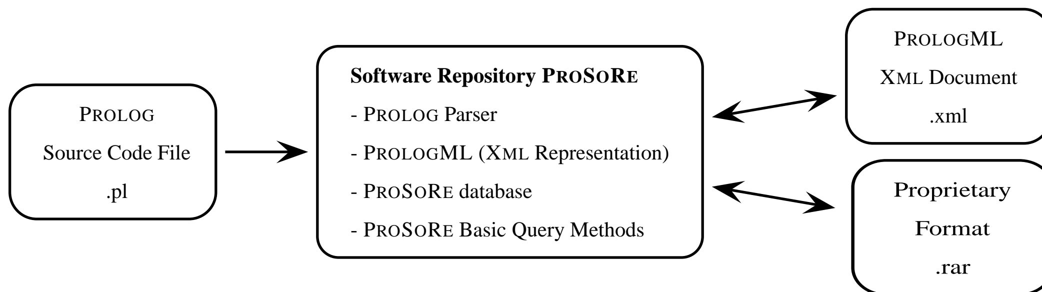


Figure 3.1: Software Repository ProSoRe

We use the software repository ProSoRe, e.g., for the following software analysis:

- to determine the files, which contain rules of a certain predicate,
- to determine the predicates, which call a certain predicate and vice versa,
- to classify predicates, files, modules or units into certain groups,

- to determine dependencies between files, modules, units, or packages
- to create statistics, e.g., about cross reference calls, about the frequency of occurrence of certain predicates, about unnecessary predicates, about dead code,
- to calculate, e.g., a slice of a predicate. A slice of a predicate consists of all files, containing all necessary predicates and variable settings, which are needed to run this predicate.

As it is also possible to write the XML representation back to PROLOG [35], it is conceivable to use refactoring methods directly on the XML representation, and write the result back to the original PROLOG source code. In order to analyze the source code of other languages, we can change the parser, which reads the source code and generates an XML representation.

In Section 3.1, we first describe PROLOGML, which is used to represent PROLOG source code in XML. Then, we describe the XML data managing library PL4XML. Finally, Section 3.2 describes the PROSORE database.

3.1 Representation of Source Code in PROLOGML

PROLOGML represents PROLOG source code in an XML document. It allows to represent disjunctive formulas in the head of a rule. Thereby, the top-disjunction of the rule head is represented as a list, just like the top-conjunction of the rule body. Further disjunctions and conjunctions, which are nested in the head or body are marked with special tags. This XML representation of PROLOG source code is about five times larger as the original source code.

In PROLOG, we use the term representation called field notation to represent XML documents. The structure of the field notation is similar to an XML document, and it can easily transformed into an XML document and vice versa. We always talk about an XML document, although we are not using an XML document in PROLOG.

PROLOGML represents the whole source code of all files in *one* single XML document. It consists of three main parts (cf. Figure 3.2): the *program*, including the *source code*, the *source tree*, and the *exported predicates*. The structure of the corresponding XML document contains the three main elements

```
<source_tree>, <exported_predicates>, <program>.
```

The XML document looks as follows:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<rar>
  <program>...</program>
  <source_tree>...</source_tree>
  <exported_predicates>...</exported_predicates>
</rar>
```

The DTD of this part of the XML document looks as follows:

DTD:

```
<!ELEMENT rar source_tree exported_predicates program>
```

The element `<program>` contains the rules and directives of the PROLOG files. The element `<source_tree>` contains the *hierarchy tree* of the system. It stores the file paths and associates the files to diverse sub-hierarchies. In the DDK, the hierarchy tree contains the levels `system`, `sources`, `unit`, `module`, and `file`. At last, the element `<exported_predicates>` contains the names of the exported predicates of the SWI-PROLOG modules. In the following, we explain each part of this XML document in detail.

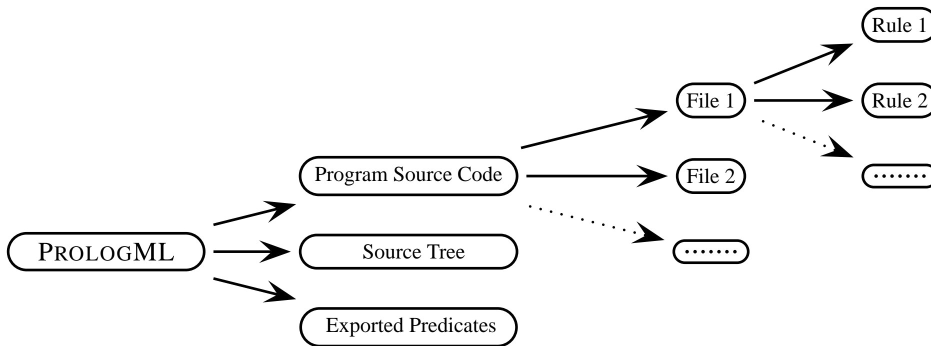


Figure 3.2: The Structure of PROLOGML

3.1.1 Source Code

The element `<program>` contains the parsed *source code*, which we want to examine. The element consists of multiple `<file>` elements. The content of a parsed file – the predicate rules – are stored in these `<file>` elements. Each `<file>` element contains `<rule>` elements, and the `<rule>` elements contain the PROLOG rules. In the following, we show an example of the XML structure of the element `<program>`. Then, we give the corresponding DTD and at last, we explain how we represent PROLOG rules in XML.

Example 3.1 (Program Section) The `<program>` section of PROLOGML contains multiple `<file>` elements. The content of each parsed file – the rules – is written into an own `<file>` element.

```
<program>
  <file name="..." path="..." date="...">
    <rule ...>...</rule>
    ...
  </file>
  ...
</program>
```

The DTD of the element <program> looks as follows:

DTD:

```
<!ELEMENT program (file*)>

<!ELEMENT file (rule*)>
<!ATTLIST file
   name CDATA #required
   path CDATA #required
   date CDATA #implied
   mouse_click CDATA #implied>
```

A <file> element consists of <rule> elements, and the attributes name, path, date, and mouse_click. The attribute name contains the name of the file. path contains the path to the file. The path begins relative to the root, which we specify in the root attribute of the *root element* of the hierarchy tree. The attribute mouse_click is a reference to an XML configuration document, which is used to configure the visualization of graphs, and which is passed as an additional parameter to the corresponding methods. The attribute mouse_click will be explained in Appendix B.2 in detail.

The element <file> is not to be mixed with the element <file> of the hierarchy tree. In the hierarchy tree, it is used to locate files on the physical storage device, and in <program>, it is used to store the content of a file. In the following, we describe how we store the content of a file – the rules – in PROLOGML.

Rules in PROLOGML In PROLOGML, each rule is represented by a <rule> element. Each <rule> element has the attribute module and the optional attribute operator. The <rule> element contains a <head> element and a <body> element, which are both optional. The <head> element contains <atom> elements and the <body> element contains <atom> and <formula> elements. An <atom> element contains the attributes module, predicate, and arity. A <formula> element contains the attribute junctor. <atom> and <formula> elements contain <term> and <var> elements. Additionally, the <formula> element contains <atom> elements, too (cf. Figure 3.3). Thus, the DTD looks as follows:

DTD:

```
<!ELEMENT rule head? body?>
<!ATTLIST rule
   module CDATA "user"
   operator CDATA implied>

<!ELEMENT head atom*>
```

```
<!ELEMENT body atom* formula*>

<!ELEMENT atom term* var*>
<!ATTLIST atom
   module CDATA "user"
   predicate CDATA #required
   arity CDATA #required>

<!ELEMENT formula atom* term* var*>
<!ATTLIST formula
   junctor CDATA #required>

<!ELEMENT term term* var*>
<!ATTLIST term
   functor CDATA #required>

<!ELEMENT var EMPTY>
<!ATTLIST var
   name CDATA #required>
```

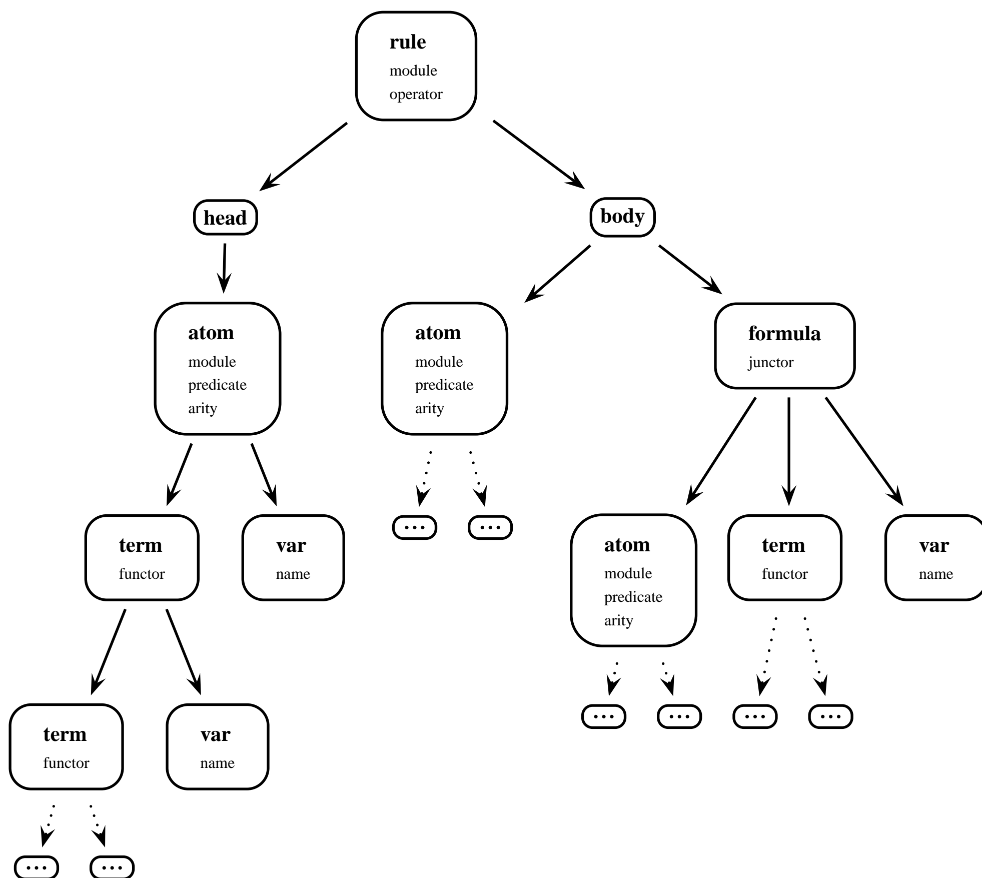


Figure 3.3: The DTD of the Element <rule>

The attributes of the element `<rule>`:

`module`
: determines the module name, in which the rule is defined.

`operator`
: contains the operator of the rule. The default value is "`:-`".

The attributes of the element `<atom>` are:

`module`
: contains the module name of the predicate. If the predicate in the rule has no module name, the module name is added. The module name is either derived from the list of *exported predicates* (cf. Section 3.1.2), or it is the default module name `user`.

`predicate`
: contains the name of the predicate.

`arity`
: contains the arity of the predicate.

The element `<formula>` contains the attribute `junctor`, which determines the name of the junctor. The element `<term>` contains the attribute `functor`, which determines the name of the functor. Constants are represented by the element `<term>`, too.

Example 3.2 (Term Element)

```
<term functor="1"/>
```

The element `<var>` contains the attribute `name`, which determines the name of a variable. In the following examples, small PROLOG methods are given and we show, how these methods are represented in PROLOGML.

Example 3.3 (PROLOG in XML) Assume that the PROLOG file `increment.pl` contains the following rule:

```
increment(X, Y) :-
   Goal = add(1, X, Y),
   call(Goal).
```

If the file is not configured as a module, then the rule belongs to the default module `user`. In PROLOGML this rule looks as follows:

```
<rule module="user" operator=":-">
  <head>
    <atom module="user" predicate="increment" arity="2">
      <var name="X"/>
      <var name="Y"/>
    </atom>
  </head>
```

```
  <body>
    <atom module="user" predicate="=" arity="2">
      <var name="Goal"/>
      <term functor="add">
        <term functor="1"/>
        <var name="X"/>
        <var name="Y"/>
      </term>
   </atom>
   <atom module="user" predicate="call" arity="1">
     <var name="Goal"/>
   </atom>
  </body>
</rule>
```

Example 3.4 (PROLOG in XML) Assume that the PROLOG file `delete.pl` contains the following rule:

```
delete(Xs, E, Ys) :-
   findall( X,
      ( member(X, Xs),
        not( E = X ) ),
      Ys ).
```

The file is not configured as a module, too. Therefore, the rule belongs to the default module `user`. In PROLOGML this rule look as follows:

```
<rule module="user" operator=":-">
  <head>
    <atom module="user" predicate="delete" arity="3">
      <var name="Xs"/>
      <var name="E"/>
      <var name="Ys"/>
    </atom>
  </head>
  <body>
    <atom arity="3" module="user" predicate="findall">
      <var name="X"/>
      <term functor=",">
        <term functor="member">
          <var name="X"/>
          <var name="Xs"/>
        </term>
        <term functor="not">
          <term functor="=">
            <var name="E"/>
            <var name="X"/>
          </term>
        </term>
      </term>
      <var name="Ys"/>
```

```
    </atom>
  </body>
</rule>
```

Example 3.5 (PROLOG in XML) Assume that the PROLOG file `list.pl` contains the following rule:

```
list_info(Ys) :-
   ( Ys = [],
     writeln('Empty List.') )
   ;
   writeln('Full List.') )
```

Again, the file is not configured as a module. The rule belongs to the default module `user`. In PROLOGML this rule looks as follows:

```
<rule module="user" operator=":-">
  <head>
    <atom module="user" predicate="list_info" arity="1">
      <var name="Ys"/>
    </atom>
  </head>
  <body>
    <formula junctor="or">
      <atom arity="2" module="user" predicate=",">
        <term functor="=">
          <var name="Ys"/>
          <term functor="[]"/>
        </term>
        <term functor="writeln">
          <term functor="Empty List."/>
        </term>
      </atom>
      <atom arity="1" module="user" predicate="writeln">
        <term functor="Full List."/>
      </atom>
    </formula>
  </body>
</rule>
```

PROLOGML to PROLOG

The call

```
prologml_to_prolog(+PrologML)
```

writes the rules contained in PROLOGML back into PROLOG files. The method above calls iteratively the method `xml_to_prolog/2` for each represented file contained in PROLOGML. Using this method, all comments are lost and the layout of the source code is changed by rewriting the source code to PROLOG files.

Michael Müller has refined the XML representation PROLOGML and the method `xml_to_prolog/2` [35]. He keeps the comments and the layout of the original source code, because his parser inserts layout positions and comments into the XML structure, which can be rewritten back using his method `xml_to_prolog/2`.

3.1.2 Hierarchy Tree and Exported Predicates

In order to locate the source code files, we use the *hierarchy tree* which contains the paths and filenames of the source code files. The hierarchy tree is an XML document. In the hierarchy tree, we do not only store paths and filenames, we group files together, too, independent of their real location in the file system.

In SWI-PROLOG, there exists the possibility to define PROLOG *modules*. All predicates which are defined in a module are encapsulated. They only can be called outside the module with leading module name, except predicates which are *exported*.

The Hierarchy Tree

Having the hierarchy tree, we call a method which extracts all contained filenames of the hierarchy tree. Then, we parse the extracted files and transform the content to PROLOGML. The original XML hierarchy tree is completed with some further attributes, e.g., the attribute `date`. This attribute contains the file date and is used to organize updates. The completed hierarchy tree is added to PROLOGML (cf. Figure 3.4).

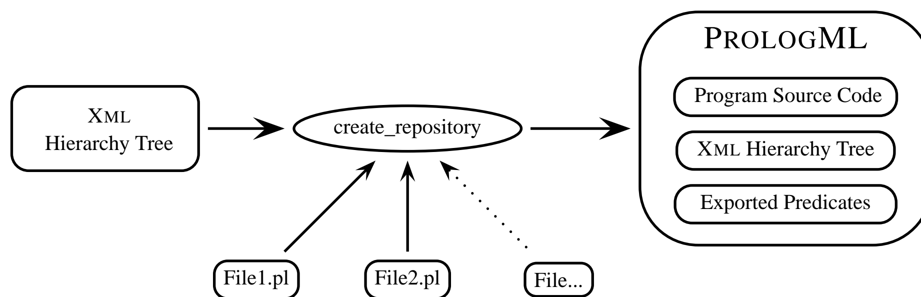


Figure 3.4: Usage of the Hierarchy Tree

Structure of the Hierarchy Tree In PROLOGML, the hierarchy tree is given by the element `<source_tree>`. It contains a *virtual* hierarchy of all files of the system. The hierarchy tree does not have to be identical with the file system hierarchy. It can be an arbitrary hierarchy, too. The DTD of the hierarchy tree looks as follows:

DTD:

```
<!ELEMENT source_tree level* file*>

<!ELEMENT level sub_level* file*>
<!ATTLIST level
   name CDATA #implied
   path CDATA #required
   root CDATA #required
   xml_root CDATA #implied
   lowest_level CDATA #required
   source_code dislog|prolog|php|java #required
   alias CDATA #implied
   open CDATA #implied
   close CDATA #implied
   mouse_click CDATA #implied>

<!ELEMENT sub_level sub_level* file*>
<!ATTLIST sub_level
   path CDATA #required
   name CDATA #implied
   alias CDATA #implied
   open CDATA #implied
   close CDATA #implied
   mouse_click CDATA #implied>

<!ELEMENT file sub_level*>
<!ATTLIST file
   name CDATA #implied
   date CDATA #implied
   path CDATA #required
   consulted_from CDATA #implied
   alias CDATA #implied
   open CDATA #implied
   close CDATA #implied
   mouse_click CDATA #implied>
```

Each element of the hierarchy tree can have further nested elements. The hierarchy tree can have any depth.

Except for the element `file`, the tag names are arbitrary. It can be any name, e.g., using the hierarchy of the DDK, the tag names are `system`, `sources`, `unit`, `module`, `file`, and potentially `predicate`.

We append each contained predicate of a file to the corresponding branch of the hierarchy tree. Therefore, we use the element `<predicate>`, containing the attribute `name`. We just do this in order to be able to view the corresponding predicates defined in a file, e.g., in the hierarchy browser of the GUI of SCAV.

Each element can have further attributes, too, e.g., to specify the mouse click actions or pop-up menus.

The following example shows a part of the DDK hierarchy tree. Later, we explain how to generate the hierarchy tree of the DDK, automatically.

Example 3.6 (DDK Hierarchy Tree)

```
<source_tree>
  <system name="DisLog" source_code="dislog"
    path="dislog" root="/home/DisLog/">
    <sources name="sources" path="sources">
      <unit name="basic_algebra"
        path="sources/basic_algebra">
        <module name="basics"
          path="sources/basic_algebra/basics">
          <file name="increment.pl"
            path="sources/basic_algebra/basics/increment.pl"/>
          <file .../>
      ...
  </system>
</source_tree>
```

The used attributes of the elements in the hierarchy tree have the following meaning:

`name`
: Each element can have the attribute `name`. The value of this attribute will be displayed in the hierarchy browser (cf. Appendix A.4). If this attribute is missing, we display the tag name of the corresponding element in the hierarchy browser.

`path`
: Each element has the attribute `path`. This is a unique ID of each element in the tree, which is used by some methods.

  The attribute `path` in the element `<file>` determines the relative path of a file in the file system. We get the absolute filename of a file, if we concatenate the value of the attribute `root` (which only is an attribute of the root element in the hierarchy tree) with this relative path.

  The absolute filename is needed in order to find the location of a file, e.g., if we generate the `<program>` section of PROLOGML or if we want to show the content of a file in an editor.

`root`
: This attribute has to be defined in the root element of the hierarchy tree. It completes the relative path of the source code files (attribute `path` of the elements `<file>`) to an absolute path.

`xml_root`
: If we want to insert JAVA or PHP source code into the software repository, we need corresponding XML documents, which represent the source code. These documents can be JAML or PHPML documents.

  Like the attribute `root`, the attribute `xml_root` determines the prefix of the JAML or PHPML files. This attribute has to be defined in the root element of the hierarchy. The XML files will only be used by JAVA or PHP source code. Using PROLOG

source code, the original source code files are parsed and transformed into the PROLOGML representation by the internal parser.

`lowest_level`
The lowest level in the hierarchy tree contains the name of the leaf level. Sometimes, we add to the leaves further levels. Then we have to know, which was the original lowest level. For the DDK, the lowest level is `file` and we add levels with the name `predicate`.

`source_code`
This attribute has to be defined in the first level of the hierarchy. It tells the parser the language of the source code files. Currently, we can parse PROLOG source code files and XML representations of JAVA (JAML [12]), or PHP (PHPML [14]). Correspondingly, the value of the attribute `source_code` is `java`, `phpml`, or `dislog`.

`consulted_from`
While generating PROLOGML, we check if the currently parsed file consults further files. We add all consulted files automatically to the hierarchy tree. In PROLOGML, we mark the consulted files with the attribute `consulted_from`, containing the relative filename of the original file, from which the file is consulted.

`alias`
This attribute is a reference to an element of an additional XML configuration document. This additional configuration document is passed to some methods. It is used to configure, e.g., the visualization of the hierarchy tree in a hierarchy browser. Depending on the reference, the visualization of the corresponding element has, e.g., a certain thumbnail in the browser beside the item. Files, which are consulted from another file, automatically get the value `alias=consulted_file`. We describe the corresponding XML configuration file in Appendix A.4.

`mouse_click`
This attribute is a reference to an element of the additional XML configuration document mentioned above. Depending on this reference, we configure the action, if we click in the hierarchy browser on a text item.

`open`, `close`
The attributes `open` and `close` are references to the attribute `alias` of the element `<icon>` in the XML configuration document. These attributes define the icon of an expanded and collapsed node in the visualization of the hierarchy tree. The attribute `open` defines the icon of a node in the tree, which is expanded, and the attribute `close` defines the icon of a node in the tree, which is collapsed.

`date`
After generating PROLOGML, we add the attribute `date` to the elements `<file>`.

This attribute contains the date of the last modification of the file. It is needed for observing updates of the file and updating an existing PROLOGML representation. The date and time of the file are retrieved using the PROLOG built-in methods `time_file/2` and `convert_time/2`. The formatted date value looks as follows:

```
date="Mon Aug 23 13:01:37 2004"
```

The hierarchy tree can be visualized in the hierarchy browser of SCAV. We can use the hierarchy tree to select certain sub-hierarchies and to execute certain methods on a part of the PROLOGML representation. These actions can be configured in the corresponding XML configuration document of SCAV (cf. Section 7.1).

Generating a Hierarchy Tree There exist several ways for generating a hierarchy tree.

- In order to map the subdirectories and the contained files of a given directory `Dir` in a hierarchy tree, we call

  ```
  file_system_to_xml(+Dir, -Tree_1).
  ```

  This method is based on the file system interaction of SWI-PROLOG.

  We have to prepare the result, such that we can use it together with SCAV. We add the attribute `root` to the root element of the hierarchy tree. This attribute has the value of the starting directory `Dir`. Therefore, we call

  ```
  filter_redundant_paths(Tree_1, Tree_2).
  ```

- The hierarchy tree of the DDK can be computed automatically calling

  ```
  dislog_sources_to_tree(-Tree).
  ```

  Using this method, we generate the hierarchy tree of the DDK using the facts of the predicates `dislog_unit/2` and `dislog_module/3`. These facts contain the unit and module structure of the DDK and all files, which belong to the DDK and which will be consulted, when we start the DDK.

  ```
  dislog_unit(?Unit, ?Modules).
  dislog_module(?Module, ?Path, ?Files).
  ```

  The consultation order is kept, too. We retrieve the path of the DDK by calling

  ```
  dislog_variable_get(home, -DDK_Path).
  ```

Just as described above, we have to call `filter_redundant_paths/2`, too, in order to retrieve a hierarchy tree which can be used together with the parser. The parser transforms PROLOG source code into the XML representation PROLOGML. We describe the parser in Subsection 3.2.2.

- In general, we call

```
sources_to_tree(+Mode, +Root, -Tree), or
sources_to_tree(+Mode, +Root, +XML_Root, -Tree),
```

in order to generate a hierarchy tree of an arbitrary directory.

`Mode`
can be `dislog`, `prolog`, `java`, or `php`. For PROLOG, we parse the original source code files. For JAVA, we need JAML [12] and for PHP, we need PHPML [14]. JAML is an XML representation of JAVA source code files, and PHPML is an XML representation of PHP source code files.

`Root`
is the directory, containing the source files. The path to the files will be stored relative to the value of `Root`.

`XML_Root`
contains the root path of the corresponding XML files of the source code files. Can be omitted, if there exists no corresponding XML files.

`Tree`
is the XML tree of the source code files.

Using `sources_to_tree/4`, the resulting tree can be used directly together with the parser. We do not need to call the method `filter_redundant_paths/2` before.

The Global Variable of the Hierarchy Tree The hierarchy tree is saved into a global PROLOG variable and can be retrieved after the source code has been parsed. In order to retrieve the hierarchy tree , we call

```
sca_variable_get(source_tree, Source_Tree).
```

The Exported Predicates of PROLOG Modules

In SWI-PROLOG, there exists the possibility to define PROLOG *modules*. In order to define a module, the *module definition* has to be the first directive of the file. Each module only can be defined in *one* file.

All predicates which are defined in a module are encapsulated. They only can be called outside the module with leading module name, except predicates which are *exported*. Exported predicates can be called without leading module name, although they are defined

in a module. The predicates have to be exported in the corresponding module, in which they are defined. Predicates, which are already defined and exported in another module, cannot be exported a second time.

Example 3.7 (Header of a PROLOG File) The following example shows the header of a PROLOG file, which defines the module `rar`. The predicate `xml_file_to_db/1` is exported. Calling this predicate, we do not need to write the module name in front of the predicate name.

```
:- module( rar, [ xml_file_to_db/1 ]).
```

If we want to call a non-exported method `xml_file_to_db/2`, we have to put the module name `rar` in front:

```
?- xml_file_to_db('file.pl').

Yes

?- rar:xml_file_to_db('file.pl', Exp_Preds).

Yes
```

We need the list of exported module predicates to determine the module affiliation of a predicate, if it is called without its module prefix. Otherwise, it would be allocated to the default module `user`, which all other predicates, which are not defined in a module, belong to. The element `<exported_predicates>` contains a list of the exported predicates of each SWI-PROLOG module.

```
<exported_predicates>
  <module name="rar" file="/.../rar.pl">
    <atom module="rar" predicate="xml_file_to_db" arity=1"/>
    ...
  </module>
  ...
</exported_predicates>
```

The corresponding DTD looks as follows:

DTD:

```
<!ELEMENT exported_predicates (module*)>

<!ELEMENT module (atom*)>
<!ATTLIST module
   name CDATA #required
   file CDATA #required>

<!ELEMENT atom EMPTY>
<!ATTLIST atom
   module CDATA #required
   predicate CDATA #required
   arity CDATA #required>
```

The Global Variable of the Exported Predicates The list of exported predicates is saved into a global PROLOG variable and can be retrieved after the source code has been parsed. In order to retrieve the list of exported predicates, we call

```
sca_variable_get(exported_predicates, -Exp_Preds).
```

3.1.3 Management of PROLOGML using PL4XML

We use semi-structured data – such as XML data – as method-input and output, and as representation of the examined source code. In order to represent and manage the semi structured data, we use PL4XML.

PL4XML is a declarative XML query, transformation and update language [49, 51], which is implemented in and fully interleaved with the logic programming environment of SWI-PROLOG. It is designed and developed by Dietmar Seipel and has been continuously improved and extended in the last years. It is part of the DISLOG DEVELOPERS' KIT. The acronym PL4XML can be read as

- programming language for XML, or
- PROLOG for XML.

PL4XML uses two alternative representations for XML data in PROLOG, the field notation and the graph notation. It consists of the two modules `fn_query` and `gn_query` within the unit `xml` of the DDK. In this work, we only use the module `fn_query`. This module is divided into the following components, which form the sub-language FNQUERY: FNPATH, FNSELECT, FNTRANSFORM, and FNUPDATE.

PL4XML supports mechanism, which facilitate the easy access of the attribute values. Reading, writing and updates are supported. The library PL4XML has been used in other projects, too. For example, in [18] it is shown how mathematical knowledge in MATHML can be managed nicely using PL4XML. We use PL4XML, in order to represent and query the semi-structured data in this work. In the following we summarize elementary facts of PL4XML. For detailed information about PL4XML we refer to [49].

Syntactically, both XML and PROLOG are based on nested term structures. The following subsection shows how XML structures can be represented and queried in PROLOG. Since XML can be used for representing complex objects, this gives the way of handling complex objects in PROLOG or deductive databases.

Complex Objects in PROLOG The field notation consists of *association lists* for representing lists of attribute/value pairs. This data structure is familiar to LISP programmers. An association list $[a_1 : v_1, \ldots, a_n : v_n]$ uses the infix operator ":" for pairing attributes a_i with their associated values v_i (when a_i and v_i are PROLOG terms).

For example,

```
[module:user, predicate:increment, arity:2]
```

is an association list. Association lists are PROLOG terms themselves; thus, it is possible to have nested association lists, where some of the values v_i are association lists. E.g.,

```
[atom:[module:user, predicate:increment, arity:2]]
```

is an association list with the attribute `atom`, whose the corresponding value is itself an association list.

Association lists have got several advantages when compared to ordinary PROLOG facts: Firstly, the sequence of attribute/value pairs $a_i : v_i$ is arbitrary. Secondly, values v_i can be accessed by attributes a_i rather than by argument positions. Thirdly, the database schema can be changed at run time. Fourth, null values can be omitted, and new values can be added at run time.

The following section introduces a PROLOG representation – called field notation – of XML elements that represents the attribute/value pairs and the sub-elements as association lists. This section shows that XML elements in field notation can be queried and modified very elegantly.

XML Objects in PROLOG The following XML example shows the head of a PROLOG rule. The head consists of the element `<atom>`, which contains the attributes `module`, `predicate`, and `arity`. Furthermore, the `<atom>` element contains two elements `<var>`, each containing the attribute `name`.

Example 3.8 (Head of a PROLOG Rule)

```
<head>
  <atom module="user" predicate="increment" arity="2">
    <var name="U"/>
    <var name="V"/>
  </atom>
</head>
```

An XML element

```
<T a1="v1" ... an="vn">...</T>
```

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. Thereby, `T:As:C` is called an FN-triple. For example, the XML element above can be represented by the following FN-triple:

```
head:[]:[
  atom:[module:user, predicate:increment, arity:2]:[
    var:[name:'U']:[],
    var:[name:'V']:[] ] ]
```

Definition (Field Notation):

1. If `T` and `C` are PROLOG terms and `As` is an association list, then the PROLOG term `O = T:As:C` is called an FN-triple with the tag `T` and the contents `C`.
   - If $As = [a_1 : v_1, ..., a_n : v_n]$, then each a_i is called an attribute of O and v_i is called the corresponding value.
   - If $As = [\,]$, then O can alternatively be represented as $T : C$.
2. Given an FN-triple $O = T : As : C$, such that $C = [c_1, ..., c_m]$ is a list of FN-triples, then each c_i is called a sub-element of O.

In the following, the abbreviation `Head` for a frequently needed FN-triple contains the head of a rule with one entry, which is given by the XML element of the example above.

Selection Given an FN-triple `O` and two PROLOG terms `A` and `X`. The binary infix-predicate ":=" allows for accessing the sub-elements and the attribute values of `O`.

- The call `X := O/A` computes all most general substitutions θ, such that `X`θ is a sub-element with the tag `A`θ of `O`θ. Equivalently, it is possible to write `X := O^A`.
- The call `X := O@A` computes all most general substitutions θ, such that `X`θ is the value of the attribute `A`θ of `O`θ.

The predicate "`:=`" can handle *complex path expressions* for selecting sub-components.

Example 3.9 (Selection)

```
?- X := Head/atom, Y := Head/atom@predicate.

X = atom:[module:user, predicate:increment, arity:2]:[
          var:[name:'U']:[], var:[name:'V']:[]]
Y = increment

Yes
```

It is also possible to use *variables* as selectors. Then, all possible selection paths (using child or attribute selectors) can be computed:

```
?- X := Head/atom/S.

X = var:[name:'U']:[]
S = var;

X = var:[name:'V']:[]
S = var;

X = 'U'
S = var@name;
```

```
X = 'V'
S = var@name

Yes
```

Example 3.10 (Selection) Now, the three rules of the examples 3.3, 3.4, and 3.5 are added to the XML document `XML`. We want to find the rules in the XML document, which contain a disjunction in the body.

```
?- X := XML/rule::[/body/formula@junctor=or].

X = rule:[module:user]:[
  head:[]:[
    atom:[module:user, predicate:list_info, arity:1]:[
      var:[name:'Ys']:[] ] ],
  body:[]:[
    formula:[junctor:'or']:[
      atom:[arity:2, module:user, predicate: (',')]:[
        term:[arity:2, functor: (=), module:user]:[
          var:[name:'Ys']:[],
          term:[functor:[]]:[] ],
        term:[arity:1, functor:writeln, module:user]:[
          term:[functor:'Empty List.']:[] ] ],
      atom:[arity:1, module:user, predicate:writeln]:[
        term:[functor:'Full List.']:[] ] ] ] ]

Yes
```

Example 3.11 (Selection) Again, the three rules of the examples 3.3, 3.4, and 3.5 are added to the XML document `XML`. We want to find the rules in the XML document, which have at least arity 3.

```
?- X := XML/rule::[/head/atom@arity > 2].

rule:[module:user]:[
  head:[]:[atom:[module:user, predicate:delete, arity:3]:[
    var:[name:'Xs']:[],
    var:[name:'E']:[],
    var:[name:'Ys']:[] ] ],
  body:[]:[...]:[...]]

Yes
```

Assignment It is possible to assign new values to attributes or elements. The following statement updates the `predicate` attribute of the `<atom>` element of the `<head>` element `Head`.

```
?- X := Head*[/atom@predicate:'inc'].

X = atom:[module:user, predicate:inc, arity:2]:[
      var:[name:'U']:[],
      var:[name:'V']:[]]

Yes
```

`A` is obtained by first selecting the sub-element with the tag `atom`, and by then modifying the `predicate` attribute. The following statement inserts a new sub-element into the selected `<atom>` element:

```
?- X := Head/atom*[/var:[name:'W']:[]].

X = atom:[module:user, predicate:increment, arity:2]:[
       var:[name:'U']:[],
       var:[name:'V']:[],
       var:[name:'W']:[] ]

Yes
```

Updates in Elements The following query selects an `<var>` element containing the attribute `name = 'U'` and modifies the value of `name` to `'W'`:

```
?- X := Head/atom *
      [/var::[@name='U']@name:'W'].

X = atom:[module:user, predicate:increment, arity:2]:[
       var:[name:'W']:[],
       var:[name:'V']:[] ]

Yes
```

The following query selects the `<atom>` element with the attribute

```
predicate = increment
```

and modifies the attribute `name` in all its `<var>` elements to `W`:

```
?- X := Head/atom::[@predicate=increment]*[/var@name:'W'].

atom:[module:user, predicate:increment, arity:2]:[
   var:[name:'W']:[],
   var:[name:'W']:[]]

Yes
```

The following query adds the attribute assignment `time="10:00"` to all `<atom>` elements of all `<var>` elements with attribute `name = 'V'`:

```
?- X := Head/atom*[/var::[@name='V']@time:'10:00'].

X = atom:[module:user, predicate:increment, arity:2]:[
       var:[name:'U']:[],
       var:[name:'V', time:'10:00']:[]]

Yes
```

3.2 The PROSORE Database

The PROSORE database is a mixture of PROLOG facts and an XML representations of rules. Thereby, each fact stores a part of PROLOGML, namely the rules. The PROLOG source code files are not part of the software repository PROSORE. They are just the input of the repository and are not continuously consistent with the repository.

All methods and facts described in the following sections belong to the PROLOG module `rar`, and therefore, using these methods, we have to write the module name `rar` in front of the methods, separated by a colon. In general, this looks as follows:

```
Predicate = Module:Predicate_Name(...).
```

Thereby, `Module` is the module name `rar`, and `Predicate` is the called method. In the following, we omit the module name in order to shorten long lines.

In Subsection 3.2.1, we describe the PROSORE database, which is an indexed datastructure of PROLOGML. The indexes allow fast access to the rules. Subsection 3.2.2 describes the basic methods to access the PROSORE database, and how to create, reset, save and load the PROSORE database. Subsection 3.2.3 describes how to update the PROSORE database, Susbsection 3.2.4 explains how to retrieve call dependencies between predicates and cross references between files, Subsection 3.2.5 introduces the hierarchy of the DDK, and at last, we describe some special predicate groups and meta-call predicates in Subsection 3.2.6

3.2.1 The Structure

Unfortunately, XML documents in field notation are not accessible in a fast way. In order to search for a certain element, we have to go through the whole document. To ensure a fast access of the rules and directives represented in PROLOGML, the PROSORE Database splits the XML representation into rules.

In the PROSORE database, each represented rule and directive contained in PROLOGML is extracted and stored as PROLOG fact, which is indexed by predicate name, by filename and by the called predicates. So, this internal representation of PROLOGML in the software repository is a mixture of indexed PROLOG facts. Thus, we can quickly search for

- all rules defining a given predicate,

- all rules in a given file, or
- predicates calling a certain predicate and vice versa.

The whole PROSORE database consists of the *facts* mentioned above and some further *global variables*. The global variables contain the hierarchy tree and the list of *exported module predicates*. Both are stored using field notation, too.

All facts and variables of the PROSORE database belong to a certain namespace. This enables us to use several sets of the PROSORE database at the same time by using different namespaces.

In the following subsections, we describe the PROSORE database in detail. First, we describe the *global variables* of the PROSORE database, and the PROSORE database *facts*. Then, we explain how to use the PROSORE database *namespaces* and at last, we compare the PROSORE database with an alternative possibility of data storage.

The Hierarchy Tree and the Exported Predicates

There is nearly no change in the storage of the hierarchy tree or the exported predicates compared with PROLOGML. These two parts of PROLOGML are just extracted from PROLOGML and saved in the two global variables:

```
source_tree, exported_predicates.
```

The PROSORE database variable `source_tree` stores the hierarchy tree of the system. It is the same XML document, which we describe on page 44; the hierarchy tree is written and saved in field notation.

The PROSORE database variable `exported_predicates` stores the predicates, which are exported from PROLOG modules. It is the same XML document, which we describe on page 49; it is written and saved in field notation, too. Both PROSORE database variables are defined globally and can be accessed by the method calls

```
sca_variable_get(+Variable, -Value).
sca_variable_set(+Variable, +Value).
```

Using these methods, we pay attention to the current namespace (cf. page 61) of the PROSORE database, too.

The PROSORE Database Facts

The facts of the PROSORE database are stored in the module `rar`. Although, we sometimes suppress the module name `rar` in order to shorten long lines, we always have to put the module name `rar` in front of each database fact. Otherwise, we cannot access the PROSORE database.

The facts are divided into three groups with the predicates

```
rar:rule/6, rar:is_called_by/6, rar:leaf_to_path/4.
```

Only the facts `rule/6` are extracted from PROLOGML. The other facts are calculated and only exist in the PROSORE database.

Using SWI-PROLOG, facts can be accessed very fast. We can accelerate the access of facts, if we index them. In order to index a fact, we use the SWI-PROLOG predicate `index/1`, which can index up to four parameters of a fact.

Example 3.12 (Indexing) In order to index the first and second parameter of `rule/6` facts, we write:

```
:- index(rar:rule(1, 1, 0, 0, 0, 0)).
```

Each indexed parameter should be an atom. It should not be a *term* or a *string*. Otherwise, we slow down searching for a certain fact. We need a fast index for the filenames and the predicates. But each filename is a string and each predicate consists of the term

```
Predicate = (Module:Predicate_Name)/Arity.
```

In order to access the facts in a fast way – even though the important parameters consists of terms and strings – we additionally calculate hash values for the predicate name (first parameter) and the filename (second parameter) and use these hash values as first and second parameter of a fact. The third an fourth parameter of the fact are the original values, then. The hash values are calculated using the PROLOG built-in method `hash_term/2`.

The rule/6 facts store the source code (the `<program>` section of PROLOGML) in fragments. The source code of the examined system is split into predicate rules. Each rule is represented in field notation and each field notation is stored in a fact of the PROSORE database. So, each rule is contained in a separate fact. Additionally, each fact has indices, in order to access the definition of an predicate quickly. The facts have the following arguments:

```
rule(Hash_1, Hash_2, Predicate, File, Rule, Namespace).
```

`Hash_1`, `Hash_2`
: these arguments are hash values of the arguments `Predicate` and `File`, respectively, which accelerate the access to the rule head and the access to the filename of the file, in which the rule is defined. Using the DML methods, the corresponding hash values are calculated automatically. Usually, the user does not get in touch with the hash values.

`Predicate`
: is the head predicate of the rule in `(M:P)/A` notation: `M` is the module name, `P` is the predicate name and `A` is the arity of the head predicate.

`File`
: is the name of the file in which the predicate is defined.

`Rule`
is the XML representation of the rule, in field notation.

`Namespace`
is the namespace of the current PROSORE database (cf. page 61).

Remark: In PROLOG, *directives* have no head atom. Directives start directly with the operator "`:-`" or "`?-`". PROLOG directives are represented in PROLOGML like rules, except, that the head of the rule has no atom. In order to access rules without head predicate in the PROSORE Database, each rule without head predicate is indexed by the predicate name `no_head_` with a consecutive number as suffix (third parameter of the fact `rule/6`). In order to differ between directives and rules in the PROSORE Database, we just compare the head of the rule in PROLOGML and its index. We retrieve all directives of a file calling

```
file_to_directives(+File, -Directives).
```

The is_called_by/6 facts build an inverted search index for quickly finding all predicates, which are called by a given predicate. There exists several `is_called_by` facts for one `rule` fact, depending on the number of called predicates of a rule.

After a `rule` fact is asserted, we assert for each called predicate of the rule one `is_called_by` fact to the PROSORE database.

Implementation:

```
1  insert_rule(File_1, P_1, Rule) :-
2     rar:insert(rule, P_1, File_1, Rule),
3     ( forall( rar:calls_pp(Rule, P_1, P_2),
4           rar:insert(is_called_by, P_2, File_1, P_1) ) ).
```

The method `insert_rule/3` first inserts a rule `Rule` of file `File_1` and with head predicate `P_1` to the PROSORE database (line [2]). Then, the method looks for all called predicates `P_2` in the rule `Rule` and insert for each called predicate `P_2` the inverse rule to the PROSORE database, too (lines [3-4]).

The `is_called_by/6` facts have the following arguments:

```
is_called_by(Hash_1, Hash_2,
   P_1, File_2, P_2, Namespace).
```

`Hash_1`, `Hash_2`
are hash values of the arguments `P_1` and `File_2`.

`P_1`, `File_2`, `P_2`
`P_1` is the name of a predicate, which is called from predicate `P_2` of the file `File_2`.

`Namespace`
is the namespace of the current PROSORE database (cf. page 61).

Example 3.13 (The Representation of a Rule in the PROSORE Database) The following predicate `p/0` is defined in the file `f.pl`:

```
p :-
  q1,
  q2.
```

This rule is represented in the PROSORE database by three facts:

```
rule(16652038, 33196, (user:p)/0, 'f.pl',
  rule:[module:user]:[
    head:[]:[
      atom:[module:user, predicate:p, arity:0]:[]],
    body:[]:[
      atom:[arity:0, module:user, predicate:q1]:[],
      atom:[arity:0, module:user, predicate:q2]:[]] ],
  standard).

is_called_by(16588069, 33196,
  (user:q1)/0, 'f.pl', (user:p)/0, standard).

is_called_by(16588037, 33196,
  (user:q2)/0, 'f.pl', (user:p)/0, standard).
```

The `is_called_by/6` facts differ in the first and third parameters.

The leaf_to_path/4 facts are used for accessing the hierarchy of the parsed source code quickly. The facts have the following arguments:

```
leaf_to_path(Hash, File, Path, Namespace).
```

`Hash`
: is a hash value of the argument `File`.

`File`
: is a leaf of the source tree, stored in the global variable `source_tree`.

`Path`
: contains the name of each element/hierarchy level on the path from the file to the root of the hierarchy tree.

Example 3.14 (The Content of the Argument Path)

```
<leaf_to_path level="file" name="algebraic.pl">
   <leaf_to_path level="module" name="basics">
      <leaf_to_path level="unit" name="basic_algebra">
      ...
      </leaf_to_path>
   </leaf_to_path>
</leaf_to_path>
```

The corresponding DTD looks as follows:

DTD:

```
<!ELEMENT leaf_to_path leaf_to_path?>
<!ATTLIST leaf_to_path
   level CDATA #required
   name CDATA #required>
```

The hierarchy in this tree is reversed to the hierarchy of the source tree. This means that the hierarchy begins with a leaf and ends with the root of the source path.

`Namespace`
is the namespace of the current PROSORE database (cf. page 61).

The Namespaces

The namespace of the facts and variables of the PROSORE database can be accessed and changed using

```
sca_namespace_get(-Namespace).
sca_namespace_set(+Namespace).
```

The PROSORE database facts and variables, which have different namespaces will not interact with each other. After PROLOG has been started, the initial namespace is always `standard`. If we want to use another database, we have to change the namespace before using one of the following elementary methods.

```
insert/3, insert/4,
replace/3, replace/4,
replace_or_insert/3, replace_or_insert/4,
update/3, update/4,
select/3, select/4,
delete/3, delete/4
sca_variable_set/2,
sca_variable_get/2,
```

The changed namespace will be active until it is changed again. If we reload a saved PROSORE database from a file, the data will be loaded into the currently, active namespace.

Analyzing the PROLOG Database

There exists another possibility to generate the database containing the source code. Instead of representing the source code in XML, we could consult the corresponding PROLOG files, which we want to analyze, into the current running PROLOG system. This has the following advantages:

- We can use the PROLOG internal built-in predicates to analyze the source code; e.g., in order to retrieve properties of predicates and clauses or dependencies between predicates, we could use the built-in methods

```
predicate_property/2,
clause_property/2,
clause/[2,3].
```

- Incremental updates are easier to manage, because we would work directly on the source code files without a further intermediate level.

But, consulting the source code into the PROLOG system has the following disadvantages:

- We have to consult the system, which we want to analyze into the same system, which makes the analysis; interferences of predicate symbols and namespaces could cause problems.

- Incorrect source code could prevent consulting the source code.

Using the PROSORE database for representing the source code has the following advantages:

- We do not have problems with names and namespaces between the running system and the analyzed source code.

- We can insert the source code of other languages to our database, e.g., we can use PHPML [14], representing PHP source code in XML, or JAML [12], representing JAVA source code in XML.

- The syntax of the analyzed source code does not need to be essentially correct; incorrect source code can be analyzed, too.

Considering the runtime, there is no advantage or disadvantage. Using the PROSORE database, we are in some operations slower, and in other operations faster than the method, which consults the source code into the PROLOG system.

3.2.2 Basic Access Methods

We have implemented basic methods to create, save and load the PROSORE Database. To select the facts of the PROSORE Database easily, we have implemented special selection methods.

Creating the Repository

The set of files which we want to insert to the PROSORE database has to be given in the *hierarchy tree*. Adding a large collection of PROLOG files to the PROSORE database can take some time. But the PROSORE database can be saved into a file, such that it can be quickly reloaded later. Given a hierarchy tree `Tree_1`, the method

```
create_repository(+Tree_1, -Tree_2)
```

creates the PROSORE database. If the source code, which we want to add, contains PROLOG, then the method executes the following steps in detail:

Implementation:

```
1   create_repository(T_1, T_3) :-
2      Type := T_1@source_code,
3      memberchk(Type, [prolog, dislog]),
4      retrieve_files_from_tree(T_1, Root, Existing_Files),
5      init_parsing(T_1, T_2),
6      parse_files(T_2, Root, Existing_Files, T_3).
7
8   retrieve_files_from_tree(T, Abs_Root, Existing_Files) :-
9      Rel_Root := T@root,
10     tilde_to_home_path(Rel_Root, Abs_Root),
11     tree_to_files(T, Relative_File_Names),
12     maplist( concat(Abs_Root),
13        Relative_File_Names, Absolute_File_Names),
14     sublist( exists_file,
15        Absolute_File_Names, Existing_Files ).
16
17  init_parsing(T_1, T_2) :-
18     reset_rar_database,
19     clear_tree(T_1, T_2),
20     alias_to_fn_triple(predicate_groups, PG),
21     assert_predicate_groups(PG).
22
23  parse_files(T_1, Root, Existing_Files, T_3) :-
24     maplist( program_file_to_xml,
25        Existing_Files, Xmls ),
26     sca_variable_get(exported_predicates, Exp_Preds),
27     checklist( xml_file_to_db(Root, Exp_Preds),
28        Xmls ),
29     rar:add_date_to_files(T_1, T_2),
30     rar:add_ps_to_file_content(T_2, T_3),
31     reverse_tree(T_3),
32     sca_variable_set(source_tree, T_3).
```

The following steps are executed in order to add PROLOG files to the PROSORE database (lines [1-6]):

- First, we verify the attribute `source_code` of the tree `T_1` and check, if the source code is PROLOG (lines [2-3]).

- Then, we retrieve all files contained in the tree. We get the common *file system root* of the files, too (line [4]).
- We initialize the system (line [5]), so that we can start parsing the files (line [6]).

In order to retrieve all files of the tree, we execute `retrieve_files_from_tree/3` (lines [8-15]):

- We get the root path of the source code. The root path is given in the hierarchy tree. We replace the *Unix* shortcut tilde "~", which is possibly in the root path, with the corresponding *home path* of the current user (lines [9-10]).
- Then we extract all relative filenames of the hierarchy tree (line [11]).
- The filenames are relative to the root path. We concatenate the root path and the relative filenames (lines [12-13]).
- At last, we check, if all files of the generated file list exist and delete none existing files of the file list (lines [14-15]).

In order to prepare the PROSORE database, so that we can insert new facts to the database, we call `init_parsing/2` (lines [17-21]):

- We reset the PROSORE database in order *not* to have old parts of a previously parsed source code. Thereby, we just retract the corresponding facts and global variables of the PROSORE database (line [18]).
- We clear the tree: we delete the content of each `<file>` element in the hierarchy tree, and we delete additional added `<file>` elements, which do not belong to the hierarchy tree (line [19]). These additional `<file>` elements have the attribute `alias:consulted_file`. The additional elements are added automatically to the hierarchy tree during parsing the corresponding files contained in the hierarchy tree before. So, if we parse the tree a second time, we first have to delete these entries.
- We get the predicate groups, which are saved in a file, and we assert them to have fast access to these groups (lines [20-21]).

At last, we parse each PROLOG file and insert the content into the PROSORE database (lines [23-32]):

- The content of each file is pre-parsed into a rudimentary XML structure. Additionally, an XML document of all exported predicates is generated. The corresponding field notation is stored in the variable `exported_predicates` (lines [24-26]).
- Now the rudimentary XML structure is parsed and the rules and directives are inserted into the PROSORE database (line [27]).
  - The rudimentary XML structure is parsed and transformed into the XML structure described in Section 3.1.1. Thereby, we also search for calls, which consult further files. We parse these additionally consulted files, too, and we add these files to the hierarchy tree. In the hierarchy tree, we add to the corresponding `<file>` elements three additional attributes:

    `alias, mouse_click, consulted_from`

  - The hash values for the indices of the fact `rule/6` are calculated and the fact `rule/6`, with the hash values, the predicate name, the file name, and the generated field notation of the rule is asserted to the PROSORE database.

  - The inverse call dependencies of the rule are determined and the facts `is_called_by/6` are added to the PROSORE database.

- For each parsed file of the hierarchy tree `T_1` the attribute `date` is added to the element `<file>`; the attribute `date` contains the file date (line [29]).

  Using this attribute, we identify the files, which are updated after generating the PROSORE database. Thus, we are able to make incremental updates.

- We add the corresponding `<predicate>` elements to the content of each element `<file>` of the hierarchy tree, in order to be able to visualize the corresponding predicates of a file in the hierarchy browser (line [30]).

- At last, an inverse tree index is generated and the extended hierarchy tree `T_3` is returned (lines [31-32]). We store the new hierarchy tree, charged with additional information in the global PROSORE database variable `source_tree` using

  ```
  sca_variable_set(source_tree, T_3).
  ```

During parsing the source code files, we print each name of the currently parsed file to the console.

Example 3.15 (File Parsing) If we just want to parse all PROLOG files contained in the directory and subdirectories of the directory `test_environment`, we call

```
?- rar:sources_to_tree(prolog, '~/test_environment', Tree_1),
   rar:parse_sources_to_rar_database(Tree_1, Tree_2).

Yes
```

And, if we just want to parse all files of the DDK, we call

```
?- rar:sources_to_tree(dislog, '~/DisLog', Tree_1),
   rar:parse_sources_to_rar_database(Tree_1, Tree_2).

Yes
```

Remark: If we want to parse only one source code file, we do not need the hierarchy tree. The filename is sufficient. We call

```
rar:file_to_db(+File),
```

in order to fill the PROSORE database with the content of the file `File`. We cannot use this method to parse more than one file using, e.g., `checklist/2`. Each time the method `file_to_db/1` is called, the PROSORE database is reset. The consequence is that files cannot be aggregated using this method. In Section 3.2.3 we describe an incremental update method for inserting additional rules of a file to the PROSORE database.

Resetting the PROSORE Database

In order to delete all content of the PROSORE database and to reset the global PROSORE database variables, we call

```
repository_reset.
```

Saving and Loading the PROSORE Database

The PROSORE database can be saved in two different formats.

- It is possible to save the PROSORE database as a listing of all PROLOG facts and global variables in a file; this is the proprietary format. Files saved in this format have the extension "`.rar`". This format can be saved and reloaded from the software repository PROSORE very quickly. Saving the PROSORE database in the `rar` format, the following information is stored in the file:
  - all `rule` facts in the format
    ```
    rule(Predicate, File, Rule),
    ```
  - all `is_called_by` facts in the format
    ```
    is_called_by(Predicate_1, File_2, Predicate_2),
    ```
  - all `leaf_to_path` facts in the format
    ```
    leaf_to_path(File, Path),
    ```
  - the global variable `source_tree` in the format
    ```
    sca_variable(source_tree, Tree),
    ```
  - and the global variable `exported_predicates` in the format
    ```
    sca_variable(exported_predicates, EP).
    ```
- The other possibility is to export the PROSORE database to PROLOGML and save it as XML document. Files saved in this format have the extension "`.xml`". The advantage of the XML format is that XML is a general standard, which is *human-readable*. The disadvantage is that saving or loading PROLOGML is about 10 times slower than saving or loading the proprietary PROSORE database format.

  Saving the PROSORE database in the XML format, only the `rule` facts, the global variables `source_tree` and `exported_predicates` are saved. The hash values of facts will not be saved. They will be restored by loading a saved database.

Importing PROLOGML from a file, the content is transferred into the PROSORE database format and the hash values of the PROSORE database will be reconstructed. Inverse indices are reconstructed, too, because these indices are not contained in PROLOGML.

The following calls are used for saving and loading a file `File` in the format `rar` or `xml`:

```
save_repository(+rar|xml, +File).
load_repository(+rar|xml, +File).
```

The following elementary methods, modify the PROSORE database, regardless, if the consistency of the PROSORE database is still guaranteed or not. Dependencies between the program rules and the reversed indices are not proven and adapted. In Section 3.2, we describe methods, which keep consistency.

Arguments First of all, we describe the *arguments* of the methods `select`, `insert`, `delete`, `replace` and `update`, because all of these methods use the same arguments. The methods look as follows:

```
Method_Name(+Type,  +Index, +Object).
Method_Name(+Type,  +Index_1, +Index_2, +Object).
```

`Type`
: can be `rule`, `is_called_by`, or `leaf_to_path`. The value of parameter `Type` refers to the corresponding facts of the PROSORE database. If `Type` has the value `rule`, we consider `rule` facts, if `Type` has the value `is_called_by`, we consider `is_called_by` facts and if `Type` has the value `leaf_to_path`, we consider `leaf_to_path` facts.

`Index`, `Index_1`, `Index_2`
: are indices, with which we can find a fact very quickly. Each index refers to a hash value, which enables a fast search of a given value. Thereby, `Index`, `Index_1` often is the name of a predicate, and `Index_2` is the corresponding filename.

`Object`
: contains the information, which we want to insert, e.g., a `Rule`.

Selection

We select facts with the methods

```
select(+Type,  ?Predicate, ?Object).
select(+Type,  ?Predicate, ?Filename, ?Object).
```

`select/[3, 4]` selects the fact of the PROSORE database, which corresponds with the argument `Type` and the parameters `Predicate`, or `Filename`. If there does not exists a proper fact, this call fails.

Example 3.16 (Selection) In order to select a rule of the PROSORE database we call

```
?- rar:select(rule, (user:increment)/2, 'increment.pl', Rule).

Yes
```

and if we want to retrieve the head predicates, which call `(user:increment)/2`, we call

```
?- rar:select(is_called_by, (user:increment)/2,
              'increment.pl', Head_Predicate).

Yes
```

Example 3.17 (Finding Meta-Call Predicates) We need a list of all meta-call predicates in order to slice a predicate correctly. If we want to find meta-call predicates, there exists several possibilities. One possibility is to search for a predicate, which

- is called in the body of a rule, and which
- contain at least one atom in the passed arguments.

We look, if there exists rules of the passed atoms. If this is the case, it is possible that the predicate calls a rule which has the head predicate of the passed atom. We suggest the predicate as a meta-call predicate.

Implementation:

```
1  find_possible_meta_predicate_1((M:P)/A) :-
2     rar:select(rule, _Predicate, _File, Rule),
3     Functor := Rule/body/atom::[
4        @module = M,
5        @predicate = P,
6        @arity = A ]/term@functor,
7     rar:select(rule, (_:Functor)/_, _, _).
```

Another possibility is to search for rules, which contain known meta-call predicates in the body, e.g., the meta-call predicate `call/1`. One of the arguments of the meta-call predicate has to be bound to a argument of the head predicate of the rule. Then, we suggest the head predicate as meta-call predicate, too.

Implementation:

```
1  find_possible_meta_predicate_2(P) :-
2     rar:select(rule, P, _File, Rule),
3     [user, call, 1] := Rule/body/atom@[
4        module, predicate, arity],
5     not( rar:special_predicate(P, _, _, _, _) ).
```

3.2.3 Updates

We have implemented methods which update the ProSoRe database by simultaneously keeping the consistency of the ProSoRe database. These predicates build on the elementary DML predicates described later. Using these methods, it is easy to change, and update the ProSoRe database. The dependencies between the facts

```
rule/6, is_called_by/6, leaf_to_path/4
```

and the hierarchy tree are kept consistent.

Insertion of Rules The following method is used to insert the rules of a certain file into the ProSoRe database. We have to pass a hierarchy branch, to which the rules of the file belongs to. If the path of the branch does not exist in the hierarchy tree, yet, it will be created. The rules of the file will be added to the ProSoRe database, even if the rules already exist.

```
insert_rules(+Hierarchy_Branch, +File).
```

The rules will be indexed with the filename `File`. The method computes and inserts all inverse rules, adds the filename to the corresponding branch of the hierarchy tree and determines and inserts the corresponding `leaf_to_path` facts.

Deletion of Rules The call

```
delete(+File)
```

deletes all rules and inverse rules, which are indexed with the filename `File` in the ProSoRe database. Thereby, it does not matter, if the original source code file exists, or has been changed in the meantime. This is ignored, because the original source code files are not a part of the software repository ProSoRe. All corresponding file entries in the branches of the hierarchy tree will be deleted, too.

Updating Rules After one or more files have changed, we incrementally update the ProSoRe database by calling

```
update_repository.
```

- First, this searches for the changed, the deleted and the newly added files by comparing the *file date* of the files written in the hierarchy tree with the *file date* of the files in the file system.
- All corresponding rules of the *deleted* and *changed* files will be deleted from the ProSoRe database. We delete the `is_called_by` and the `leaf_to_path` facts, too, so that the ProSoRe database is kept consistency.

- Then, the rules from the changed files are inserted to the PROSORE database.
- At last, the rules of the files, which are newly added to the hierarchy tree are inserted to the PROSORE database and the *file dates* of the hierarchy tree, which is stored in the global PROSORE database variable `source_tree` will be refreshed.

Elementary Update Methods

First, we describe two methods, which add new facts to the PROSORE database. Then, we describe two different methods in order to substitute facts, and at last, we describe how to delete facts of the PROSORE Database.

Insertion of Facts The following method always adds the fact, disregarding, if facts with the same indices are already existing. We insert a fact to the PROSORE database with the methods

```
insert(+Type,  +Predicate, +Object).
insert(+Type,  +Predicate, +Filename, +Object).
```

`insert/[3, 4]` asserts a fact to the PROSORE database, regardless, if there exists such a fact yet or not. If there exists such a fact or not is decided by the indices `Index`, `Index_1`, or `Index_2`. If we use `insert`, it is possible that there exist more than one fact with the same indices.

Example 3.18 (Insertion) In order to insert the predicate `(user:increment)/2` to the PROSORE database, we call

```
?- rar:insert(rule, (user:increment)/2, 'increment.pl', Rule).

Yes
```

whereas `Rule` contains a rule written in field notation, as we described above. We use a generic method, which asserts the facts

```
rule/6, is_called_by/6+, leaf_to_path/4
```

to the PROSORE database by using only one method:

Implementation:

```
1  insert(Type, V1, V2, Object) :-
2     sca_namespace_get(DB),
3     hash_term(V1, H1),
4     hash_term(V2, H2),
5     Fact =.. [Type, H1, H2, V1, V2, Object, DB],
6     assertz(rar:Fact).
```

Thereby, `Type` is either `rule`, `is_called_by`, or `leaf_to_path`. `V1` is the first indexed parameter, which is usually a predicate in `(M:P)/A` notation. `V2` is the second indexed parameter, which is usually a filename. We have implemented a method without the second parameter, too. `Object` is the payload of the fact. This is usually a rule in field notation.

- First, the method retrieves the current namespace (line [2]).
- Then, the method calculates corresponding hash values for the first and second passed parameters (lines [3-4]).
- All values are built to a fact (line [5]) with the predicate symbol `Type`.
- The fact is asserted to the PROSORE database (line [6]).

Insertion of Facts Conditional This method first proofs, if an fact with the same index or indices is already existing. If we insert facts with the methods

```
update(+Type,  +Predicate, +Object),
update(+Type,  +Predicate, +Filename, +Object),
```

we first check up, if a fact with the same indices exists yet. If there exists one fact with the same index/indices, nothing will be done. If there does not exists the fact, it will be asserted. In both cases the method-call will be true. We insert facts to the PROSORE database in dependency of already existing facts.

Substitution of Existing Facts The following just *substitutes* an *existing* fact, the next *substitutes* or *inserts* a fact.

We substitute facts with the methods

```
replace(+Type,  +Predicate, +Object),
replace(+Type,  +Predicate, +Filename, +Object).
```

`replace/[3, 4]` first checks up, if a fact with the same index/indices exists yet. If there exists no such fact, the call fails. If there exists one or more facts with the same index/indices, all of these existing facts will be retracted and the new fact will be asserted.

Example 3.19 (Replacing) In order to replace the rule `(user:increment)/2` in the PROSORE database, we call

```
?- rar:replace(rule,
      (user:increment)/2, 'increment.pl', New_Rule).

Yes
```

whereas `New_Rule` contains the new rule written in field notation.

Substitution or Insertion of Facts We *substitute or insert* facts with the methods

```
replace_or_insert(+Type, +Predicate, +Object).
replace_or_insert(+Type, +Predicate, +Filename,
   +Object).
```

`replace_or_insert` is similar to `replace`. First, it checks up, if a fact with the same index/indices exists yet. If there exists one or more facts with the same index/indices, they will be retracted and the new fact will be asserted. If there exist no such fact with the same indices yet, the new fact will be asserted. The difference to `replace` is that the call will *not fail*, if the fact does not exists, yet. The fact will be inserted, regardless if there exists such a fact or not.

Deletion We delete facts with the methods

```
delete(+Type,  +Predicate, ?Object),
delete(+Type,  ?Predicate, ?Filename, ?Object).
```

`delete/[3, 4]` retracts all facts of the chosen `Type` with the chosen indices and payload. In the second variant of `delete`, it is not allowed that at the same time, both indices are unbound. Either the index `Index_1`, or the index `Index_2` have to be bound.

3.2.4 Call Dependencies

In the following, we describe basic methods, with which we determine coherences between predicates, and files. Using these methods, we query the PROSORE database and we find out call dependencies between predicates, and cross references between files, or, in general, between packages.

In order to speed up requests, the PROSORE database stores PROLOGML internally in several facts using hash indices as mentioned in Section 3.2.1. We have implemented methods for querying the PROSORE database, taking advantage of this matter.

Call Dependencies between Predicates

The following methods of the module `rar` relate predicates and called predicates. One of the two last parameters always have to be bound:

```
calls(predicate:?Pred_1, predicate:?Pred_2).
calls_pp(+Rule, ?Pred_1, ?Pred_2).
```

The method `calls/2` selects a rule of the PROSORE database with the head `Pred_1`, and determines all called predicates in the body of the rule, even if they are called from a meta-call predicate. The format of `Pred_1`, `Pred_2` is

```
(Module:Predicate_Name)/Arity.
```

The method `calls_pp/3` is used in order to retrieve the calls of an already selected rule.

Implementation:

```
1  calls(predicate:?P_1, predicate:?P_2) :-
2     ground(P_1),
3     select(rule, P_1, _File, Rule),
4     calls_pp(Rule, P_1, P_2).
5
6  calls_pp(Rule, P_1, P_2) :-
7     ground(Rule),
8     Head := Rule/head/atom,
9     term_to_predicate(Head, P_1),
10    not( ignore_calls_pp(P_1) ),
11    Element := Rule/body/descendant::atom,
12    subelement_to_predicate(Element, P_2),
13    not( ignore_calls_pp(P_2) ).
```

First, we check, if `Rule` is ground (line [7]); otherwise, some operations of PL4XML do not work. Then, we retrieve the head `P_1` of the rule `Rule` (lines [8-9]). We check, if the head does not belong to a list of predicates, which we ignore permanently (line [10]). Afterwards, we query for the body element of the rule and retrieve the elements contained in the body element. We search for all predicates in the elements, including predicates called from meta-call predicates (lines [11-12]). At last, we check if the called predicate belongs to the list of ignored predicates (line [13]).

If the first parameter `P_1` is unbound, and the second parameter `P_2` is bound, we can search for calling predicates, too. Thereby, we use the inverse index of the facts `is_called_by`.

Implementation:

```
1  calls(predicate:P_1, predicate:P_2) :-
2     ground(P_2),
3     calls_pp_inv(P_1, _File_1, P_2).
4
5  calls_pp_inv(P_1, File_1, P_2) :-
6     not( ignore_calls_pp(P_2) ),
7     rar:select(is_called_by, P_2, File_1, P_1),
8     not( ignore_calls_pp(P_1) ).
```

Example 3.20 (Call Dependencies) In the following we show some examples for accessing the ProSoRe database. Assume that the following rules are defined in the Prolog file `basic_algebra/basics/incremental.pl`:

```
increment(X, Y) :-
   Goal = add(1, X, Y),
   call(Goal).

add(I, X, Y) :-
   Y is X + I.
```

- Predicates, which are called within a meta-call predicate such as `call/1` are found by the method `calls/2`, too. If predicates are composed within the body of a predicate and it is traceable how these predicates are composed, then these predicates are found:

```
?- calls(predicate:(user:increment)/2, predicate:P).

P = (user:call)/1;
P = (user:add)/3

Yes
```

- If we want to know, which predicates call the predicate `(user:is)/2`, we call

```
?- calls(predicate:P, predicate:(user:is)/2).

P = (user:add)/3

Yes
```

  In this call, the second parameter is ground and the variable `P` is unbound. Because of this, the method `calls/2` calls the method `calls_pp_inv/3`, which uses the `is_called_by` facts of the PROSORE database to find the result.

Cross References between Files

The following computes calling relationships between files, based on the rules contained in the files.

```
calls(file:?File_1, file:?File_2).
```

One of the two parameters always have to be bound:

Implementation:

```
1  calls(file:F_1, file:F_2) :-
2     ground(F_1),
3     select(rule, P_1, F_1, Rule),
4     calls_pp(Rule, P_1, P_2),
5     select(rule, P_2, F_2, _).
```

The method `select/4` determines the rules of the file `F_1` (line [3]), then the method `calls_pp/3`, which is defined above, determines the predicates called in the rule body (line [4]), and again, the method `select/4` determines the file `F_2` containing rules for these predicates (line [5]).

The method below retrieves the files, which call a certain file:

Implementation:

```
1  calls(file:F_1, file:F_2) :-
2     ground(F_2),
3     select(rule, P_2, F_2, _),
4     select(is_called_by, P_2, F_1, _).
```

In order to determine the predicates of file F_1, which call predicates of file F_2, we have implemented the methods

```
calls(file:?F_1, predicate:?P_1,
   file:?F_2, predicate:?P_2).
```

Remark: At least two Variables have to be ground. This can be either F_1 and P_1, or, for a reverse search, F_2 and P_2.

Cross References between Packages

The following method computes calling relationships between packages, based on the rules contained in the files. One of the two method parameters Level_1:Path_1 and Level_2:Path_2 always have to be bound.

```
calls(?Level_1:Path_1, ?Level_2:Path_2).
```

Thereby, Level is a certain level in the hierarchy tree and Path is the path of the corresponding level. For the DDK, Level can be sources, unit, module or file.

We visualize the cross references between packages in order to structure the source code hierarchically. Thereby, we determine the calls from one package to another package. If a package containing only basic predicates calls a predicate of a package containing only higher level predicates, we reorganize the corresponding predicates of the packages. But sometimes, there are unavoidable calls between packages. Predicates, which are defined as *multifile* and which are distributed over several packages cause cross references between packages, although these calls are not relevant. E.g., the predicate test/2 is called in nearly every package. This predicate is used to test the functionality of the methods defined in a file, module, or unit. The definition always is contained in the corresponding package. Another example would be the predicate dportray/[1-4], which is used to pretty print the output of a method. These predicate calls cause unwished cross references. In order to hide these calls, we put these predicates on a list of ignored methods.

In the following subsection, we explain methods in order to retrieve the predicates defined in a file or package.

3.2.5 The Hierarchy

The DDK has the following structure: the whole source code is contained in the package system, which contains packages named sources. Each sources package contains

several packages named `unit`, each `unit` package contains several packages named `module` and each `module` package contains several files (cf. Figure 3.5).

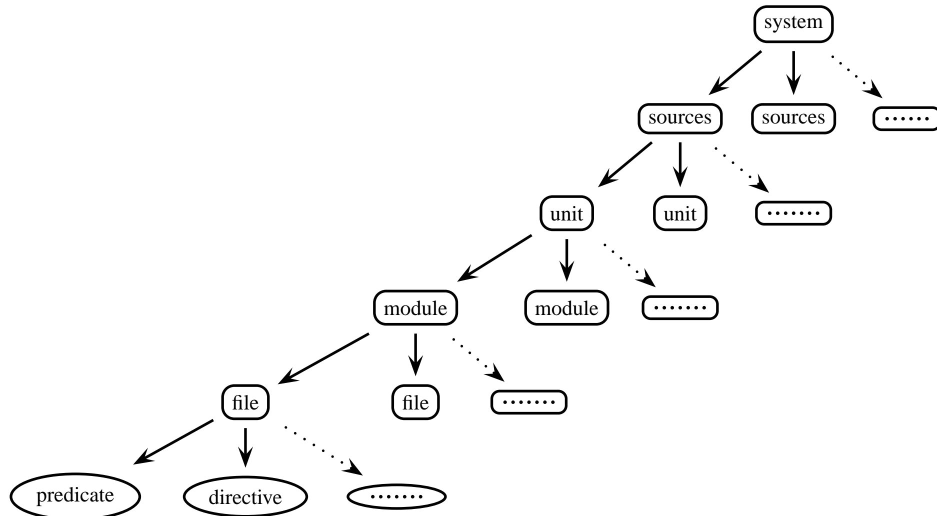


Figure 3.5: The Structure of the DDK Hierarchy

In order to query the available instances of the packages `system`, `sources`, `unit`, `module`, or `file`, we call

```
package_names(+system|sources|unit|module|file,
       -Names).
```

We want to query the ProSoRe database in order to retrieve which file or package contains which predicates, file or sub-package.

In order to retrieve the predicates, which are defined in a file, we call

```
contains(file:?File, predicate:?P).
```

Implementation:

```
1  contains(file:File, predicate:P) :-
2     rar:select(rule, (M:P)/A, File, Rule),
3     [M, P, A] := Rule/head/atom@[module, predicate, arity].
```

The method *excludes* directives contained in the file (line [3]). It can be used reverse, too, in order to retrieve the filename, in which a predicate is defined.

Example 3.21 (Containment) We determine the predicates `P_1`, which are defined in file `F_1`.

```
?- F_1 = 'basic_algebra/basics/incremental.pl',
   contains(file:F_1, predicate:P_1).

P_1 = (user:incremental)/2;
P_1 = (user:add)/3

Yes
```

In general, if we want to retrieve all predicates or sub-packages of a package or file, we call

```
contains(?Level_1:Path_1, ?+Level_2:Path_2).
```

Thereby, `Level` is `predicate` or one of the hierarchy levels of the DDK and `Path` is the corresponding name of the predicate or the path of the hierarchy level.

It is not allowed that both arguments are unbound. Either the first argument, or the second argument has to be bound. The method can be used reverse, too, in order to determine to which higher level packages a given package belongs to.

3.2.6 Predicate Groups and Meta-Call Predicates

We assign each predicate to a certain *predicate-group*. The different predicate-groups are additional divided into classes. The most important predicate-group is the group of *meta-call* predicates.

Location and Assertion The associations between predicates and predicate-groups are defined in an XML configuration document. The corresponding file, containing the XML configuration document, is associated to the alias

```
predicate_groups.
```

The content of this file and the association between this alias and the file can be changed at any time by the user (cf. Appendix A.5).

This XML configuration document is shared by the software repository PROSORE and the libraries RAR and VISUR. The content of the XML configuration document is asserted to PROLOG. This accelerates the access to the diverse predicate-groups. The methods which use the content of this file load and assert the last used configuration associated to the alias `predicate_groups` automatically.

Example 3.22 (Predicate Groups) In this example, we first retrieve the filename of the XML configuration document of the predicate groups, and then, we assert the content of the file.

```
?- alias_to_file(predicate_groups, Filename),
   assert_predicate_groups(Filename).

Yes
```

The asserted facts look as follows:

```
rar:special_predicate(P, Group, Class, Type, Calls).
```

These facts are queried and used by the corresponding methods, e.g., the PROLOG source code parser, and the visualization methods. In the following, we explain the parameters of these facts.

Configuration Each group of the XML configuration document defines certain predicate properties, which specify, e.g., if the predicate should be hidden or duplicated in a graph, or if the predicate is a *meta-call* predicate, which calls further predicates. The SWI-PROLOG built-in predicates are automatically added to the group `built_in` and the class `prolog`.

The knowledge about which predicate belongs to the group of *meta-call* predicates is needed during parsing the PROLOG source code into the PROSORE database. Otherwise, we do not obtain all call dependencies.

Every other predicate that is not a member of one of the groups defined in the configuration file belongs to the group `default` and the class `default`. The following example shows an extract of the XML configuration document. In Appendix D.2, we present a further example of this document.

Example 3.23 (Configuration)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<predicate_groups>
   <predicates group="meta" class="a">
      <atom predicate="maplist" arity="2" module="user"
            calls="[1, n]"/>
      ...
   </predicates>
   <predicates group="built_in" class="rar" type="duplicate">
      <atom predicate="sca_variable_get" arity="2" module="user"/>
      ...
   </predicates>
   <predicates group="built_in" class="prolog" type="duplicate"/>
   <predicates group="default" class="default" type="single/>

   ...
</predicate_groups>
```

The DTD of this XML document looks as follows:

DTD:

```
<!ELEMENT predicate_groups (predicates*)>

<!ELEMENT predicates (predicate*)>
<!ATTLIST predicates
```

```
   group CDATA #required
   class CDATA #required
   type (single|
      duplicate_node|hide_node|
      single_node_hide_subtree|
      duplicate_node_hide_subtree|
      hide_sub_tree|
      hide_node_hide_subtree) #required>

<!ELEMENT atom EMPTY>
<!ATTLIST atom
   module CDATA #required
   predicate CDATA #required
   arity CDATA #required
   calls CDATA #implied>
```

The element `<predicate_groups>` contains `<predicates>` elements. Each element `<predicates>` defines a group of predicates with the same properties. The element `<predicates>` contains several elements `<atom>` and the following attributes:

`group`
: This attribute is used to differ between multiple groups, and can be any group name, except for the group of meta-call predicates.

  If `group=meta`, the predicates in this group belong to the group of meta-call predicates. Creating a call graph, we look in the passed parameters of a meta-call predicate. There, we are looking for predicates that are called by this meta-call predicate in order to create the complete call graph. In the call graph, an edge will connect the meta-call predicate and the called predicates.

  Remark: Meta-call predicates, which call predicates with changing arity cannot be added to the list of meta-call predicates. E.g., the meta-call predicate

```
dislog_file_to_file(Operator, I_Files, O_Files)
```

  cannot be added to the list of meta-call predicates, because the arity of the called predicate `Operator` depends of the list-length of the argument `I_Files` of this meta-call predicate. A possibility would be to add this meta-call predicate repeatedly to the list, each repetition with an incremented arity.

`class`
: can be any name, too. It is for subgrouping a group.

`type`
: This attribute is used in order to specify the appearance of predicates in rule/goal graphs. It can be one of the following values:

- `single`
  This node is only visualized one time in the picture. This causes a lot of crossing edges, which are connected to this node.
- `duplicate_node`
  In the visualization, the node is duplicated each time the corresponding predicate is called. E.g., the node for the predicate `writeln/1` will be duplicated each time `writeln/1` is called. This reduces the amount of crossing edges so that we obtain a better overview in graphs.
- `hide_node`
  The corresponding node of this predicate is not visualized in a picture; but all nodes, which corresponding predicates are called by the predicate of the hidden node are shown.
- `single_node_hide_subtree`
  The node is visualized one time, but all following predicates, which are called by the corresponding predicate are not shown.
- `duplicate_node_hide_subtree`
  Every time this node is called, it is visualized without its subtree.
- `hide_node_hide_subtree`
  The whole subtree of a predicate is not visualized.

Each `<atom>` element has the following attributes:

`predicate`
: contains the name of the predicate.

`arity`
: contains the arity of the predicate.

`module`
: contains the module name of the predicate.

`calls`
: For meta-call predicates, this attribute contains the positions where the meta-call predicate calls a predicate. It is a list. Each position of the list is `n` or an integer. `n` means that the meta-call predicate does not call the corresponding argument. Any integer `I` means, that there is a call at this position and the call changes the arity of the called predicate by the value of the integer `I`.

Example 3.24 (Assertion of Predicate Groups) The following fact can be retrieved in PROLOG, after the XML configuration document `pgroups.xml` is asserted:

```
pgroups.xml:

<?xml version="1.0" encoding="ISO-8859-1" ?>
```

```
<predicate_groups>
  <predicates group="meta" class="a">
    <atom predicate="maplist" arity="2" module="user"
          calls="[1, n]"/>
  </predicates>
<predicate_groups>
```

Calling the following asserts the predicate `maplist/2` to the group of meta-call predicates. Afterwards, we query for the properties of this meta-call predicate.

```
?- assert_predicate_groups('pgroups.xml').

Yes

?- rar:special_predicate((user:maplist)/2,
                      meta, a, duplicate_node, [1, n]).

Yes
```

Chapter 4

Source Code Analysis

The library RAR (Reasoning about Rules) contains complex methods to query the source code, to support refactoring, and to view the results in tables. It is possible to calculate bad smells (anomalies) in the source code. Bad smells are, e.g., *duplicated code*, *dead code*, calls to *undefined methods*, or certain *recursive methods*, which can be implemented iteratively.

- Duplicated source code inflates the source code unnecessary and increases the time and effort of changing methods ([66, 67]).
- Dead code overloads the programmer of a software project. Source code, which is not used anymore should be determined and deleted.
- Undefined methods are methods, which are called somewhere in the source code, but which are not defined anywhere. The result is that parts or even the whole program does not work. To avoid undefined methods, the source code should on the one hand contain *test* methods, which test the functionalities of the program. E.g., a possibility is to call the necessary methods of a program with a certain set of parameters and compare the calculated results with well known results. To be sure that everything works fine, the tests have to be regularly executed, especially after extensive programming. On the other hand, we can search for undefined methods, using call dependencies.
- Using PROLOG, methods are often implemented recursively. But not every method needs to be programmed recursively. Certain methods can be programmed iteratively, too. Using design patterns, it is possible to detect these methods (cf. [53]).

For organizing a large source code project, it is important to know about the *dependencies* between methods, files, and, in general, between packages. We assume that methods which depend on each other, belong to a similar scope. Knowing about the source code dependencies, we summarize related methods in files. Then, depending on the size of the program, we arrange files to packages with similar scopes. We arrange files to packages, which depend on each other and which belong to an equivalent scope. Finally, we arrange

packages to further, bigger packages and so on. Thereby, we try to obtain a hierarchical structure of files and packages, too.

A typical way of programming in PROLOG is to write a new method in an arbitrary file or in a test file. If the new method is tested and it works correct, we move the method to the file, to which the method really belongs. This is the file, which has a similar scope like the new method itself. If we know about call dependencies of a new method and about the call dependencies of the existing methods, it is much easier to assign a new method to the right file. Assigning methods to appropriate files, which are additionally sorted in a hierarchical structure, makes source code much more readable. Knowing about method dependencies helps to maintain the source code. We can extend the functionality of a method or change existing methods, because we know about further consequences.

We use dependency graphs to visualize the method dependencies of a software project. Visualizing dependencies makes it easy to trace and to understand the coherences between the methods.

The DDK has a hierarchical structure consisting of several abstract levels. Each level contains packages. Using call spectra of predicates, we relate predicates to corresponding packages, which have a similar spectrum.

4.1 Call Dependencies

The knowledge about call dependencies between predicates, files, modules, and units helps understanding the structure of a software project. Knowing about the structure helps to refactor and to improve the structure of the source code. Additionally, the visualization of these dependencies facilitates the source code handling. E.g., it makes it easy to delete predicates, to move predicates from one file to another, or to place new predicates in the source code.

All call dependencies between different packages trace back on the call dependencies of predicates. Therefore, it is necessary to know about the call dependencies between predicates, the memberships of each predicate to the files, and to know to which package a file belongs to (cf. Figure 4.1). In Section 3.2, we introduced elementary methods for retrieving the memberships of predicates, elementary methods for retrieving call dependencies between predicates, and elementary methods for retrieving *cross references* between files.

The methods of this section build on these elementary predicates of the software repository PROSORE. In the following subsections, we first define call dependencies between predicates, and we treat *file-file call dependencies* and in general, *package-package call dependencies*. Then, we determine the strongly connected components of a calls, and at last, we show how to extract design information of JAVA source code using JAML and how to extract design information of PHP source code using PHPML.

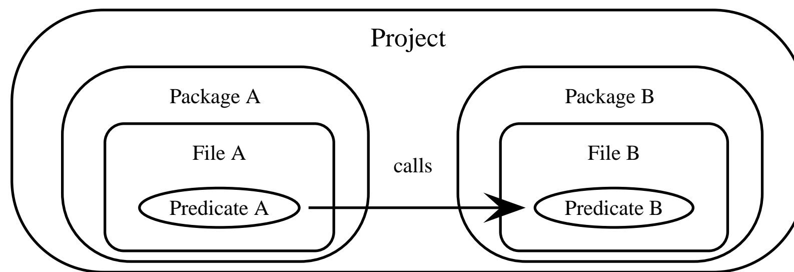


Figure 4.1: Memberships and Dependencies

4.1.1 Calls between Predicates, Files and Packages

In this subsection, we first define the call dependencies between predicates. Especially, we have to pay attention to meta-call predicates. Using the call dependencies between predicates and the membership of a predicate to a package, we determine the call dependencies between files and packages.

Calls between Predicates

We define the *general* rule/goal graph and the *extended* rule/goal graph. The *extended* rule/goal graph – which we always use – can handle the fact that some predicates are able to call a predicate, which is passed in one of their arguments. We call these predicates *meta-call* predicates. Meta-call predicates allow for higher-order programming and are used to call terms constructed at run time. Well-known examples of meta-call predicates in PROLOG are

```
apply/2, bagof/3, call/1, checklist/2,
findall/3, forall/2, ignore/1, maplist/[2,3],
not/1, once/1, setof/3, sublist/3.
```

Example 4.1 (Meta-Call Predicates) The evaluation of `maplist/2` in the following rule `write_list/1` calls `write(A)` for each element `A` of the list `As`:

```
write_list(As) :-
   maplist( write,
      As ).
```

The extended rule/goal graph visualizes meta-call predicates in a way that also shows the predicates called in the arguments of the meta-call predicate. Figure 4.2 shows the extended rule/goal graph of the rule mentioned above; it was generated using SCAV.

The definition of the extended rule/goal graph is based on the general definition of the rule/goal graph.

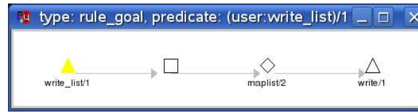


Figure 4.2: The Extended Rule/Goal Graph of write_list/1

Definition (Rule/Goal Graph):

The rule/goal graph is a bipartite graph, which visualizes the head predicates of a rule in dependency of their rules and their body predicates. Thereby, one partition are the head and body nodes, and the other partition are the rule nodes.

- Each head predicate (predicate symbol) is visualized by a node. If a predicate is defined by several rules, the corresponding head predicate only is visualized by *one* node.
- Each rule is visualized by an own node.
- Each predicate of the body is visualized by a node.
- The node of the head predicate is connected to the rule nodes, which belong to this head predicate. The edges are directed.
- The rule node is connected to each of its body predicates. The edges are directed, too.

The general rule/goal graph of a *program* is created by iteratively executing the definition above for each rule of the program. Thereby, it is possible that a head predicate is called within the body of another predicate. Depending on the configuration of SCAV, we either generate for the called body predicate a new node, or, we only use one node and connect an edge to this node for each call.

Given a PROLOG program P and a rule r of P

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

with body atom B_i. p_X is the predicate symbol of an atom $X = p(t_1, ..., t_n)$, i.e. $p_X = p$. We visualize this rule as follows:

Example 4.2 (Rule/Goal Graph) Let us assume that our program consists of the following two rules:

```
write_list(Xs) :-
   maplist( write,
      Xs ).

writeln_list(Xs) :-
   maplist( writeln,
      Xs ).
```

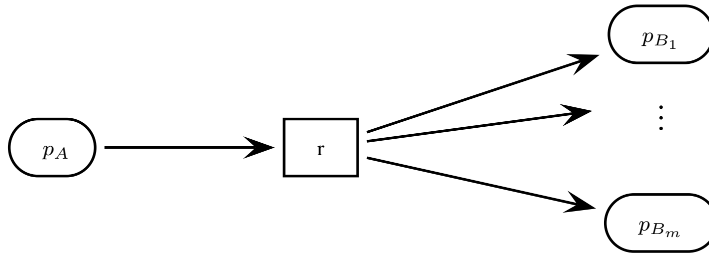


Figure 4.3: Rule/Goal Graph

We visualize the corresponding rule/goal graph in Figure 4.4. There, we add the arity of an atom to the predicate symbol, too, e.g., we write `write/1` instead of `write`.

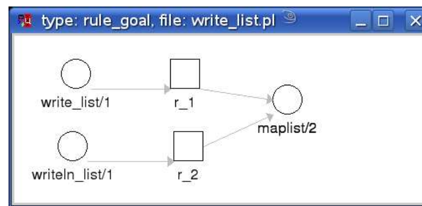


Figure 4.4: The rule/goal graph of write_list/1

Having a large program, we have a lot of rules, which call the same atom, e.g., the atom `maplist/2`. Therefore, we configure SCAV, so that certain predicates are always replicated in the graph, each time they are called. This makes it easier to arrange the nodes, so that we get a better overview of the visualization (cf. Figure 4.5).

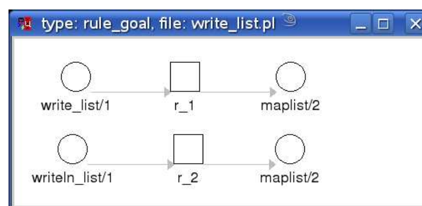


Figure 4.5: The Rule/Goal Graph of write_list/1

Embedded Calls Meta-call predicates call further atoms, which are passed in their arguments. Meta-call predicates are able to change the arity of the called atom, which is passed in their arguments. E.g., in the example above, we do not see the arity of the atom with the predicate symbol `write`. Only the predicate symbol `write` is passed in the

arguments of the meta-call predicate `maplist`, but `write/1` is called. We can specify the arity of the called atom in dependency of the meta-call predicate (cf. Section 3.2.6).

In Figure 4.6, we visualize the *call* of the meta-call predicate `maplist/2` calling the atom `write/1`. Thus, we draw meta-call predicates using a rhombus, and we draw normal predicates using circles. Using SCAV, the appearance of a predicate can be individually configured (cf. Section 6.1).

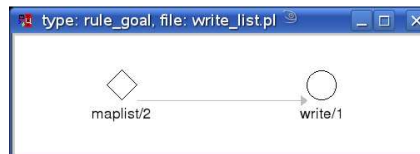


Figure 4.6: Embedded Call from a Meta-Call Predicate

In order to be able to see which predicate transitively calls which other predicates, the meta-call predicates have to be always duplicated. Otherwise, it is not possible to relate the transitive called predicates of several meta-call predicate calls to the corresponding rules (cf. Figure 4.7).

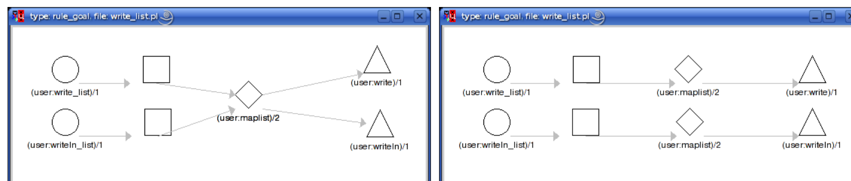


Figure 4.7: Comparison of not duplicated and duplicated Meta-Call Predicate

The Extended Rule/Goal Graph takes meta-call predicates into account. The graph visualizes the head predicates of a rule in dependency of the rules, the body predicates and the meta-call predicates. The definition of the extended rule/goal graph is similar to the definition of the general rule/goal graph.

Definition (Extended Rule/Goal Graph):

- Each head predicate is visualized by a node. If a predicate is defined by several rules, the corresponding head predicate only is visualized by *one* node.
- Each rule is visualized by an own node.
- Each predicate of the body is visualized by a node, independently, if it is a meta-call predicate or not. Meta-call predicates are always duplicated each time they are called.

- Each predicate, which is passed as parameter of a meta-call predicate is visualized by a node. Thereby, we have to know, which predicates are meta-call predicates. We have to know in which of the parameters of a meta-call predicate further predicates are passed and we have to know if the meta-call predicate changes the arity of a passed and called predicate, too.
- The node of the head predicate is connected to the rule nodes, which belong to this head predicate.
- The rule node is connected to each of its body predicates.
- Body predicates, which are meta-call predicates are connected to the nodes of their passed predicates. All edges are directed.

The extended rule/goal graph of a *program* is created by iteratively executing the rules above for each rule of the program.

The Goal Graph We define further graphs, implying the definition of the extended rule/goal graph. The goal graph visualizes the dependencies between the head predicates and the body predicates. Each head and body predicate is visualized by a node. The *goal graph*, does not create rule nodes. We draw an edge from each head predicate to all of its body predicates, independently to which rule of the head predicate the body predicate belongs to.

The extended Goal Graph is defined analogous to the goal graph and the extended rule/goal graph. It takes meta-call predicates into account. There exist no rule nodes and each head and body predicate is visualized by a node.

Further Graphs We visualize further graphs, too. The *file dependency graph* (cf. Figure 4.8) shows the dependencies between the files. For each file, we draw a corresponding node in the graph. If a method of file `F_1` calls a method of file `F_2`, we draw a directed edge from file `F_1` to file `F_2`. Another way to obtain the file dependency graph is to use the goal graph and to condense all nodes, which represent methods of the same file to one node.

Other graphs are the *module dependency graph*, the *unit dependency graph* or any other *package graph*. In SCAV, the user can define further graph structures, and visualize them in a picture.

Calls between Files

We have implemented diverse methods to retrieve call dependencies between files. If we want to retrieve the predicates, which are calling a predicate of file `F_1`, but which are not defined in file `F_1` (external predicates), we call

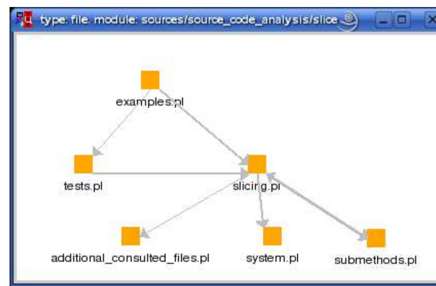


Figure 4.8: File Dependency Graph of the Module *Slice*

```
file_dependency:file_called_from_external_predicates(
   +F_1, -Calls).
```

This generates a list of the calls from *external predicates*; this means, each call comes from a predicate, which is not defined in file `F_1`, but which calls a predicate of file `F_1`. The method is defined in the following way:

Implementation:

```
1   file_called_from_external_predicates(F_1, F_Ps_2) :-
2      findall( F_2:P_2-F_1:P_1,
3         file_predicate_is_external_called(
4            F_1:P_1, F_2:P_2),
5         F_Ps_1 ),
6      list_to_ord_set(F_Ps_1, F_Ps_2).
7
8   file_predicate_is_external_called(F_1:P_1, F_2:P_2) :-
9      rar:select(rule, P_1, F_1, _),
10     rar:select(is_called_by, P_1, F_2, P_2),
11     not( F_1 = F_2 ).
```

In line [9], we select a head predicate `P_1` and the corresponding file `F_2` of a rule. Then, we look for calls of this predicate, using the inverse rule facts of the ProSoRe database (line [10]). We extract the corresponding file `F_2` of the predicate `P_2`. At last, we verify that the two filenames `F_1` and `F_2` are different. Using `findall/3`, we get all *external predicates* and using `list_to_ord_set/2`, we get a sorted list without duplicates (lines [2-6]).

Each element of this list has the following structure:

```
Element = F_2:P_2-F_1:P_1.
```

This means that predicate `P_2` of file `F_2` calls predicate `P_1` of file `F_1`.

If we want to know the reverse calls, this means, if we want to know the predicates of file `F_1` calling external predicates, we call

```
file_dependency:file_calls_external_predicates(
   +F_1, -Calls).
```

Each element of this list `Calls` has the following structure:

```
Element = F_1:P_1-F_2:P_2.
```

Again, this means that predicate `P_1` of file `F_1` calls the predicate `P_2` of file `F_2`.

To determine the external predicates, which are called by the directives of a file, we call

```
file_dependency:
   directive_of_file_calls_external_predicates(
      +F_1, -Calls).
```

This methods retrieves the directives and determines the called predicates. Afterwards, it looks if the called predicates are defined in another file. The format of each element of the resulting list `Calls` is the same as described above:

```
Element = F_1:P_1-F_2:P_2.
```

Thereby, the name of `P_1` always contains the prefix `no_head_`.

Calls between Packages

Knowing about the membership of predicates and files, and knowing about the predicate- and file-dependencies, we derive the dependencies of modules, units or in general of packages. In order to determine the cross over dependencies of a package, we call

```
rar:calls(?Package_1, ?Package_2).
```

Thereby, `Package_1`, `Package_2` has the following structure:

```
Package_1 = Level_1:Path_1,
Package_2 = Level_2:Path_2.
```

Either the level or the path of the term package has to be bound. For the DDK, `Level_1`, and `Level_2` can be `system`, `sources`, `unit`, `module`, or `file`. This call determines the name of the packages `Path_2`, which belong to the level `Level_2` and which contain at least one predicate that is called by a predicate of the package `Path_1` of level `Level_1`. `calls/2` uses basic methods in order to retrieve the dependencies between the packages.

Implementation:

```
1   calls(Package_1, Package_2) :-
2      Package_1 = Level_1:Path_1,
3      ground(Level_1)
4      ; ground(Path_1),
5      !,
6      rar:contains(Package_1, predicate:P_1),
7      rar:calls(predicate:P_1, predicate:P_2),
8      rar:contains(Package_2, predicate:P_2).
9   calls(Package_1, Package_2) :-
10     Package_2 = Level_2:Path_2,
11     ground(Level_2)
12     ; ground(Path_2),
13     !,
14     rar:contains(Package_2, predicate:P_2),
15     rar:calls(predicate:P_1, predicate:P_2),
16     rar:contains(Package_1, predicate:P_1).
```

In order to speed up requests, `calls/2` consists of two rules. Depending if either a part of the first argument is ground, or a part of the second argument is ground, we call the predicates in a different order (lines [6-8] and [14-16]). Line [6] retrieves a predicate `P_1` contained in the chosen package `Package_1`. Then, we search for a predicate `P_2`, which is called by `P_1` (line [7]). At last, we look for the package to which the predicate `P_2` belongs to (line [8]).

4.1.2 Strongly Connected Components

A *strongly connected component* of a directed graph is a maximal set of vertices, such that every vertex is reachable from every other vertex. The call dependency graphs are directed graphs, which may contain cycles and strongly connected components, respectively.

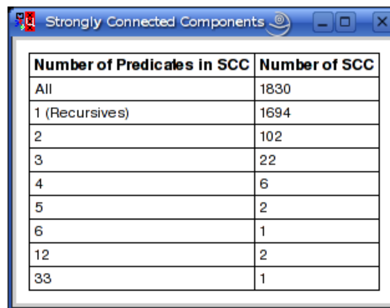
In our statistics, we differentiate between the number of vertices, and predicates respectively, which a strongly connected components contains. Figure 4.9 shows a table with the summary of the strongly connected components. The first column shows the number of predicates of a strongly connected component. The second column shows the number of strongly connected components having a certain number of predicates. Strongly connected components having only one predicate are directly recursive predicates. In PROLOG, such predicates are often used to process lists.

In order to generate this table, we just call

```
table_strong_components.
```

This generates a temporary database (cf. Section 4.2) of the strongly connected components, using methods of the PROLOG library *ugraphs*. Then, it visualizes the table containing the results. We can query the facts of this temporary database after it has been created, by calling

```
strong_component(-SCC).
```



Strongly Connected Components

Number of Predicates in SCC	Number of SCC
All	1830
1 (Recursives)	1694
2	102
3	22
4	6
5	2
6	1
12	2
33	1

Figure 4.9: Strongly Connected Components

Each fact contains a list of predicates belonging to the same strongly connected component.

We need to know about the strongly connected components, in order to determine the called levels.

4.1.3 Dependencies in PROLOG, JAVA or PHP

[22], [23], and [53] demonstrate reasoning about various types of source code such as PROLOG rules or JAVA programs. Using VISUR, the call dependencies between the methods, modules, units or classes of PROLOG, JAVA, or PHP source code can be visualized.

Consulted PROLOG Source Code

In the Sections 3.1 and 4.1, we showed how we represent PROLOG in XML and how to define corresponding predicates, which can be used to detect dependencies in this representation. In Section 3.2.1, we compared this possibility of detecting call dependencies with following possibility: instead of using an XML representation in order to reason on the source code, we can define querying predicates which work directly on the consulted source code of the PROLOG environment using the built-in predicate `clause/2`.

Example 4.3 (Querying Consulted Predicates)

```
calls_consulted(P/A, Functor/Arity) :-
   functor(Term, P, A),
   clause(Term, Body),
   body_to_predicate(Body, Element),
   functor(Element, Functor, Arity).

body_to_predicate(Body, Element) :-
   Body = (Element, _).
body_to_predicate(Body, Element) :-
   Body = (_, Body_Part),
   body_to_predicate(Body_Part, Element).
```

This possibility works fine for small projects. Unfortunately, we can not use `clause/2` to search for all predicates, which are called by a certain predicate.

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);
      }
   }
}
```

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>
```

Extracting Design Information

We can recover design information from JAVA source code in JAML representation in two steps; the rules for analyzing the code are separated into the two files `basic` and `analysis`.

The first file, `basic`, defines some basic relations between methods and classes using path expressions in FNSELECT. 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.

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 [48] 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).
```

We have implemented a plug-in to manage JAML source code files, which we explain in Appendix C.2 in detail.

Dependencies of PHP Source Code

In the diploma thesis [14], Christian Graiger defines the XML representation PHPML of PHP source code. Using PHPML, we can visualize the class hierarchies and the method calls of PHPML.

Example 4.4 (Class-Declarations Representation in PHPML) The attribute `name` of the element `<class-declaration>` contains the name of the class; the subelements `<superclass>` and `<interfaces>` contain information about the superclass and the implemented interfaces.

Using SCAV, the PHPML documents of each file is split into parts, each part containing the XML representation of a class. We select a class using `select/4` of the PROSORE database.

```
<class-declaration name="PhpParseTree" is-interface="false"
  is-userdefined="true" is-abstract="false" is-final="false">
  <superclass>
    <class name="ParseTreeImpl" is-interface="false"
      is-userdefined="true" is-abstract="false" is-final="false"/>
  </superclass>
  <interfaces>
    <class name="ParseTree" is-interface="true"
      is-userdefined="true" is-abstract="true" is-final="false"/>
  </interfaces>
  <keyword>class</keyword>
  <identifier>PhpParseTree</identifier>
  <keyword>extends</keyword>
  <extends-expression>
    <identifier>ParseTreeImpl</identifier>
  </extends-expression>
  <block>...</block>
</class-declaration>
```

To visualize the class hierarchy, the predicate `calls/2` can be implemented as follows:

```
calls(class:C_1, superclass:C_2) :-
   rar:select(rule, C_1, _, Xml),
   Class := Xml/_/'class-declaration',
   C_1 := Class@name,
   C_2 := Class/superclass/class@name.

calls(class:C, interface:I) :-
   rar:select(rule, C_1, _, Xml),
   Class := Xml/_/'class-declaration',
   C := Class@name,
   I := Class/interfaces/class@name.
```

To visualize the method calls, the predicate `calls/2` can be implemented as follows:

```
calls(method:M_1, method:M_2) :-
   rar:select(rule, _, _, Xml),
   D := Xml/_/'method-declaration',
   \+ ('true' := D@'is-constructor'),
   M_1 := D@name,
   I := D/_/'method-invocation',
   M_2 := I@name.
```

4.2 Anomalies, Software Metrics, and Design Patterns

A PROLOG file contains rules and directives. Several rules, having the same predicate symbol, define a predicate or method, respectively. These rules build the definition of a predicate. The directives are instructions, which are executed when the file containing directives is consulted.

PROLOG Modules Predicates can be encapsuled in PROLOG modules. In order to define a PROLOG module, the first statement of a file has to be the module definition. In a file, only *one* PROLOG module can be defined. Conversely, the content of several files can be combined to one PROLOG module. Although several files can be combined to one PROLOG module, this should only be done rarely. It is not easy to maintain PROLOG modules, which are divided into several files.

Dynamic Predicates *Dynamic* defined predicates are facts or predicates, which definitions can be changed during runtime. We change the definition of a predicate by asserting further facts or rules to or by retracting existing facts or rules of the program during runtime.

Multifile Predicates If the rules of a predicate are spread of over several files, we get a warning of the PROLOG interpreter. However, the rules of *multifile* defined predicates can be written in several files.

Discontiguous Predicates Normally, the rules of a predicate are in succession. The succession of rules of *discontiguous* defined predicates can be interrupted by rules of other predicates.

The DDK The DDK has a hierarchical structure. The top level of this hierarchy are the *libraries* and *sources*. The *libraries* contain external source code. The *sources* contain the source code of the DDK. It is divided into *units*, which summarize in the next level the *modules*. The *modules* summarize the *files*. In order to navigate through the diverse hierarchy levels of the DDK, we summarize the *sources*, *units*, *modules*, or *files* in the following way to *packages*:

```
Package = Level:Path
```

Thereby, `Level` can be `sources`, `unit`, `module`, or `file` and `Path` is the complete path to the corresponding package.

The hierarchical structure of the DDK corresponds to the file system hierarchy. The root directory *DisLog/sources/* contains the unit-directories, each unit-directory contains module-directories and each module-directory contains the source-code files.

Actually, the DDK contains about 9 units, 90 modules and about 750 files. It has about 138.000 LoC and about 9.700 predicates, whereby each predicate has averaged 2 rules. The DDK consists of about 320 *dynamic* defined predicates, 70 *multifile* defined predicates, and about 80 *discontiguous* defined predicates. Figure 4.10 compares the statistics of the year 2007 with the statistics of the year 2004. The last column shows the percentage of changes. We see, in the past three years, all scopes increased.

System Statistics

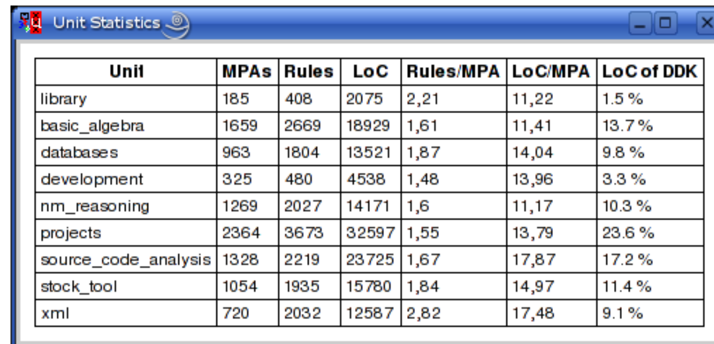
Attribute	2007	2004	%
Units	9	8	+13
Modules	90	58	+55
Files	753	444	+70
Prolog Modules	107	47	+128
Predicates	9705	7657	+27
Rules	17355	14039	+24
Directives	539	443	+22
Dynamic	326	110	+196
Multifile	74	63	+17
Discontiguous	77	58	+33
LoC	138888	100653	+38

Figure 4.10: System Statistics of the DDK

We generate an actual statistics of the DDK calling the method

```
statistics:table_system_statistics/0.
```

Figure 4.11 shows for each *unit* of the DDK the number of predicates, rules and LoC. Furthermore, we show the averaged rules per predicate and the averaged LoC per predicate. We see that a predicate consists of averaged approximately two rules and twelve lines of source code. The last column shows the percentage part of the unit of the whole DDK.



Unit	MPAs	Rules	LoC	Rules/MPA	LoC/MPA	LoC of DDK
library	185	408	2075	2,21	11,22	1.5 %
basic_algebra	1659	2669	18929	1,61	11,41	13.7 %
databases	963	1804	13521	1,87	14,04	9.8 %
development	325	480	4538	1,48	13,96	3.3 %
nm_reasoning	1269	2027	14171	1,6	11,17	10.3 %
projects	2364	3673	32597	1,55	13,79	23.6 %
source_code_analysis	1328	2219	23725	1,67	17,87	17.2 %
stock_tool	1054	1935	15780	1,84	14,97	11.4 %
xml	720	2032	12587	2,82	17,48	9.1 %

Figure 4.11: Unit Statistics of the DDK

We generate an actual *unit* statistics of the DDK calling the method

```
statistics:table_units_ps_rules_loc/0.
```

Remark: In order to speed up the creation of this table, the methods above generate a temporary database (cache), which stores the results for further usage. This temporary database is used in Section 4.2.3, too. In the following subsections, further temporary databases are created. Once a temporary database is generated, all further, equivalent requests are calculated very fast. If a certain method is called again, the corresponding temporary database will not be generated again. The temporary database facts are only deleted after *generating*, *reloading*, or *updating* the PROSORE database. They are not saved anywhere on disc for later usage.

For both statistical methods mentioned above, the PROSORE database of the software repository PROSORE has to be generated or loaded before. We can do this using the GUI of SCAV. In order to start the GUI, we just call `visur_gui`.

We have implemented further statistics about the usage of the source code. We present these statistics in tables, too. In the following subsections, we present how often a predicate is called, which predicates are not defined and which predicates are never called (dead code). At last, we describe the detection and the refactoring of PROLOG using design patterns.

4.2.1 Undefined Predicates and Dead Code

Undefined predicates are predicates, which are called by a method, but which are not defined somewhere in the source code. The source code contains no rules, defining these predicates. These predicates may cause problems.

Dead code is source code which is defined in the source code, but which is not called by any method of the source code. But not every method, which is not called by any other method is automatically dead code. Only source code, which never is *used* or *needed* is called dead code. In the following, we describe how to detect dead code and undefined predicates in the source code.

Undefined Predicates

In order to find undefined predicates, we first generate a list of all called predicates of the source code. Thereby, we also pay attention to predicates, which are called from meta-call predicates. Then, we search for the definition of each predicate of this list. We call

```
undefined_code:list_undefined_code(-List),
```

in order to get a simple list of the undefined predicates.

Implementation:

```
1  list_undefined_code(Undefined_2) :-
2     findall( P,
3        undefined_code(P),
4        Undefined_1 ),
5     list_to_ord_set(Undefined_1, Undefined_2).
6
7  undefined_code(P) :-
8     rar:select(is_called_by, P, _, _),
9     not( rar:select(rule, P, _, _) ),
10    not( excluded_undefined_code(P) ).
11
12 excluded_undefined_code(P) :-
13    built_in:predicate(P),
14    !.
15 excluded_undefined_code(P) :-
16    dead_and_undefined_code_excluded:
17       excluded_undefined_code(P, _).
```

First, we select all predicates, which are called in the body of a method (line [8]). Therefore, we generated the *inverse rule* facts, which are part of the PROSORE database. Then, we exclude the predicates, for which a rule exists using the *rule* facts of the PROSORE database (line [9]).

Normally, there exists no definition of built-in predicates in the analyzed source code. Therefore, built-in predicates would be wrongly identified as undefined predicates. We exclude PROLOG and XPCE built-in predicates, too (line [10]). These predicates do not have rules in the PROSORE database. They are defined internally. We query, if a predicate is a built-in predicate or not, using the method

```
built_in:predicate/1.
```

We complete the list of built-in predicates (line [16]) by adding further predicates to the file

```
source_code_analysis/rar/built_in.pl
```

In order to exclude further predicates from the list of undefined predicates, we have an *exclusion list*. Each predicate mentioned in this list does not appear in the result list, too. The exclusions are defined in the file

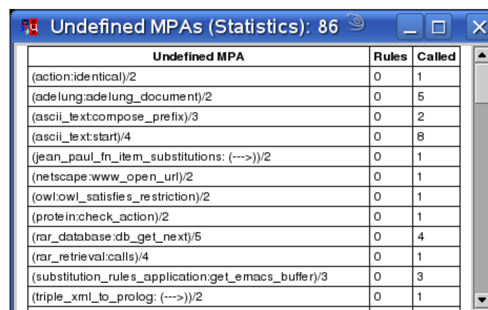
```
source_code_analysis/statistics/
   dead_and_undefined_code_excluded.pl
```

Last, we remove duplicates and sort the result alphabetically (line [5]).

In order to search for all undefined predicates in the source code, and to list them in a table (cf. Figure 4.12), we just call

```
undefined_code:table_undefined_code/0.
```

This table includes some statistical information, too. The first column of the table shows the *predicate name* of the undefined predicate, the second shows how many rules this predicate has and the third column shows, how often this predicate is called from other predicates. Undefined predicates always have zero rules; this means that the value of the second column always is zero.


Undefined MPAs (Statistics): 86

Undefined MPA	Rules	Called
(action:identical)/2	0	1
(adelung:adelung_document)/2	0	5
(ascii_text:compose_prefix)/3	0	2
(ascii_text:start)/4	0	8
(jean_paul_fn_item_substitutions: (--->))/2	0	1
(netscape:www_open_url)/2	0	1
(owl:owl_satisfies_restriction)/2	0	1
(protein:check_action)/2	0	1
(rar_database:db_get_next)/5	0	4
(rar_retrieval:calls)/4	0	1
(substitution_rules_application:get_emacs_buffer)/3	0	3
(triple_xml_to_prolog: (--->))/2	0	1

Figure 4.12: Statistics about "Undefined Predicates"

Instead of a statistical analysis of the calls of the undefined predicates, we generate an extended table of undefined predicates, too. The following call shows a table listing files and predicates, which call undefined predicates (cf. Figure 4.13).

```
undefined_code:table_undefined_code_extended/0.
```

Again, the first column lists the predicates, for which we did not find any rules (undefined predicates). The second column lists the files, which contain one or more predicates calling the undefined predicate of the first column. The third column lists the corresponding predicates.

Not every suggested undefined predicate is really an undefined predicate. Sometimes, a method just calls a fact, which only is asserted at runtime. As we cannot differentiate between the call of a predicate and the call of a fact, we search for the definition of each

Undefined MPAs (Extended): 86

Undefined MPA	Caller File	Caller MPA
(action:identical)/2	sources/projects/protein_pathways/soup_graph_search.pl	(protein_pathway_graph:performed)/2
(adelung:adelung_document)/2	sources/projects/adelung/adelung.pl	(adelung:adelung_gui)/1
	sources/projects/adelung/adelung_1.pl	(adelung_1:adelung_process_single_document_1)/2
	sources/projects/adelung/adelung_word_lists.pl	(user:adelung_coarsen)/0
		(user:adelung_key_to_found_word_in_text)/1
		(user:adelung_to_word_list)/0
(ascii_text:compose_prefix)/3	sources/xml/xml_pillow/xml_pillow_www_support.pl	(xml_pillow_www_support:rel_to_abs_url_text)/3
(ascii_text:start)/4	sources/xml/xml_pillow/xml_pillow_www_support.pl	(xml_pillow_www_support:abs_to_rel_url)/3
		(xml_pillow_www_support:rel_to_abs_url_text)/3
		(xml_pillow_www_support:url_split)/2
		(xml_pillow_www_support:xml_pillow_w_s_path_info)/5

Figure 4.13: Undefined Predicates

called predicate. If it is a fact, which is only asserted at runtime, no definition exists and we would wrongly identify this fact as an undefined predicate.

Facts, which are asserted at runtime should always be declared `dynamic`, in order to avoid runtime exceptions. In PROLOG, this is not obligatory necessary; but in SWI-PROLOG, it is recommended to declare such facts `dynamic`. We synchronize the result of the search with the list of the dynamic defined facts. Therefore, we have implemented a method for searching `dynamic`, `multifile`, and `discontiguous` defined predicates. The method

```
statistics:find_predicate_property(
        +dynamic|multifile|discontiguous, -P_Files)
```

returns the corresponding list `P_Files` of predicate files tuples. Each tuple contains a predicate and the corresponding file, in which the predicate is defined. In order to find dynamically defined facts, we have implemented the method mentioned above in the following way. Thereby, we search for directives in the PROSORE database, which look in the original PROLOG file as follows:

```
:- dynamic rule/6, is_called_by/6, leaf_to_path/4.
```

This source code snippet is represented in PROLOGML in the following way:

```
<rule module="rar">
  <head/>

  <body>
    <atom arity="1" module="user" predicate="dynamic">
      <term functor=",">
        <term functor="/">
          <term functor="rule"/>
          <term functor="6"/>
        </term>
        <term functor=",">
          <term functor="/">
            <term functor="is_called_by"/>
            <term functor="6"/>
```

```
          </term>
          <term functor=",">
            <term functor="/">
              <term functor="leaf_to_path"/>
              <term functor="4"/>
            </term>
          </term>
        </term>
      </term>
    </atom>
  </body>
</rule>
```

Implementation:

```
1   find_predicate_property(Type, P_File_2s) :-
2      findall( P-File,
3         predicate_of_type(Type, File, P),
4         P_File_1s ),
5      list_to_ord_set(P_File_1s, P_File_2s).
6
7   predicate_of_type(Type, File, (M:P2)/A) :-
8      rar:select(rule, (M:-)/-, File, Rule),
9      Atom := Rule/body/atom::[
10        @module=user, @predicate=Type, @arity=1],
11     ( Term := Atom/_/term::[@functor='/']
12     ; Term := Atom/term::[@functor='/'] ),
13     P2 := Term-nth(1)/term@functor,
14     A := Term-nth(2)/term@functor.
```

We search for all rules in the ProSoRe database (lines [2-4]). There, we filter the rules without head predicate, because dynamic defined facts are defined in a Prolog directive (line [8]). Then, we search, if the first predicate of the rule has the predicate symbol `Type`, in our case "`dynamic`" (lines [10-11]). At last, we retrieve the predicates and the arities using PL4Xml (lines [12-15]) and sort them alphabetically (line [5]). The list `P_File_2s` consists of the following structure:

```
P_Files = [P_1-File_1, P_2-File_2, ...].
```

In this list, the predicate and the corresponding filename, in which the predicate is defined either `dynamic`, `multifile`, or `discontiguous`, are separated through a minus. We subtract the *dynamic* defined predicates from the list of undefined predicates.

Sometimes, we just want the undefined predicate calls of a certain unit, module, or file. In order to retrieve the undefined predicate calls of one of these packages, we call one of the following methods

```
undefined_code:undefined_code_of_package(
   +Package, -List).

undefined_code:undefined_code_of_package_extended(
   +Package, -List).
```

Example 4.5 (Undefined Code)

```
?- undefined_code:undefined_code_of_package(
      unit:'sources/basic_algebra', List),
   writeln_list(List).

List = [(user:dislog_module_description)/2,
        (user:graphicals)/2,
        (user:manual_test_example)/2,
        (user:xml_file_to_db)/3],

Yes

?- undefined_code:undefined_code_of_package_extended(
      unit:'sources/basic_algebra', List),
   writeln_list(List).

List = [
   [ (user:dislog_module_description)/2,
     'utilities/code_analysis.pl',
     (user:dislog_module_to_files)/2],
    ... ],

Yes
```

The first call just retrieves an alphabetically sorted list of the undefined predicates, which are called in the specified package. The second call retrieves an alphabetically sorted list of lists. The first entry of each list is an undefined predicate, the second entry is the file, in which the predicate – third entry – calling the undefined predicate, is defined. Thereby, each undefined predicate is called from a predicate of the specified package.

Dead Code

We try to detect dead code by searching for predicates, which are not called by any other predicate. But not every predicate, which is not called by another predicate is dead code. E.g., test predicates `(user:test)/2` or methods, which are used to start a GUI, are not called by another method. Methods, which are stand alone methods and are just designed to do certain calculations are not dead code, too. These predicates are often defined in PROLOG modules and are exported in the corresponding file using the source code in the following example.

Example 4.6 (Directives) The following directive defines the module `rar` and exports the predicate `xml_file_to_db/3`.

```
:- module( rar, [xml_file_to_db/3] ).
```

We assume that all predicates of the list of exported predicates do not belong to dead code and we delete all exported predicates from the list of dead code. We exclude the test predicates of all modules and with arbitrary arity, too.

In some further cases, necessary source code is wrongly identified as dead code. Calls, by which the called predicate is constructed at runtime are difficult to detect, because, there exists many ways of doing so. A simple way for constructing a call is shown in the following example. This example contains two facts, which can be called.

Example 4.7 (Call Construction)

```
parent(Name) :-
   ( Type = father
   ; Type = mother ),
   apply(Type, Name).

father(mayer).
mother(mueller).
```

Calling `parent(Name)` constructs the facts `father(Name)` and `mother(Name)`, which are, then, called by the method `apply/2`.

It is even more complex to find call dependencies, if the predicate name is passed as an argument to the constructing method. The following example passes the called fact as argument `Type` to the method `parent/2`.

Example 4.8 (Call Construction)

```
parent(Type, Name) :-
   apply(Type, [Name]).
```

In order to find `father/1` or `mother/1` facts, we call

```
?- parent(father, Name).

Name = mayer

Yes

?- parent(mother, Name).

Name = mueller

Yes
```

We use this construction method to call different *dynamic* facts of the PROLOGML data - base. In order to find predicates in the PROSORE database fast, we have added supplementary hash values to the called fact. In the following example, `select/4` constructs a predicate call. A further example is given on page 70.

Example 4.9 (Call Construction)   We call

```
?- rar:select(rule, P, File, Rule).

Yes

?- rar:select(is_called_by, P_1, File_2, P_2).

Yes
```

in order to select the following facts:

```
rule(H_1, H_2, P, File, Rule, NS).
is_called_by(H_1, H_2, P_1, File_2, P_2, NS).
```

The implementation of `select/4` looks as follows:

Implementation:

```
1  select(Fact, V_1, V_2, Result) :-
2     sca_namespace_get(NS),
3     hash_term(V_1, H_1),
4     hash_term(V_2, H_2),
5     apply(Fact, [H_1, H_2, V_1, V_2, Result, NS]).
```

There exists further possibilities to compose calls. E.g., we can use `atom_concat/3`, `concat_atom/2`, `sub_atom/5`, `atom_prefix/2` to generate new callable atoms. As long as the called predicate parts are named in the source code, and as long as there exists a path to the constructed predicate parts in the source code, it is possible to detect and reconstruct the called predicate by knowing the method of how it is constructed. But searching for all of those possibilities in the source code would heavily increase the runtime during searching call dependencies.

Additionally, there also exists possibilities to construct predicates without having the called predicate named in the analyzed source code; they can be defined in an additional, non PROLOG document.

Example 4.10 (Call Construction)   We define calls of the menus of SCAV in an XML document. The menu methods, which we want to execute by selecting a certain menu item are read from the XML document and are constructed at runtime. These call dependencies do not appear in the PROLOG source code. The calls to the methods are implemented using a variable and, e.g., the meta-call predicate `call/1`. It is practically impossible to search for such dependencies.

To exclude these predicates from dead code, we have implemented the method

```
dead_code:list_dead_code(-Dead_Code_Ps)
```

in the following way:

Implementation:

```
1   list_dead_code(DCs_2) :-
2      exported_ps(Exported_Ps),
3      findall( P,
4         ( select(rule, P, _, _),
5           not( member(P, Exported_Ps) ),
6           not( is_called_from_another_predicate(P) ) ),
7         DCs_1 ),
8      list_to_ord_set(DCs_1, DCs_2).
9
10  is_called_from_another_predicate(P_1) :-
11     rar:select(is_called_by, P_1, _, P_2),
12     not( P_1 = P_2 ).
13
14  exported_ps(Exported_Ps_2) :-
15     sca_variable_get(exported_predicates, XML_Exported_Ps),
16     findall( (M:P)/A,
17        ( member(Element, XML_Exported_Ps),
18          [P, A] := Element/atom@[predicate, arity],
19          M := Element@name ),
20        Exported_Ps_1 ),
21     list_to_ord_set(Exported_Ps_1, Exported_Ps_2).
```

We exclude the list of exported predicates from dead code, because the exported predicates can be main predicates of interfaces or starting predicates of GUIs. Therefore, first, we retrieve the list of exported predicates (line [2]). Then, we search for all predicates, which are *not* member of the list of exported predicates and which are *not* called from another predicate (lines [3-7]).

In order to find the predicates, which are called from another predicate, we defined the method `is_called_from_another_predicate/1` (line [10]). First, we select an inverse rule (line [11]). All predicates stored in the first parameter of an inverse rule is called by the predicate, stored in the third parameter of an inverse rule. It is possible that predicates call themselves (recursive predicates). These recursive predicates are not identified as dead code, if we only use the trivial constraint

```
not( select(is_called_by, P_1, _, _) ).
```

Using the constraint `is_called_from_another_predicate/1` instead, we include predicates, which call themselves recursively, but which are not called by other predicates (line [12]) in the list of dead code, too.

We list all possible dead code predicates of the source code in a table by calling the method

```
dead_code:table_dead_code.
```

Figure 4.14 shows a table of dead code predicates of the unit

```
source_code_analysis.
```

The first column of the table shows the file, containing dead code. The second column shows the name of the predicates, which are not called, the third column shows the number of rules and the fourth column shows, how often this predicate is called from any other predicates. Recursive calls are not counted in this column. Dead code is not called from other predicates, and therefore, all entries of the fourth column have the value `0`.

Dead Code of unit sources/source_code_analysis: 72

File	Predicate	Rules	Called
sources/source_code_analysis/common/multiset.pl	(multiset:fill_multiset)/3	5	0
	(multiset:multiset_max)/2	1	0
sources/source_code_analysis/common/sca_variables.pl	(user:sca_variable_increment)/1	1	0
sources/source_code_analysis/diagrams/chart.pl	(chart:barchart)/0	1	0
sources/source_code_analysis/rar/calls_ll.pl	(rar:calls_ff_inv)/2	1	0

Figure 4.14: Dead Predicates

Sometimes, we just want the dead code of a certain *unit*, *module*, or *file*. In order to retrieve the dead code of one of a packages, we call

```
dead_code:dead_code_of_package(+Package, -List).
```

Example 4.11 (Dead Code)

```
?- dead_code:dead_code_of_package(
      unit:'sources/source_code_analysis', List),
   writeln_list(List).

List = [['source_code_analysis/common/basics.pl',
         (user:add_slash)/2, 1, 0],
        ['source_code_analysis/common/basics.pl',
         (user:atom_to_term_sure)/2, 1, 0],
        ...],

Yes
```

We retrieve an alphabetically sorted list of lists. Each list contains a filename, and a predicate belonging to the selected package. Furthermore, the lists contains the amount of rules of the predicates and how often this predicate is called. “0” means that the predicate is not called in the whole source code.

In order to exclude the predicates mentioned above from the list of dead code, we use a manually created list of predicates, which are surely no dead code. We add the *exported predicates* mentioned above to this list, too. This list can be also manually completed. It is stored in the file

```
source_code_analysis/statistics/
   dead_and_undefined_code_excluded.pl.
```

In order to view the excluded predicates, which have been deleted in the meantime from the source code, we call

```
dead_code:
   not_existing_but_excluded_dead_code(-Ps).
```

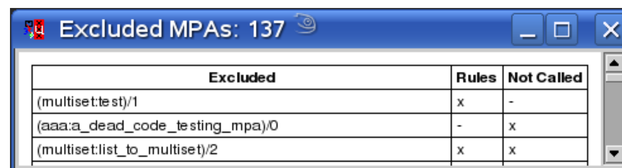
This method extracts all predicates from the list of *excluded* dead code predicates, which are even *not* defined in the source code. We can manually delete these predicates from the list of excluded dead code. If we want to search for predicates in the excluded dead code, which are in the meantime called by any predicate or exported and which need not to be excluded from the list of dead code anymore, we call

```
dead_code:called_but_excluded_dead_code(-Ps).
```

In order to view a table with the list of all *excluded predicates*, we call

```
dead_code:table_excluded_dead_code.
```

This table lists in the second column, if there exists any rules of the predicate, and in the third column, if the predicate is called or not (cf. Figure 4.15).



Excluded	Rules	Not Called
(multiset:test)/1	x	-
(aaa:a_dead_code_testing_mpa)/0	-	x
(multiset:list_to_multiset)/2	x	x

Figure 4.15: Excluded Dead Code Predicates

4.2.2 Predicate Properties

The DDK contains about 10.000 predicates. The following call generates a table of the predicates of a file, which lists statistics properties of the predicates:

```
table_predicate_statistics.
```

First, this method opens a dialog with a list of all files of the project. We choose a filename and we click the button "*Show Predicates in Table*". Only the predicates defined in the chosen file are visualized in the table. Instead of choosing a file of the list, we call

```
table_predicate_statistics(+Filename).
```

The table (cf. Figure 4.16) shows the predicate name in the first column; the second column shows how many rules this predicate has, the third column shows, how often a predicate is called from a predicate defined in the same file, the fourth column shows the number of calls to predicates defined in the same file, the fifth column shows the number of calls from predicates defined in another file, and the last column shows the number of calls to predicates defined in another file.

Predicate Statistics of File: sources/source_code_analysis/gxl/merge_nodes.pl

Predicate/Directive	Rules	From Internal	To Internal	From External	To External
(user:add_component_node)/3	2	1	1	0	57
(user:change_edge_ids)/4	2	1	2	0	1
(user:change_from_and_to)/4	2	2	0	0	13
(user:edges_to_contract_edge)/4	1	1	0	0	12
(user:gxl_to_gxl_with_contracted_nodes)/3	1	0	4	5	56
(user:outer_node_to_inner_nodes)/2	1	1	0	0	24
(user:outer_nodes_and_remaining_edges_to_outer_gxl_edges)/3	1	1	7	0	14
(user:v_w_to_gxl_edge_ids)/3	1	0	0	1	1
(user:v_w_to_gxl_edges)/2	1	3	0	0	0

Figure 4.16: Predicate Statistics

Predicate Types

We differentiate between two types of predicates:

- *distributed* predicates are predicates, for which rules in several files exist, regardless of, whether the predicate is defined as multifile or not. We retrieve distributed predicates by calling

```
distributed_predicates(-Ps, -Number_Of_Ps).
```

Implementation:

```
1   distributed_predicates(Ps_4, Length) :-
2      rar:package_to_ps(file:_, Ps_1),
3      sublist( is_distributed_p,
4         Ps_1, Ps_2 ),
5      length(Ps_2, Length),
6      findall( [P, File],
7         ( member(P, Ps_2),
8           rar:select(rule, P, File, _) ),
9         Ps_3 ),
10     list_to_ord_set(Ps_3, Ps_4).
11
12  is_distributed_p(P) :-
13     rar:select(rule, P, File_1, _),
14     rar:select(rule, P, File_2, _),
15     not( File_1 = File_2 ).
```

First, we retrieve all predicates of the project (line [2]). Then, we filter the predicates, which consist of rules written in two different files (lines [3-4]). Therefore, we use the method `is_multifile_p/1`. This method searches for a given predicate `P` two rules (lines [13, 14]), which are written into two different files (line [15]). Finally, we just assign all files, containing corresponding predicate rules to the multifile predicates (lines [6-9]) and sort the resulting list `Ps_3` of distributed predicates (line [10]).

- *multifile* predicates are predicates, for which a directive exists, defining the predicate multifile. We retrieve predicates, which are multifile (implementation cf. page 103), by calling

  ```
  predicates_of_type(multifile, -Ps).
  ```

In order to retrieve a table of the predicates, which are multifile or distributed, we call

```
table_multifile_predicates/0.
```

We show the result in a table (cf. Figure 4.17). The table shows each predicate together with the files, in which at least one rule of the predicate exists. Predicates, which are assigned to only one file are multifile predicates and predicates, which are assigned to several files are distributed predicates.

Multifile Predicates: 73; distributed Predicates: 37

Predicate	Rules in File
(user: (-->))/2	sources/nm_reasoning/nmr_interfaces/smodels_interface.pl
	sources/stock_tool/stock_spooler/html_to_pl.pl
	sources/stock_tool/stock_spooler/html_to_pl_hopp.pl
(user: (<=))/2	sources/basic_algebra/basics/functions.pl
(user:cardinality_constraints_calculus)/4	sources/projects/cardinality_constraints/calculus_basic.pl
	sources/projects/cardinality_constraints/calculus_improved.pl
	sources/projects/cardinality_constraints/calculus_opt.pl

Figure 4.17: Multifile and Distributed Predicates

If we want to view the files, in which a predicate property is defined, we call

```
table_defined_predicates(
      +dynamic|multifile|discontiguous).
```

In order to determine predicates, which contain rules in different packages, we call

```
multilevel_predicate(
      +Package_1, +Package_2, -Predicate).
```

Thereby, each package `Package` consists of `L:N`, the level of a packages and the corresponding name or path of the package. `L` has to be bound and `N` can be a free variable.

Implementation:

```
1   multilevel_predicate(L_1:N_1, L_2:N_2, P) :-
2      findall( P-N_1-N_2,
3         ( rar:select(rule, P, File_1, _),
4           rar:select(rule, P, File_2, _),
5           not( File_1 = File_2 ),
6           rar:select(leaf_to_path, File_1, Element_1),
```

```
7           rar:parse_element(Element_1, _, L_1, N_1),
8           rar:select(leaf_to_path, File_2, Element_2),
9           rar:parse_element(Element_2, _, L_2, N_2),
10          not( N_1 = N_2 ) ),
11        Ps_1 ),
12     list_to_ord_set(Ps_1, Ps_2),
13     member(P-_-_, Ps_2).
```

First, we search for a predicate defined in two different files (lines [3-5]). Then, we look to which package with level `L_1` and `L_2` each of the two files belong to (lines [6-7, 8-9]). We retrieve the corresponding level names and compare, them (line [10]). In order not to retrieve duplicated results, we search for all predicates using `findall/3`, then we sort the result and at last, we return the predicates, one by one using `member/2` and backtracking.

4.2.3 File Statistics

Now, we introduce some statistics, which present useful information about files. E.g., we list the average number of rules of a predicate or we show the *files* containing directives, defining certain predicate properties.

We have implemented a method which generates a statistics of the parsed files (cf. Figure 4.18). This statistics lists for each file

- the LoC,
- the number of directives,
- the number of outgoing calls of the directives,
- the number of predicates,
- the number of rules,
- the average number of rules per predicate,
- the average number of LoC per predicate,
- the number of predicates, which are called from predicates of other files (external predicates), and
- the number of calls from external predicates to predicates of the file (incoming calls),
- the number of predicates, which are calling external predicates, and
- the number of outgoing calls.

The call

```
file_statistics:table_file_statistics,
```

generates a statistics table of all parsed files. We can specify the files, which we want to see in the statistics table, too. We just call

```
file_statistics:table_file_statistics(+Files).
```

File Statistics: Files: 749; Red Files: 16; Yellow Files: 48

	Name	Statistics										
	File	LoC	Directives		MPAs				Incoming		Outgoing	
			Directives	Calls Out	MPAs	Rules	Rules/MPA	LoC/MPA	Called MPAs	Calls	MPAs Calling	Calls
1	'library/arrays.pl'	184	1	0	14	35	2,5	13,1	9	12	0	0
2	'library/assoc.pl'	152	1	0	15	27	1,8	10,1	3	3	0	0
3	'library/lists_sicstus.pl'	364	1	0	33	65	2	11	262	278	1	1
4	'library/loops.pl'	204	1	0	8	27	3,4	25,5	124	124	0	0
5	'library/ordsets.pl'	352	1	0	30	87	2,9	11,7	724	909	0	0
6	'library/ugraphs.pl'	819	2	0	85	167	2	9,6	49	82	22	31
7	'sources/basic_algebra/basics/algebraic.pl'	23	0	0	1	2	2	23	0	0	0	0
8	'sources/basic_algebra/basics/arithmetic.pl'	429	0	0	43	80	1,9	10	145	179	3	7
9	'sources/basic_algebra/basics/clause_database.pl'	222	0	0	23	32	1,4	9,7	22	25	10	14

Figure 4.18: File Statistics

In order to speed up viewing this table next time, this generates a temporary database. It is the same database as mentioned in the introduction of Section 4.2 and which is used by the *unit* statistics.

A file, which only contains predicates, which do not call external predicates, and which simultaneously contains no predicates, which are called from external may be unnecessary. In this case, we have to pay attention to the *directives* of such a file. The directives could include parameter settings, which are necessary for the whole source code, e.g., definitions of multifile, dynamic, and discontiguous predicates, or there can be further files consulted in a directive. We highlight such files yellow in the table. If the files *additional* have *no* directives, we highlight them *red*. The label of the table gives information about the number of yellow and red files.

Several files, combined to one PROLOG module is not recommend and can confuse inexperienced programmers. Doing so is unusual in SWI-PROLOG. In order to generate a table (cf. Figure 4.19) of the PROLOG modules, which consists of at least `Number` files, we call

```
module_statistics:table_prolog_module_to_files(
   +Number).
```

The argument `Number` determines the minimum number of files, in which a PROLOG module is split.

Prolog Modules split into at least 2 Files (module "user" is ex

Module	File
visur	sources/source_code_analysis/visur/cross_references.pl
	sources/source_code_analysis/visur/graph_common_methods.pl
	sources/source_code_analysis/visur/graphs.pl
	sources/source_code_analysis/visur/gui.pl
	sources/source_code_analysis/visur/gui_class.pl
rar	sources/source_code_analysis/rar/calls_ll.pl
	sources/source_code_analysis/rar/calls_pp_patch.pl

Figure 4.19: Table, showing the Modules consisting of several Files

Implementation:

```
1   prolog_module_to_files(Number, Module_Files_3) :-
2      get_prolog_modules(Modules),
3      findall( [Module, File],
4         ( member(Module, Modules),
5           rar:select(rule, (Module:_)/_, File, _) ),
6         Module_Files_1 ),
7      list_to_ord_set(Module_Files_1, Module_Files_2),
8      sublist(
9         composed_of_at_least_files(Number, Module_Files_2),
10           Module_Files_2, Module_Files_3 ).
11
12  composed_of_at_least_files(N, Module_Files, [Module, _]) :-
13     findall( File,
14        member([Module, File], Module_Files),
15        Files ),
16     length(Files, L),
17     L >= N.
```

First, we retrieve all PROLOG modules of the project (line [2]). Then, we retrieve for each file the corresponding module name (lines [3-7]). If a file contains a module, the predicates defined in this file belong to this module. Otherwise, they belong to the common module `user`. Afterwards, we filter all modules, which consists of at least `N` files (lines [8-10]).

4.2.4 PROLOG Design Patterns

Now, we describe the detection and the refactoring of patterns (cf. [53]). We detect directly recursive predicates having a special form that could in fact be replaced using the meta-call predicate `maplist/3`. Other interesting patterns 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.

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. Figure 4.20 shows a statistics about how many rules with directly recursive predicates are contained in the DDK units. Sometimes, we can replace these recursive calls using *maplist/2* or *maplist/3*.

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 implementation of `vector_multiply/3` can be obtained using the meta-call predicate `maplist/3`:

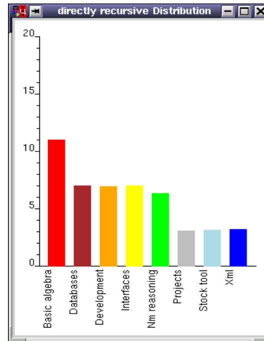


Figure 4.20: Rules with Directly Recursive Predicates (in %)

```
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 the DDK. 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. [44], 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 maintainability:

```
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 `P`.

`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`:

Implementation:

```
1   list_argument(Atom, (M:P)/A, N, X:Xs) :-
2      [M, P, A] := Atom@[module, predicate, arity],
3      Argument := Atom-nth(N)^_,
```

```
4       '.' := Argument@functor,
5       X := Argument-nth(1)^var@name,
6       Xs := Argument-nth(2)^var@name.
7
8    variable_argument(Atom, (M:P)/A, N, Variable) :-
9       [M, P, A] := Atom@[module, predicate, arity],
10      Argument := Atom-nth(N)^_,
11      Variable := Argument@name.
```

Line [3] selects the N-th argument of Atom, independently of its tag, and, if Argument is a list (i.e., its functor is "."), then line [5] selects the head X of the list and line [6] 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.

Example 4.12 (Design Patterns)

```
?- Atom_1 =
      atom:[module:user, predicate:vector_multiply, arity:3]:[
         var:[name:'F']:[],
         term:[functor:'.']:[
            var:[name:'X']:[], var:[name:'Xs']:[]],
         term:[functor:'.']:[
            var:[name:'Y']:[], var:[name:'Ys']:[]]],
      Atom_2 =
      atom:[module:user, predicate:vector_multiply, arity: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 = (user:vector_multiply)/3, X = 'X', Xs = 'Xs', N = 2;
P = (user: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 P, that the list variables Xs and Ys do not 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 P is just an atom with "[ ]" in the positions N, N' and underscore variables in the remaining positions.

An analysis in [53] turned out that there exist 176 maplist patterns in the 4 basic level units of the DDK; 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`. Obviously, over the years the style of programming has changed towards using more meta-call predicates, which are seems to be typical for the PROLOG community.

4.3 Spectra

The spectrum of a predicate consists of the respective number of basic predicates transitively called by the predicate. Basic predicates are all built-in predicates, predicates of the level zero and predicates, which we manually defined as basic predicates.

We have implemented a possibility to assign a *spectrum* to each predicate of the parsed source code. We use this spectrum as a kind of signature of a predicate. Using *predicate spectra*, we assign predicates to packages.

Other spectra are *package spectra*. A package summarizes a set of predicates belonging to the same scope. We can visualize the number of calls from one package to all other packages in a spectrum. Using package spectra helps to design the source code hierarchical.

4.3.1 Predicate Spectra

In order to determine the spectrum of each predicate, we first need to sort the recursive clusters of predicates into the called levels (cf. page 124). All built-in predicates and the predicates of the level zero in the called levels are basic predicates. These predicates do not call further predicates of the examined source code. To calculate the spectrum of the predicates, we start with the predicates of level one and determine their spectra. Then, we consider the predicates of level two, three and so on. Due to the fact of our sorting, a predicate only calls predicates with a lower level. Considering the next higher level, the spectra of all called predicates are already calculated.

To generate the spectrum of a predicate, we just union the already existing spectra of all called predicates. The spectra of the called predicates are already existing, because they have been determined in the step before. This means, in order to determine the spectrum of a predicate, we iteratively add the basic predicates of the transitive closure to the spectrum. Never ending loops are impossible, because we determined cycles, and strong components respectively, before.

The notation of the spectrum of a predicate is a multiset, which consists of terms $BP_i : A_i$, whereby BP_i is the name of the i-th basic predicate and A_i is the corresponding number of calls of the i-th basic predicate. A_i is the sum of all calls in the transitive closure of the predicate. If the number A_i of calls is zero, the corresponding term is omitted in the multiset. In this multiset, the basic predicates are sorted alphabetically.

If we want to generate the temporary database of all predicate spectra, we call

```
create_spectra,
```

and in order to list the multiset of a predicate, we call

```
predicate_to_spectrum(?Predicate, -Multiset).
```

If the temporary database for the predicate spectra is not created yet, then, this first calls the method mentioned above in order to generate the temporary database of all predicate spectra. Then, the multiset and spectrum respectively of the queried predicate is shown.

Example 4.13 (Spectrum of a Predicate) Figure 4.21 shows the rule/goal graph of the following predicate:

```
writeln_list(Xs) :-
   checklist( writeln,
      Xs).

writeln(X) :-
   write(X),
   nl.
```

We call

```
?- predicate_to_spectrum((user:writeln_list)/1, H).

H = [(user:checklist)/2:1, (user:nl)/0:1, (user:write)/1:1]

Yes
```

The resulting multiset contains for each basic predicate the number of transitive calls of the basic predicate. The spectrum of the predicate `writeln_list/1` contains the basic predicates `checklist/2`, `write/1`, and `nl/0`, each called one time.

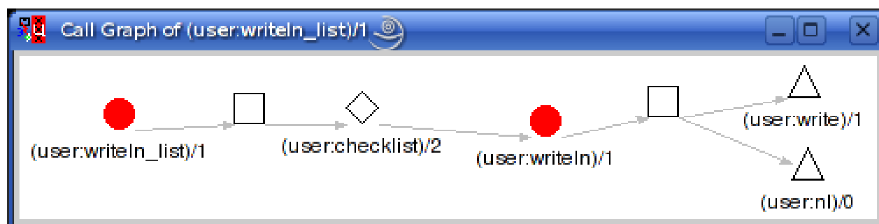


Figure 4.21: Call Graph of writeln_list/1

Example 4.14 (Spectrum of a Predicate)

```
writeln_list_with_notice(Notice, Xs) :-
   writeln(Notice),
   writeln_list(Xs).
```

In this example, we add to the body of the predicate an additional `writeln/1`. The spectrum of this predicate looks as follows:

```
?- predicate_to_spectrum((user:writeln_list_with_notice)/2, H).

H = [(user:checklist)/2:1, (user:nl)/0:2, (user:write)/1:2]

Yes
```

The spectrum of the predicate `writeln_list_with_notice/2` contains one call of the basic predicate `checklist/2`, two calls of the basic predicate `nl/0`, and two calls of the basic predicate `write/1`.

In the following example, we visualize the spectrum of a complex predicate.

Example 4.15 (Spectrum of a complex Predicate)

```
?- predicate_to_spectrum((basic_gxl:gxl_color_set)/5, H).

H = [(user:apply)/2:25,
     (user:assert)/1:8,
     (user:checklist)/2:19,
     ...]

Yes
```

Figure 4.22 shows the spectrum of the predicate `gxl_color_set/5` in a histogram. Each vertical line represents a basic predicate. The basic predicates are listed alphabetically along the x-axis. The height of each line shows how often a basic predicate is called in the transitive closure of the predicate. The predicate of the example calls 53 basic predicates.

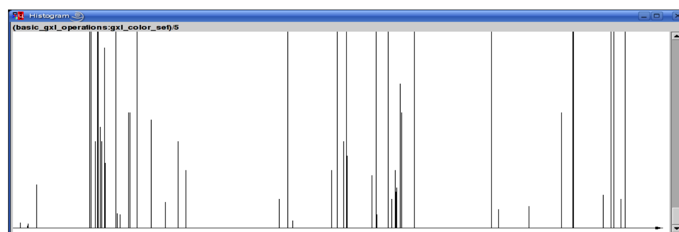


Figure 4.22: Spectrum of a Predicate

To view the spectrum of a predicate, we call

```
spectrum_to_picture(?Predicate),
```

which opens a new picture and visualizes the spectrum of the predicate `Predicate` as shown in Figure 4.22.

Remark: A simple way to count the called basic predicates of a predicate could be to use the Linux method `grep`, which can be used to count all occurrences of a certain expression, e.g., in a file. But using `grep` is not sufficient to generate the spectrum of a predicate. We need the call dependency graph to assign the calls of the basic predicates to the corresponding predicate and to count transitive calls of basic predicates.

Classifying Predicates

We classify predicates on the basis of their spectra and assign predicates to available modules or units of the DDK. Assigning predicates to packages has the advantage that we can move similar predicates together into the same module or unit. Thereby, the clearness of the source code increases and the maintenance of the source code is much easier.

In order to associate predicates to modules or units of the DDK, we first have to generate corresponding package spectra.

Defining Basic Predicates Basic predicates are predicates, which are not defined in the source code. These are the built-in predicates, and all predicates of the level zero of the sorting mentioned on page 124. The basic predicates are automatically determined during the calculation of the predicate spectra, and they are asserted as facts in the temporary database:

```
basic_predicate(BP, N, Packages, Direct_Calls).
```

In order to customize the automatically determined basic predicates, we can change or add manually further basic predicates to the temporary database by retracting and asserting new facts from and in the temporary database. Thereby, a fact consists of the following arguments:

`BP`
: is the name of the basic predicate,

`N`
: is an according to the alphabetical order of the predicate names,

`Packages`
: is a list, which contains the percentage allocation to the packages (modules or units),

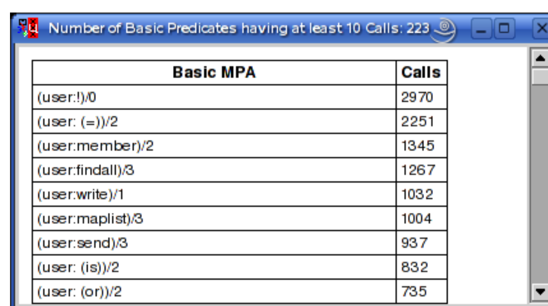
`Direct_Calls`
: is the number of direct calls (*not* transitive calls) in the whole source code. This is equivalent to the result of a search over all source code files using the method `grep`.

We have implemented a statistics, which gives an overview about all basic predicates, too. Calling

```
table_basic_predicate_call_statistics(
      +alphabetical|calls, +Min).
```

creates a table (cf. Figure 4.23) listing the basic predicates and the respective number of calls. Thereby, the argument `alphabetical` determines that the predicates are sorted alphabetically, and the argument `calls` determines that the predicates are sorted by the number of calls. `Min` is an integer, with which we determine the minimum number of calls which a basic predicate has to have in order to be included in the statistics. The headline of the table shows the number of basic predicates. E.g., some of the most frequently called basic predicates are

```
!, =, member, findall, write.
```



Number of Basic Predicates having at least 10 Calls: 223

Basic MPA	Calls
(user:!)/0	2970
(user: (=))/2	2251
(user:member)/2	1345
(user:findall)/3	1267
(user:write)/1	1032
(user:maplist)/3	1004
(user:send)/3	937
(user: (is))/2	832
(user: (or))/2	735

Figure 4.23: Number of Calls of Basic Predicates

Basic Predicate Association For each basic predicate we determine the number of direct calls. We prorate the calls of each basic predicate to the packages by determining to which packages the caller predicates belongs to. Then, we normalize each number of calls by dividing the number of basic predicate calls through the number of all direct calls. For each basic predicate we retrieve a multiset

```
[Package_1:N_1, Package_2:N_2, ...]
```

of packages. Thereby, `Package_i` is the name of the package and `N_i` is the percentage of direct calls of the basic predicate from predicates belonging to the package. Packages containing no predicates calling the basic predicate are not listed in the multiset. All basic predicate associations are calculated at once by calling

```
assign_bps_to_package(+module|unit).
```

The result of each basic predicate is stored in the fact `basic_predicate/4`, which is mentioned above. Depending on whether we want to assign the basic predicates to *module*- or *unit*- packages, the argument of this method is `module`, or `unit`.

Example 4.16 (Basic Predicate – Unit Allocation) The allocation of the basic predicate `(user:checklist)/2` with respect to the *units* of the DDK looks as follows:

```
?- basic_predicate((user:checklist)/2, _, S, DC).

S = [ basic_algebra:6.7,
      databases:1.1,
      development:0.8,
      nm_reasoning:3.2,
      projects:32.1,
      source_code_analysis:28.1,
      stock_tool:15.9,
      xml:12.1 ]
DC = 371

Yes
```

This means that 6.7 % of the 371 direct calls calling `(user:checklist)/2` are from the unit `basic_algebra`, 1.1 % are from the unit `databases`, 0.8 % are from the unit `development`, 3.2 % are from the unit `nm_reasoning`, 32.1 % are from the unit `projects`, 28.1 % are from the unit `source_code_analysis`, 15.9 % are from the unit `stock_tool`, and 12.1 % are from the unit `xml`. Units having 0.0% are not listed.

The method

```
table_basic_predicate_call_statistics_ext(
   +alphabetical|calls, +Min).
```

creates a table (cf. Figure 4.24) listing the basic predicates, the respective number of calls, and the packages to which a basic predicate belongs to.

Number of Basic Predicates having at least 100 Calls: 44

Basic Predicate	Calls	Package	Percentage
(user:!)/0	3632	library	2.8 %
		sources/basic_algebra	18.1 %
		sources/databases	12.3 %
		sources/development	1.8 %
		sources/nm_reasoning	10.1 %
		sources/projects	14.1 %
		sources/source_code_analysis	21.1 %
		sources/stock_tool	9.5 %
		sources/xml	10 %
(user: (<))/2	198	library	4.5 %
		sources/basic_algebra	15.2 %

Figure 4.24: Package Association of Basic Predicates

After we have generated the predicate spectra and the classification of the basic predicates, we assign each predicate to several packages. Each package has a percentage quota, which is proportional to the number of basic predicate-calls. We retrieve the list of package-associations calling

```
predicate_to_package(+module|unit, +P, -Packages).
```

We determine the predicate/package association in the following way: first, we retrieve the spectrum of the predicate. The spectrum contains the number of all called basic predicates of the *transitive closure* of the predicate. Then we query to which packages the corresponding basic predicates belong to. We multiply each package percentage of a basic predicate with the number of calls of the basic predicate and then, we summarize identical packages. At last, we normalize the package percentage.

Example 4.17 (Classifying Predicates)

```
?- predicate_to_package(unit, (user:writelq)/1, MS).

MS = [ library:0.129032
      basic_algebra:8.3871
      databases:4.45161
      development:0.645161
      nm_reasoning:26.1525
      projects:9.80645
      source_code_analysis:25.566
      stock_tool:5.48387
      xml:19.3783 ]

Yes
```

Calling

```
polar_diagram_predicate_package_association(
       +module|unit, +Border, +Predicate).
```

visualizes the package-association of a predicate in a polar diagram (cf. Figure 4.25). Thereby, `module`, or `unit` determines the package size, `Border` determines the minimum relevance of the corresponding package in percent, and `Predicate` is the predicate, which partition we want to visualize.

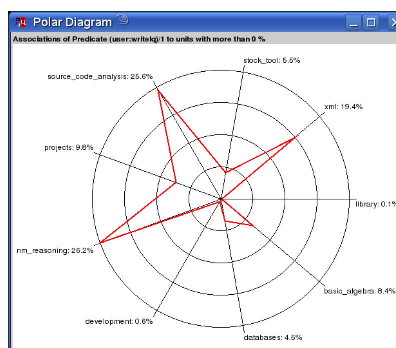


Figure 4.25: Polar Diagram showing the Package Association of a Predicate

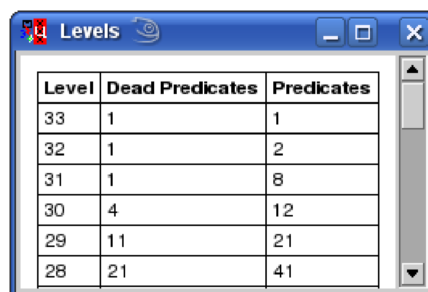
Called Level Sorting

We use the call dependencies between predicates in order to sort all predicates into *called levels*. Thereby, the called level 0 contains those predicates, which are not calling further predicates. These are, e.g., the built-in predicates, and undefined predicates. We name the predicates of the called level 0 *basic predicates*. The basic predicates are used to define the spectra.

A predicate which calls another predicate, has the same or a higher level than the called predicate. If the predicates are mutually recursive, then they belong to the same level; otherwise, the calling predicate belongs to a higher level than the called predicate. This sorting is equivalent to a topological sorting of the predicates, with the exception, that predicates belonging to the same strongly connected component are sorted into the same called level.

As it is not possible to generate a topological sorting on source code, which has cycles, we first determine the strongly connected components of the considered source code (cf. Section 4.1.2). All elements of the same strongly connected component are considered as one predicate, having the same level in the called level sorting.

Figure 4.26 and Figure 4.27, shows the result of the sorting of the DDK. In Figure 4.26, the first column shows the level of the sorting, the second column shows the number of predicates, which are not called from another predicate, and the last column shows the number of predicates belonging to this level. Predicates, which are not called could be starting predicates of calculations or GUIs. These predicates are comparable with, e.g., the `.exe` files of a *DOS* operating system.



Level	Dead Predicates	Predicates
33	1	1
32	1	2
31	1	8
30	4	12
29	11	21
28	21	41

Figure 4.26: Sorting of Predicates

In order to create the statistics table of the sorting, we just call

```
table_sorting.
```

If this is the first time we call this method, this generates two temporary databases (cf. Section 4.2). First, it generates a database of the strongly connected components, if this database does not yet exist. Second, it generates a temporary database of the sorted predicates, which is used every time the table or the histogram is requested. This temporary database can be queried, calling the facts

```
sorting(?Predicate, ?Level,
   root|no_root|directive|excluded_dead_code).
```

The predicates can be divided into diverse categories. The last argument of the facts determines to which category a predicate belongs to.

`root`
: This attribute means, that the corresponding predicate is a root in the sorted called levels. Predicates, which have the attribute `leaf` are not called in the source code. These predicates may be dead code (cf. Section 4.2.1).

`no_root`
: This attribute means, that the corresponding predicate is not a root. These predicates are called from other predicates in the source code.

`directive`
: These predicates belong to the directives in the source code. The directives are not explicitly called from another predicate in the source code, but they are executed once after a file has been consulted.

`excluded_dead_code`
: These are predicates, which are not called, but which are manually excluded of the list of dead code (cf. Section 4.2.1).

To view the histogram of the predicate sorting (cf. Figure 4.27), we call

```
histogram_sorting.
```

Thereby, the dark lines represent the number of predicates belonging to a certain level and the bright part of the line shows the number of predicates, which are not called in this level.

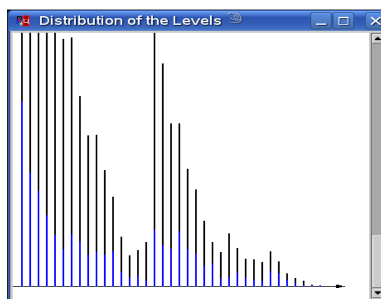


Figure 4.27: Distribution of the Predicates in the Called Levels

We show the statistical distribution of the levels of a certain package, e.g., a `unit`, in polar diagrams. Therefore, we call

```
sorting_to_polar_diagram(
      +sources|unit|module|file:?Name).
```

The polar diagram shows all available levels. The number of predicates in a package is shown by the red line (cf. Figure 4.28).

Example 4.18 (Polar Diagram of the Called Levels)

```
called_level:sorting_to_polar_diagram(
      unit:'sources/source_code_analysis').
```

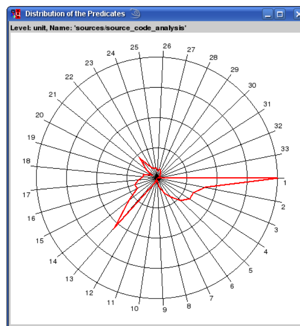


Figure 4.28: Distribution of the Predicates of the Called Levels of a Package

4.3.2 Package Spectra

We can visualize the number of calls from one package to all other packages in a spectrum. In order to retrieve all outgoing calls between a package and all other packages, we have implemented the following method:

Implementation:

```
1   outgoing_calls(Level:Path_1, Multiset) :-
2      rar:package_to_ps(Level:Path_1, Ps),
3      findall( Path_2,
4         ( member(P_1, Ps),
5           rar:calls(predicate:P_1, predicate:P_2),
6           rar:contains(Level:Path_2, predicate:P_2),
7           not( Path_1 = Path_2 ) ),
8         Calls ),
9      list_to_multiset(Calls, Multiset).
```

Line [2] retrieves all predicates `Ps` defined in package `Path_1`. Then, we search for all predicates, which are called from a predicate of `Ps` (lines [4, 5]). We determine the corresponding package of `P_2` and proof, that the called predicate does not belong to the same package (lines [6, 7]). At last, we generate a multiset.

Figures 4.29, 4.30, 4.31, 4.32 show the number of outgoing calls of the DDK units in diagrams. Thereby, the number in the headline of each diagram shows the internal calls of the corresponding unit. In each unit, the maximum of calls are internal calls. Further on, we see that the units

`basic_algebra, databases, nm_reasoning, xml`

are the basic units, which are called from the other units.

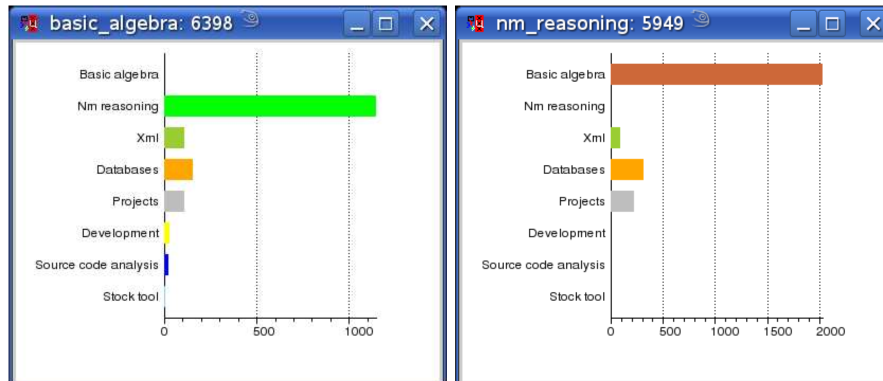


Figure 4.29: Calls of the Units *basic_algebra* and *nm_reasoning*

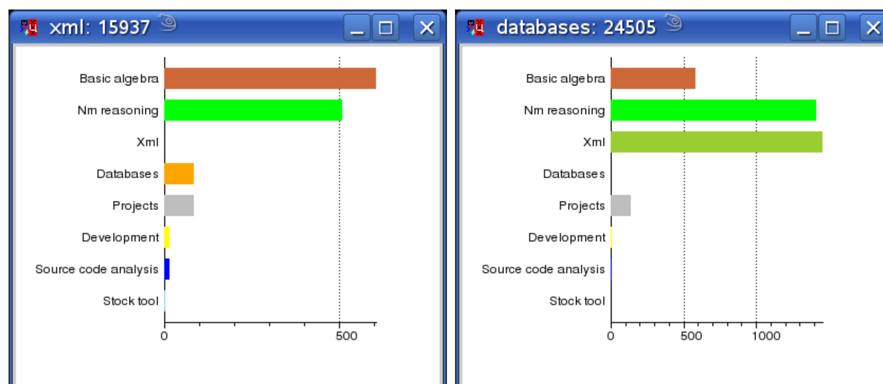


Figure 4.30: Calls of the Units *xml* and *databases*

Figure 4.33 visualizes the number of call dependencies between the units. We have faded out edges between units, which represent only few calls.

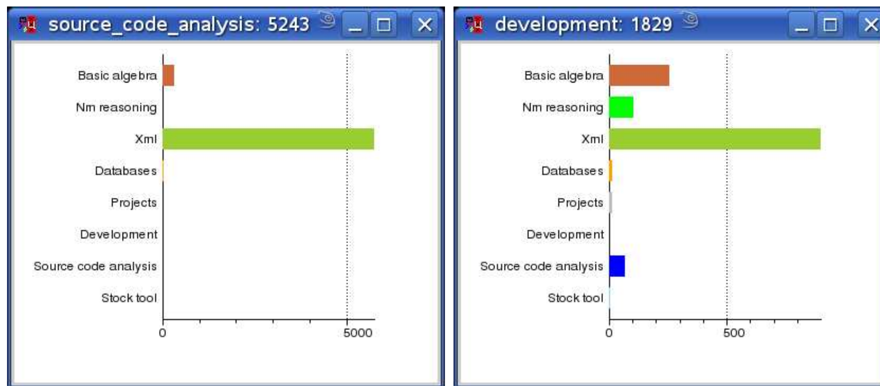


Figure 4.31: Calls of the Units *source_code_analysis* and *development*

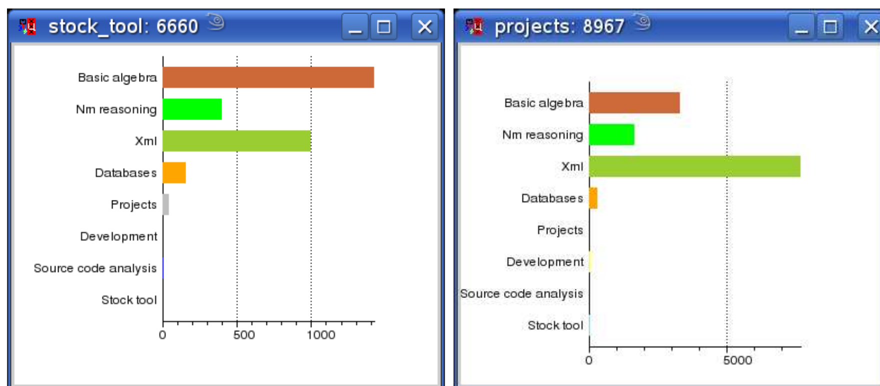


Figure 4.32: Calls of the Units *stock_tool* and *projects*

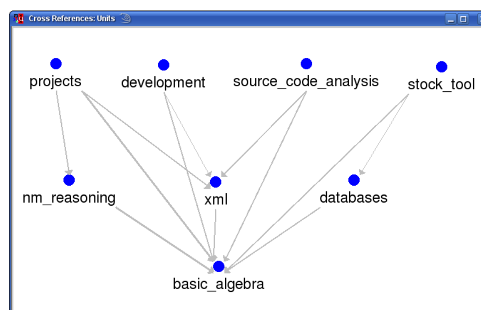


Figure 4.33: Unit Dependency Graph of the DDK in SCAV

Chapter 5

Slicing

> "A *program slice* consists of the parts of a program that affect the values computed at some point of interest. Such a point of interest is referred to as a *slicing criterion*, and is typically specified by a location in the program in combination with a subset of the program's variables. The task of computing program slices is called *program slicing*. The original definition of a program slice was presented by Weiser in 1979. Since then, various slightly different notions of program slices have been proposed, as well as a number of methods to compute them. An important distinction is that between a static and a dynamic slice. Static slices are computed without making assumptions regarding a program's input, whereas the computation of dynamic slices relies on a specific test case." [59]

Generally, in literature, *program slicing* is a technique to identify statements that may influence the computation in other statements. Program slicing can be distinguished in *forward* and *backward* slicing. Searching all statements, which influence a statement is called *backward slicing*, and searching all statements, which are influenced by a statement is called *forward slicing* [27].

Our slicing method is a kind of *static forward program slicing*. We have adapted the general definition of program slicing as it is described, e.g., in [26, 27, 28, 29, 30, 31, 59]. We search for and extract all methods of a large software project, which can be reached in the extended call dependency graph from a certain method. Our purpose is to have a full functionally and stand alone part of a large software system, including all necessary methods and variable settings. In [24], we demonstrate how we slice PROLOG source code using SCAV and FNQUERY/FNSELECT. If we just want to use a certain functionality of a large software system, then we can extract the parts of a system which are necessary to run this functionality. Reasons for obtaining a slice of a software system are debugging and refactoring a certain part of the system or exporting methods to another system.

We start slicing with a predicate, which is responsible for a certain program functionality. We extract all further called predicates in the call dependency graph of the software system, which are necessary to run the chosen predicate, and we try to find all corresponding variables, including the proper variable settings, too. Finally, we complete the slice

by adding corresponding tests to the slice.

This may be necessary, e.g., for debugging, or if we want to make only a part of the source code available to someone else. In order not to confuse him with all functionality of a software system, we don't want to give him the whole source code. We only give him this necessary part of the source code. Therefore, we extract the parts, which belong to a certain functionality. This kind of slicing is useful, too, if we want to port a large project from one platform to another platform. We can port step by step parts of the source code. In each step, we can adapt the necessary parts of a slice to the new platform.

5.1 Basic Concepts

In the following, we first describe the method, which generates the transitive closure of a predicate. Then, we describe the configuration and execution of the slicing method. We have the possibility to slice more than one predicate in one step. The slicing method applies step by step – depending on the user configuration – the methods which we introduce in Section 5.2.

Transitively Depended Predicates

In order to calculate the transitive closure of one or more predicates, we use the *extended predicate dependency graph* (call graph for predicates) of the source code (cf. Section 4.1). In the extended predicate dependency graph, we consider if a predicate is called from a meta-call predicate and include these calls.

The following call, which is defined in the PROLOG module `slice`, calculates the transitive closure. Additionally, we give the list of files, in which the predicates of the transitive closure are defined.

```
slice:tc_calls(+Ps, -TC_Ps, -TC_Files),
```

`Ps`
: is a list of predicates `(M:P)/A`, which we want to slice.

`TC_Ps`
: contains the names of the predicates in the transitive closure, respectively, it contains all predicates, from which the predicates `Ps` are depending.

`TC_Files`
: contains the files, in which the resulting predicates are defined.

Slicing

In this section, we describe the configuration of the slicing method. In order to slice several methods and to create a working subsystem of a large software project, we call

```
slice(+XML, -Necessary_Files, -Necessary_Ps).
```

This call creates a subdirectory with a random generated name in the directory:

```
DisLog/results/slicing/
```

We copy all necessary files including their paths into this subdirectory. Additionally, we create the file `slice_package.pl`. This file contains the *consult directives* of all files of the slice and it can be used in order to load the slice easily. The consulted files in the file `slice_package.pl` are ordered in the proper order, as it is specified in the original consulting order of the whole system.

`XML`
: contains the input, e.g., the predicates, which we want to slice and some further arguments, which specify the execution of the methods of Section 5.2. We describe the input XML document and its DTD in the following.

`Necessary_Files`
: This list contains the necessary files of our search having the proper consulting order as it is specified in the original source code. The corresponding files, which declare the predicate properties, e.g., the directives of `multifile`, `discontiguous` or `dynamic` predicates are included, too.

`Necessary_Ps`
: This list contains the necessary methods of our slicing.

The DTD of the XML document `XML`, which configures the method `slice/3` looks as follows:

DTD:

```
<!ELEMENT slice (predicates* files*)>
<!ATTLIST slice
   multifile (true|false) #required
   dynamic (true|false) #required
   discontiguous (true|false) #required
   twins (true|false) #required
   current_num (true|false) #required
   directives (true|false) #required
   consults (true|false) #required
   tests (true|false) #required>

<!ELEMENT predicates atom*>
<!ATTLIST predicates
   type (include|exclude|dynamic|
         multifile|discontiguous) #required>

<!ELEMENT atom EMPTY>
<!ATTLIST atom
```

```
    module CDATA #required
    predicate CDATA #required
    arity CDATA #required>

<!ELEMENT files file*>
<!ATTLIST files
    type (include|exclude|libraries) #required>

<!ELEMENT file EMPTY>
<!ATTLIST file
    path CDATA #required>
```

Attributes of the element `<slice>`:
: We configure the slicing of a predicate using diverse attributes.

`multifile`, `dynamic`, `discontiguous`
: If one of these booleans is set to `true`, we are trying to find out if there exists predicates in the slice result, which are defined as `multifile`, `dynamic` or `discontiguous` predicates. If we find such predicates, we check, if the corresponding directives are contained in the slice result and if not, we add the files containing the corresponding directives to the slice result. We can skip this search, because it is a time-consuming search.

`twins`, `current_num`
: If one of these booleans is set to `true`, we search for certain variables (cf. Section 5.2) used in the slice results and add all files in which these variables are set to the slice result. The variables for which we are looking are defined in the body of the method `twin_predicates_to_files/2`, which is contained in the file `slice\slicing.pl`.

`directives`
: If this boolean is set to true, we search for all directives in the slice result and add the files containing the rules of predicates called in the directives to the slice result.

`consults`
: If this boolean is set to true, we search for files, which are consulted in the files of the transitive closure and we add these files to the slice.

`tests`
: If we want to include the corresponding tests of a slice, too, we set the value of this variable `true`. Normally, the tests of a slice are not completely contained in the slice. Often necessary predicates of the tests, or the test predicates themselves are missing.

Attributes of the element `<predicates>`:
: We differentiate between several types of predicate, which we can add to the slice result or exclude from the slice result. We configure our tack using the attribute `type`. `type` can have the following values:

`include`
All methods, which we want to slice are contained in this element.

`exclude`
If we want to exclude certain methods from the slice, then we list them in this element.

`dynamic`, `multifile`, `discontiguous`
Methods, which we want to define additional `dynamic`, `multifile` or `discontiguous` are contained in this element. This is useful, if we deactivated the argument `dynamic`, `multifile` or `discontiguous` of the element `<slice>` in order to save runtime. If we know, which methods have to be defined `dynamic`, `multifile` or `discontiguous`, then we can list these methods here.

Attributes of the element `<files>`:
We differentiate between three types of files, which we can include in the slice result or exclude from the slice result. We configure our tack using the attribute `type`. `type` can have the following values:

`include`
Files, which we want additional include to the slice result.

`exclude`
Files, which we want exclude from the slice result.

`libraries`
Files, which we want to be added to the slice result, but which we do not want to be consulted, because they are libraries, which will be loaded independently, e.g., using `use_module`.

Example 5.1 (Slice)

```
?- Slice =
     slice:[multi:false, dyn:false, discont:false,
            twins:false, current_num:false, directives:false,
            tests:true, consults:true]:[
       predicates:[type:include]:[
         atom:[module:user, predicate:dread, arity:3]:[],
         atom:[module:user, predicate:dwrite, arity:3]:[],
         atom:[module:user, predicate:'<=', arity:2]:[]],
       predicates:[type:exclude]:[],
       predicates:[type:dynamic]:[
         atom:[module:user, predicate:test, arity:2]:[]],
       predicates:[type:multifile]:[
         atom:[module:user, predicate:test, arity:2]:[]],
       predicates:[type:discontiguous]:[],
       files:[type:include]:[
         file:[path:'library/loops.pl']:[]],
       files:[type:libraries]:[
```

```
            file:[path:'library/static_swi.pl']:[],
            file:[path:'library/lists_swi.pl']:[]],
          files:[type:exclude]:[] ],
      slice:slice(Slice, Files, Preds).

Yes
```

Result Report about the Slicing

After we have generated a slice, we can calculate some statistics about the slice, e.g., how many methods of the whole system are used in the slice. We display the results in percent for each `unit`, `module`, and `file` in a table (cf. Figure 5.1).

We calculate this statistics and extend the XML hierarchy tree of Section 3.1.2 by adding the corresponding values to the tree. Then, we display the content of the *extended* XML hierarchy tree in a table. In order to generate the extended XML hierarchy tree of the table, we need the names of the necessary files and the names of necessary predicates of the slice. We obtain this information from the slicing method. Additionally, we need the hierarchy tree of the system, which we obtain from the corresponding variable. The method

```
statistics:tree_to_statistics_tree(+Files, +Ps,
       +Tree, -Extended_Tree)
```

generates the extended hierarchy tree. The *extended* DTD of the tree contains the additional attributes

```
chosen, loc, predicates, necessary, rules
```

which we add to the tree elements

```
<system>, <sources>, <unit>, <module>, <file>.
```

DTD:

```
<!ELEMENT sub_level sub_level* file*>
<!ATTLIST sub_level
   chosen CDATA #required
   loc CDATA #required
   predicates CDATA #required
   necessary CDATA #required
   rules CDATA #required
   ...>

<!ELEMENT file sub_level*>
<!ATTLIST file
   chosen CDATA #required
   loc CDATA #required
   predicates CDATA #required
   necessary CDATA #required
   rules CDATA #required
   ...>
```

`loc`
: number of lines of source code included in the current tree element, which represents the system, a unit, module, or file. E.g., if the element represents a file, `loc` just contains the lines of source code of the file. If the element represents a module, `loc` represents the sum of all loc of the corresponding files, belonging to the module.

`predicates`
: number of predicates included in the current tree element, which represents the system, a unit, module, or file.

`rules`
: number of rules included in the current tree element, which represents the system, a unit, module, or file.

`necessary`
: number of methods, which are necessary for the slice of this element. Not every method of a file is necessary in the slice, because we always put the whole file to the slice, even if only one method of the file is needed.

`chosen`
: number of methods, which are chosen in the slice. This number is often larger than the necessary number of predicates, because we always summarize all methods of the files, and not single methods. Therefore, the number `chosen` and `predicates` for `files` is always the same.

In order to display this statistics in a table, we call

```
statistics:table_slice_statistics(+Extended_Tree,
      +Columns, +Limit).
```

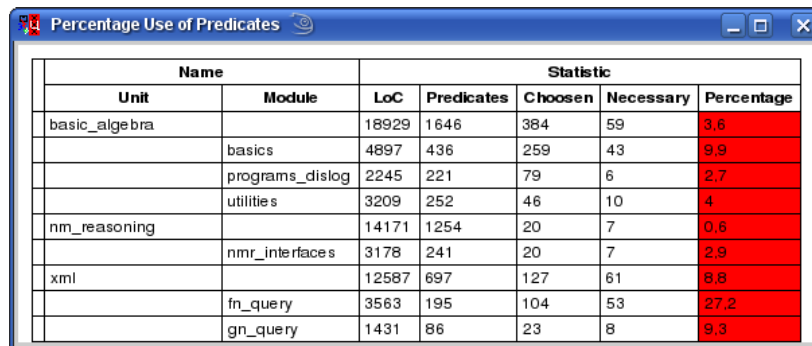
`Extended_Tree`
: contains the statistics results of the *extended* XML hierarchy tree mentioned above.

`Columns`
: is a list with an arbitrary combination of the elements

```
(system, 'System'),
(sources, 'Sources'),
(unit, 'Unit'),
(module, 'Module'),
(file, 'File').
```

Each element determines a column, which we see in the statistics table.

`Limit`
: is a percentage between 0 and 100. We specify a minimum of used predicates in percent. Only rows having a larger value will be shown in the table, e.g., 30 means that only rows with a percent usage of 30 percent and more of the predicates will be displayed in the table.



Name		Statistic				
Unit	Module	LoC	Predicates	Choosen	Necessary	Percentage
basic_algebra		18929	1646	384	59	3.6
	basics	4897	436	259	43	9.9
	programs_dislog	2245	221	79	6	2.7
	utilities	3209	252	46	10	4
nm_reasoning		14171	1254	20	7	0.6
	nmr_interfaces	3178	241	20	7	2.9
xml		12587	697	127	61	8.8
	fn_query	3563	195	104	53	27.2
	gn_query	1431	86	23	8	9.3

Figure 5.1: Percentage of Use of the Predicates in each Unit and Module of the Sliced Predicate slice/3

Example 5.2 (Visualizing the Statistics Tree) The following code slices a predicate, retrieves the source tree, transforms the tree to the statistics tree using the result of the slice (files and predicates), sets the visualized columns and the limit and visualizes the statistics tree in a table (cf. Figure 5.1). Thereby, `Slice` is, e.g., taken from Example 5.1.

```
?- Slice = slice:[...]:[...],
   slice:slice(Slice, Files, Ps),
   sca_variable_get(source_tree, Tree),
   statistics:tree_to_statistics_tree(Files, Ps, Tree, E_Tree),
   Columns = [(unit, 'Unit'), (module, 'Module')],
   Limit = 25,
   statistics:table_slice_statistics(E_Tree, Columns, Limit).

Yes
```

5.2 Special PROLOG Constructs

Our slicing method is divided into several steps. Each step deals with a special PROLOG construct. We consider the problems of the *test predicates*, *predicate properties*, *variable settings*, *directive calls*, and *further consulted files*. We describe these steps in the following subsections. Depending on the user settings, each step can be switched on or off. The user can determine, if he needs a certain slice functionality and if he wants to invest some runtime in the corresponding step, or if he wants to exclude a step from the

processing in order to save time. In the following subsections, we describe each step in detail.

5.2.1 Directives

It might be the case that there exist *directives* in the files of a slice. PROLOG *directives* are executed during consulting a file. In order to avoid error messages during consulting a file of the slice which has directives, the methods occurring in the directives have to be sliced, too. First, we collect all methods, which are called from directives in the slice. Then, we delete the methods of the directives, which are already in the slice result. Finally, we slice the remaining methods of the directives and add the result to the slice.

It might be the case that there are directives in the added files, too. Hence, we call the following method recursively until no further predicates are added.

```
directive_ps(+Necessary_Ps, +Necessary_Fs,-Ps, -Fs).
```

`Necessary_Ps`
: is the result of the sliced method and contains a list of the necessary methods.

`Necessary_Fs`
: is the result of the sliced method and contains a list of the necessary files.

`Ps`
: this list contains the methods, which should be added to the slice result.

`Fs`
: contains the files, which contain necessary methods for directives and which should be added to the slice result.

Dynamic, Multifile, and Discontiguous Declarations

PROLOG source code sometimes contains declarations, which specify additional properties of a predicate, e.g., *dynamic*, *multifile*, and *discontiguous* declarations. Such declarations are defined in directives and look like the following example.

Example 5.3 (Predicate Declarations)

```
:- multifile test/2, test/3.
:- discontiguous test/2.
:- dynamic temporary_result/4.
```

The declarations of predicate properties can be defined anywhere in the source code. We scan all files of the whole source code and search for directives containing the properties *dynamic*, *multifile* and *discontiguous*. If we find such a directive, then we compare the predicates defined in the directive and the predicates of the slice. If a predicate of the directive is contained in the slice, then we add the file containing the declaration to the slice. In order to find the files with necessary predicate declarations, we call

```
files_defining_p_property_in_slice(
      +Type, +Necessary_Files, -Files).
```

`Type`
: can be `dynamic`, `multifile` or `discontiguous`.

`Necessary_Files`
: is the result of the sliced method and contains a list of the necessary files.

`Files`
: The additional files, in which predicates of the slice are defined *dynamic*, *multifile* or *discontiguous* and which should be added to the slice result.

5.2.2 Consults

There exists different possibilities to summarize the files of a software project and to pass the files to PROLOG, such that PROLOG consults these files. There exists a list, containing all corresponding files of a software project. But sometimes, additional files are consulted from a file inside the project. We assume that someone who consults a file in another file aims something by doing this. We are able to search for additional files, which are consulted in the files of the slice, and we can add these files to the slice, too, no matter whether the methods in the additional consulted files are used by the sliced method or not. In order to find the consulted files of a given set of files, we call

```
consulted_files(+Files, -Consulted Files).
```

`Files`
: are the files of the slice.

`Consulted_Files`
: are the files, which are consulted from files in the slice.

5.2.3 Global Variables

It is a common technique to access variables not directly. Often a special implemented *get* or *set* method is used, to get or set variable values. We call these predicate pairs *twin predicates*. Twin predicates are, e.g.,

```
get_num/2 - set_num/2,
dislog_flag_get/2 - dislog_flag_set/2,
sca_variable_get/2 - sca_variable_set/2,
dislog_variable_get/2 - dislog_variable_set/2.
```

In order to retrieve the variable settings including the values of a variable, we could search for predicates having the prefix or suffix *get* or *set*. But this is not a sure method, because, it could be that someone does not use this prefix or suffix. We use a list, in which all twin predicates are defined and we only select those twin predicates, which appear in the rules of the transitive closure.

It could be the case that a certain variable is set in a file, which does not occur in the transitive closure, but which is needed, if we want to run a certain method. If the *get* method of a twin predicate appears in a file of the transitive closure, we add all files, which contain the *set* method of the twin predicate to the files of the transitive closure, too.

```
twin_predicates_to_files(+Necessary_Ps, -Files).
```

`Necessary_Ps`
: is the result of the sliced method and contains a list of the necessary predicates.

`Files`
: are the additional files, which contain necessary twin predicates and which should be added to the slice result.

There exists no additional configuration file for the twin predicates. The list of the current twin predicates is written into the body of this rule.

Additionally, we search for files containing predicates, which assert the fact

```
current_num.
```

Often, in PROLOG systems, this variable is asserted or retracted without using special methods, and sometimes, there are necessary arguments set by asserting facts with the name `current_num`.

```
find_files_containing_current_num(
   +Necessary_Ps, -Files).
```

The arguments `Necessary_Ps` and `Files` are analogous to the arguments mentioned above.

5.2.4 Test Predicates

It is a common technique to add *test predicates* to a software project. These predicates test the correct working of important methods. In order to be sure that a method works correct and that we do not change the behavior of a method, especially after refactoring a method, we need these tests. Slicing a method causes that the corresponding tests of the method are missing. All tests of the sliced methods are no more or not completely present in the slice, and have to be added in a separate step again.

In the DDK, most tests have the predicate name `test/2`. Just slicing the method `test/2` causes that we retrieve all tests of all methods. If the tests are defined in one file, we cannot extract the corresponding tests of a sliced method. However, this is not probably. Hence, we assume that the corresponding tests of a sliced method are written in an own file. In order to retrieve only the tests, which belong to a certain sliced method, we first calculate the slice without the test predicates. Having a list of all necessary methods, we create a *backward slice* of these methods. This means, we search for all methods iteratively, which call one of the methods belonging to the slice result. Then, we search this result for methods with the predicate name `test/2`, and keep in mind, in which files the test methods are defined.

The result of this procedure contains only test predicates, having a high probability in belonging to the slice. After this, we search for all methods, which are called by these test methods, considering the files, in which these tests are defined. In the next step, we slice the resulting methods, which are called by the located test predicates.

Sometimes, we get some more test predicates than necessary; if further tests are defined in a file in which a necessary test predicate of the slice is defined, then these tests are added to the slice, too, even though they are not belonging to the sliced method.

Part III

Visualization of Source Code

VISUR is used to visualize results, such as graphs, diagrams or tables. E.g., VISUR can be used to visualize call dependencies between methods or packages, links of websites, and DTDs in the hierarchy browser. We explain how to construct and visualize PROLOG call dependency graphs and cross references between packages.

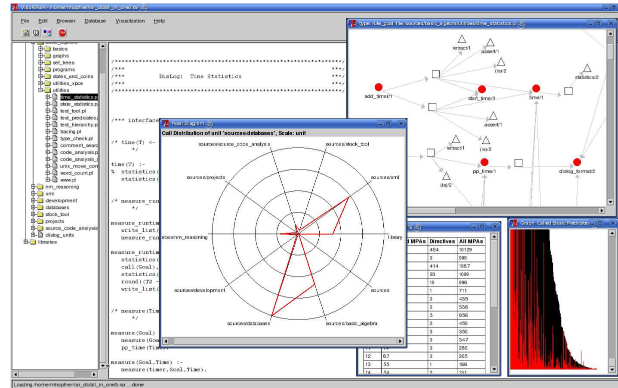


Figure 2: VISUR

GXL is a general Graph eXchange Language, which is used to exchange graphs between diverse programs. Using GXL, we can export graphs to other tools, execute tool specific calculations and import the result into VISUR. The graphs are represented by XML elements for their nodes and edges, respectively.

There are various methods for transforming a GXL document into another document. We have implemented methods for layouting graphs, validating, deleting or highlighting nodes or edges, and merging nodes into a single node. The transformations are configured by XML documents. Figure 3 shows a GXL graph with integrated bar charts as node symbols.

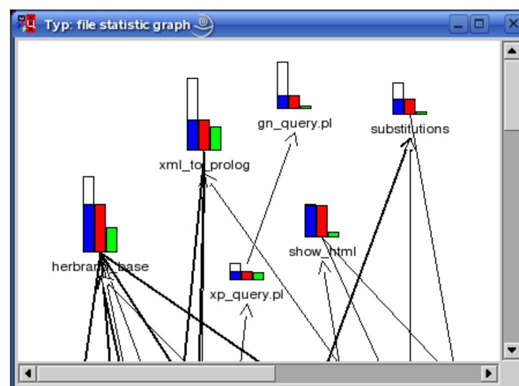


Figure 3: A Graph with Bar Charts

VISUR To visualize a GXL document, we have implemented the class *picture*. We can configure pop-up menus for each node, edge or the picture. We can extract the GXL document of a visualized graph in an arbitrary picture. VISUR visualizes the hierarchy of the source code in a hierarchy browser (left side in Figure 2). We have chosen parts of this hierarchy to perform methods on parts of the source code. Each item of the hierarchy browser contains a pop-up menu, which allows certain actions, which are adapted to the type of the chosen item. The content of a chosen file is visualized in the right window.

VISUR can be used in a wide area of source code visualization and reasoning. Having a suitable XML representation of the source code, e.g., PROLOGML, JAML [12] or PHPML [14], and having adapted basic methods, we apply methods for visualization, statistics, code reasoning, and refactoring. We generate cross reference graphs between different packages. In order to visualize dependency graphs containing meta calls, we have defined the extended Rule/Goal graph. Figure 5.2 shows, e.g., the meta-call predicate *findall/3*, which is called by `calls_uu/2` and by `calls_uu_reduce/3`. Predicates belonging to the group of meta-call predicates are able to call other predicates in their arguments. We have extended the rule/goal graph in order to include meta-calls. We show a visualization of rule based knowledge represented by XML documents. Fi-

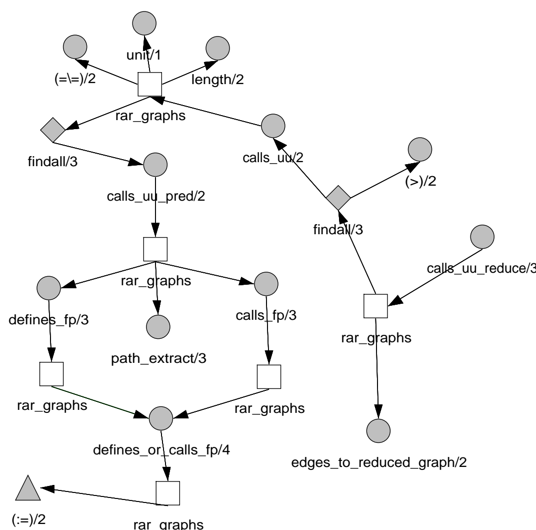


Figure 5.2: Extended Rule/Goal Graph in VISUR

nally, we visualize ER diagrams, and DTDs and give some ideas how to visualize HTML documents.

Chapter 6

A Graph Library Based on GXL

We have implemented a Graph library based on GXL, which can treat graphs in GXL documents [19, 64]. There are different possibilities to obtain GXL documents, e.g., GXL documents can be the export result of a graph visualization tool, it can be the result of calculations of SCAV based on the PROSORE database (cf. Chapter 3, Chapter 7), or it can be a manually created graph saved in a file.

> "GXL is a general Graph eXchange Language in XML, which is designed to be a standard exchange format for graphs. GXL is an XML sublanguage and the syntax is given by a XML DTD (Document Type Definition). This exchange format offers an adaptable and flexible means to support interoperability between graph-based tools." [64]

The library GXL has been used in several applications; e.g., the diploma thesis [14], [68] use the GXL library in order to visualize graphs.

We have extended the GXL DTD and implemented multiple methods, which operate on GXL documents, e.g., generating a graph, calculating different layouts for graphs, saving and loading GXL documents or visualizing the embedded graph of a GXL document in an XPCE picture. In order to use GXL documents together with other tools, our extensions can be easily added or extracted.

All XML documents, which are passed to PROLOG methods, are represented in field notation.

6.1 Layout of Extended GXL Documents

The GXL DTD can be extended by redefining certain entities of the GXL DTD, which are specially designed for this purpose. Each extension, is embedded in the newly defined element `<prmtr>`. If we want to get a pure GXL document, then we just have to remove all `<prmtr>` elements from the GXL document. In the following subsection, we describe the main parts of the original GXL DTD, and we describe our specific extensions of the GXL DTD. For further details on a pure GXL document and the original DTD of a GXL

document see [19, 64]. In Section 6.1.1, we give, among other things, an example of a default GXL document including our extensions.

6.1.1 The GXL Document Extensions for Layout

We extend the three main GXL elements <gxl>, <node> and <edge>. Using the extensions of these elements allows us to configure the layout of the nodes, the edges and the picture. Furthermore, we define user defined menus and pop-ups, as well as diverse actions, which will be executed by clicking on a picture, node or edge. In the following subsections, we describe the extended GXL elements with their new attributes including all configuration possibilities.

The Extended <gxl> Element

The body of a GXL document is enclosed in <gxl> tags. All further elements are embedded in these <gxl> tags. Some elements of a GXL document contain *IDs*. In our SWI-PROLOG implementation of the GXL library, all IDs have to be *atomic*. It will cause failures, if an ID is a *term* or an *unbound variable*.

Example 6.1 (Parts of a GXL document)

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<gxl ...>
   <graph ...>
      <node ...>...</node>
      ...
      <edge ...>...</edge>
      ...
   </graph>
</gxl>
```

The elements <node> and <edge> have further subelements. In particular, they can nest the element <graph> in their tags. This means, e.g., a GXL node can have further embedded graphs. The GXL specification of the element <gxl> contains more elements. The complete DTD of a GXL document can be downloaded and realized on the website of [64].

For our purposes, we add the element <prmtr> to the element <gxl> by redefining the parameter entity %gxl-extension;. The element <prmtr> contains the attributes label, width, height, mouse_click, and background. We specify the layout of the visualization window with these attributes.

DTD:

```
<ENTITY % graph-extension "prmtr?">
<!ELEMENT prmtr EMPTY>
<!ATTLIST prmtr
```

```
    label CDATA #implied
    width CDATA #implied
    height CDATA #implied
    mouse_click CDATA #implied
    background CDATA #implied>
```

`label`
: contains the headline of the visualization window/picture.

`width`
: is an integer, which determines the width of the visualization window/picture.

`height`
: is an integer, which determines the height of the visualization window/picture.

`mouse_click`
: is a reference to the attribute `alias` of the element `<mouse_clicks>` in a second XML document. This XML *configuration* document contains common parameters and defines, e.g., menus, pop-ups, and mouse clicks of a picture, edge symbol or node symbol. In this case, the XML configuration document defines the mouse click actions of the picture and the pop-up windows, which appear by a right mouse click on the picture. The value of this attribute can be composed of integers and letters. We describe the element `<mouse_clicks>` of the XML configuration document in Appendix B.2.

  The GXL document and the XML configuration document are passed to the visualization method and are appended to the visualizing window/picture. We can retrieve these documents any time from the picture (cf. Section 6.4.2, page 200).

`background`
: determines the background color of the visualization window. `Color` can be either an alias name or a RGB value in hexadecimal format. RGB values have the format `colour(HEX)`, e.g., `colour('#FF0000')` for red.

  We can get a list of proper color alias' using the demo tool of SWI-PROLOG. To start this tool, we call `manpce` in the PROLOG command line, then we choose the menu `File` and then `Demo programs`. At last, we choose the item `Colours` and we click the button `Open`.

The <graph> Element A GXL document can contain several `<graph>` elements. Nodes and edges of a given graph are defined by the `<node>` and `<edge>` elements (listed in any order), which can be addressed by their attribute `id`. In the specification of GXL, it is not allowed to make references between node IDs of different graphs, even if they are contained in the same GXL document. We do not change the original GXL DTD of the element `<graph>`.

The Extended <node> Element

Analogously to the element <gxl>, we add the element <prmtr> to the embedded elements of the element <node> by redefining the parameter entity %node-extension;. Some of the attributes of the element <prmtr> are required in order to visualize the graph. Others are optional. The value of the attribute id of the element <node> must be unique. In PROLOG, the value of the attribute id has to be atomic.

Example 6.2 (GXL node)

```
<node id="node_1">
   <prmtr mouse_click="node"
         handles="node" color="grey" size="medium"
         symbol="circle" x_pos="60" y_pos="40">
      <string bubble="Bubble" font_style="(times, roman, 12)">
         This is the label of the node
      </string>
   </prmtr>
</node>
```

The DTD of the element <prmtr> of a node looks as follows:

DTD:

```
<ENTITY % node-extension "prmtr">
<!ELEMENT prmtr string chart?>
<!ATTLIST prmtr
   symbol (circle|box|rhombus|
           honeycomb|triangle|
           no_symbol|text_in_box|
           text_in_ellipse|chart|CDATA) #required
   size (small|medium|large|CDATA) #required
   color CDATA #required
   mouse_click CDATA #implied
   handles CDATA #implied
   x_pos CDATA #implied
   y_pos CDATA #implied>

<!ELEMENT string #CDATA>
<!ATTLIST string
   bubble CDATA #implied
   font_style CDATA #implied>

<!ELEMENT chart bar+>
<!ATTLIST chart
   type (v_bar_graph|h_bar_graph) #required>

<!ELEMENT bar EMPTY>
<!ATTLIST bar
   height CDATA #required
```

```
color CDATA #required
position CDATA #required>
```

The Element <prmtr> contains the following attributes. These attributes are required:

`symbol`
: can be either a filename of a picture (gif or jpg) with absolute path, or it can be one of the following symbol names: `chart`, `circle`, `rhombus`, `honeycomb`, `triangle`, `no_symbol`, `text_in_box`, `text_in_ellipse`, and `box`. If we use `chart`, a configurable chart diagram will be shown as symbol of the node. We can specify this chart diagram in the subelement `<chart>` of the element `<prmtr>`.

`size`
: If the attribute `symbol` is a filename of a picture, then the value of the attribute `size` can be either a decimal number or a pair of numbers. If it is a decimal number between 0 and 100, then the size of the symbol will be scaled in percent referring to this number.

If it is a pair of numbers, e.g., `size="(34, 45)"`, it determines the x and y length of the picture in pixels. The symbol will be scaled to this absolute pixel size. If one value of the pair is unbound, then the symbol will be resized proportionally to the bound value.

If the specific symbol names mentioned above are used, then this attribute can be either an integer or one of the following aliases: `small`, `medium`, `large`. The numerical values of these aliases are defined in the XML configuration document (cf. Appendix B.2), which is – beyond the GXL document – passed to the visualization method, too. We can add further aliases for different sizes to this XML configuration document, which we can use, in the GXL document.

`color`
: determines the color of the symbol, if the specific symbol names are used. The value of the color is analogous to the background color (cf. attribute `background`, page 147). If we use a picture for the symbol, this attribute is ignored.

The following attributes of the element `<prmtr>` are optional:

`mouse_click`
: This attribute is – analogous to the element `mouse_click` of the element `<gxl>` (cf. page 147) – a reference to an XML configuration document.

`handles`
: is a reference to the attribute `alias` of the element `<handles>` of an XML configuration document (the same XML configuration document as mentioned by the attribute `mouse_click`). The value can be composed of integers and letters. The

XML configuration document defines the positions of the anchor points of a node. The edges are mounted on these anchor points. We describe the corresponding element `<handles>` of the XML configuration document in Appendix B.2.

`x_pos`, `y_pos`
: is the X and the Y position of the node. Each value is an integer. Negative values are allowed. The point (0, 0) is located on the upper left corner of the visualization window. If one of the attributes `x_pos` or `y_pos` is not set, then this coordinate of the node will be set to a randomly value.

The Element <string> contains the attributes `bubble` and `font_style`. The text of the attribute `bubble` appears in a bubble for some seconds, when the mouse pointer points on a node and is not moved.

The attribute `font_style` defines the font style. It is a triple with the following format: `(Font, Format, Size)`.

`Font`
: can be `courier`, `helvetica`, `screen`, `times`.

`Format`
: can be `bold`, `italic`, `roman`, `oblique`.

`Size`
: is an integer. Not all combinations between `Size` and `Font` are possible. Common values for `Size` are `10`, `12`, `14`, `18`, `24`.

Example 6.3 (The Element <string>)

```
<string bubble="Example 1" font_style=(screen, roman, 10)>
   Example of defining a font style
</string>
<string bubble="Example 2" font_style=(times, italic, 14)>
   Example of defining a font style
</string>
```

The text embedded in the element `<string>` contains the label of the node. Each new line will be written in a new line under the symbol of the node.

The Element <chart> Visualizing a node, the symbol of a node can be a small chart, too. The element `<chart>` defines a chart diagram as symbol of a node (cf. Figure 6.1). The element `<chart>` has the attribute `type` and the optional elements `<bar>`.

`type`
: The attribute `type` specifies the position of the bars in the chart. It can be a vertical or a horizontal bar chart. The value of the attribute `type` is `v_bar_graph` for vertical bars and `h_bar_graph` for horizontal bars.

A chart element can have several subelements `<bar>`.

The Element <bar> Each bar in the chart is defined through the subelement `<bar>`. The appearance of a bar is determined through the attributes `height`, `color`, and `position`.

`height`
: is an integer, which determines the height (vertical bar) or length (horizontal bar) of the bar in pixel.

`color`
: Each bar can have a certain color. The value of the color is analogous to the background color (cf. attribute `background`, page 147).

`position`
: is an integer, which defines the position of the bar relative to the first bar. Thereby, the position of the first bar is `0`. If we want to place a further bar beside the first bar, the position number is `1`. We can place a further bar on the first bar, too. The second bar overlaps the first bar. If the length of the first bar is larger than the second bar, we see the rest of the first bar, too. In Figure 6.1, we see that the blue bar overlaps the white bar.

Example 6.4 (Chart as Node-symbol)

```
<chart type="v_bar_graph">
  <bar height="67" color="white" position="0"/>
  <bar height="44" color="blue" position="0"/>
  <bar height="44" color="red" position="1"/>
  <bar height="23" color="green" position="2"/>
</chart>
```

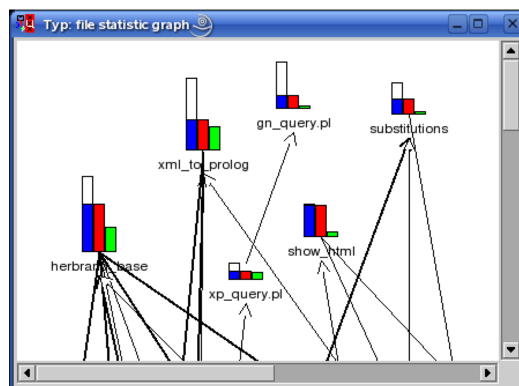


Figure 6.1: A Graph with Integrated Bar Charts as Node Symbol

The Extended <edge> Element

We add the element `<prmtr>` to the specification of a GXL edge by redefining the parameter entity `%edge-extension;`. There exists the same restrictions for the attribute `id` of an edge as for nodes. The value of the attribute `id` of the element `<edge>` must be unique and the `id` has to be atomic. The example below shows a GXL edge with the extension `<prmtr>`.

Example 6.5 (GXL Edge)

```
<edge id="edge_1" from="node_1" to="node_2">
   <prmtr mouse_click="edge"
         arrows="both" color="black" pen="2" weight="5"
         first_arrow="arrow_1" second_arrow="arrow_2">
      <calls/>
      <string bubble="Edge">
         Label of the edge
      </string>
   </prmtr>
</edge>
```

The DTD of the element `<prmtr>` of an edge looks as follows:

DTD:

```
<ENTITY % edge-extension "prmtr">
<!ELEMENT prmtr string calls?>
<!ATTLIST prmtr
   arrows (none|first|second|both) #required
   color CDATA #required
   pen CDATA #required
   first_arrow CDATA #implied
   second_arrow CDATA #implied
   mouse_click CDATA #implied
   weight CDATA #implied>

<!ELEMENT string #CDATA>
<!ATTLIST string
   bubble CDATA #implied
   font_style CDATA #implied>

<!ELEMENT calls call*>

<!ELEMENT call* #CDATA>
<!ATTLIST call
   from CDATA #implied
   to CDATA #implied>
```

The Element <prmtr> contains the following attributes. These attributes are required:

`arrows`
: determines, if the arrows of the edge are shown. The following values are allowed: `none`, `first`, `second`, and `both`. `none` shows no arrows, `first` shows an arrow at the beginning of the edge, `second` shows an arrow at the end of the edge and `both` shows an arrow at the beginning and at the end of the edge.

`color`
: determines the color of the edge. The value of the color is analogous to the background color (cf. attribute `background`, page 147).

`pen`
: is an integer, which determines the thickness of the edge line.

The following attributes of the element `<prmtr>` are optional:

`mouse_click`
: This attribute is – analogous to the element `mouse_click` of the element `<gxl>` (cf. page 147) – a reference to an XML configuration document.

`first_arrow`, `second_arrow`
: These attributes refer to the attribute `alias` of the element `<arrow>` of an XML configuration document (the same XML configuration document as mentioned by the attribute `mouse_click`). The value can be composed of letters and integers.

  The XML configuration document defines the appearance of an arrow. The attribute `first_arrow` defines the appearance of the arrow at the beginning of an edge, and the attribute `second_arrow` defines the appearance of the arrow at the end of an edge.

  It is not necessary to specify the arrows of an edge. If the attribute `first_arrow` or the attribute `second_arrow` is missing in the element `<prmtr>` and the attribute `arrows` of the element `<prmtr>` has the value `first`, `second` or `both`, then the standard arrows will be used. We describe the corresponding element `<arrows>` in detail in Appendix B.2.

`weight`
: We can give each edge a certain weighting. Some methods use the value of the attribute `weight` as a criteria in order to decide whether to, e.g., fade out an edge or to mark the edge with a certain color (cf. Section 6.3.2 and Section 6.3.3).

The Element <string> of the element `<edge>` is analogous to the element `<string>` of the element `<node>` (cf. Section 6.1.1, page 150).

The Element <calls> is only used in some call graphs. This element has no attributes. It has the optional subelement `<call>`. Each subelement `<call>` contains information about which calls from the source node to the target node the edge is representing. E.g., in a *module dependency graph*, the subelement `<call>` contains information about which predicates of a file in the module, which is represented through the starting node of the edge, are calling which predicates of another file of another module, which is represented though the target node of the edge.

The Element <call> contains information about which predicates from a file call which predicates of another file. The element `<call>` contains the following attributes:

`from`
: contains information about the source file, and

`to`
: contains information about the target file.

In the content of the element `<call>`, the called predicates are listed.

Example 6.6 (The Element <calls>)

```
<calls>
  <call from="file_handling.pl" to="input_output.pl">
    (user:concat_file_and_name)/3-(user:read_file_to_string)/2
    (user:concat_files)/3-(user:read_file_to_string)/2
    ...
  </call>
  ...
</calls>
```

The Default GXL Configuration

We store a default GXL document containing our GXL extensions in a file associated to the alias `gxl_presettings` (cf. Appendix A.5). The associated file can be changed by the user at any time. The settings of this GXL document are used to complete an incomplete GXL document, or to retrieve a GXL document from an arbitrary picture. If GXL elements or attributes in a GXL document are missing, then we query necessary GXL elements from the default GXL document and add them to the new GXL document. We explain the corresponding methods, which use the default settings in Section 6.2.3, 6.3.1 and 6.4.2. In Appendix D.3, we present a copy of the default GXL document.

To retrieve the location of the default file, we call the method

```
alias_to_default_file(gxl_presettings, -File).
```

Starting the SCAV, the corresponding file is consulted and the PROLOG fact

```
gxl_presettings(-GXL)
```

is asserted. This fact contains the default GXL document represented in field notation. The fact is queried by all corresponding methods, which complete an incomplete GXL document. If the file containing the default GXL settings is changed, then we reload the new content into this fact by calling

```
load_gxl_presettings.
```

We can also choose another default GXL document. We load the content of another file into the fact `gxl_presettings/1` calling the method

```
load_gxl_presettings(+File).
```

Again, the content of the file is stored in the fact `gxl_presettings/1`. The previous content is deleted.

In order to get an element of the default GXL settings, we call

```
gxl_presetting(+Type, -Gxl).
gxl_presetting(+Type, ?ID, -Gxl).
```

`gxl_presetting/2` returns a GXL element of the type `Type`. The following values of `Type` are possible:

`gxl`
: returns the complete default GXL document.

`head`
: returns the part of the GXL document, in which the element `<prmtr>` and the element `<graph>` without node or edges is embedded.

`head_pure`
: returns only the header of the GXL document and the element `<prmtr>`.

`graph`
: returns the element `<graph>` containing a node and an edge element.

`node`
: returns the element `<node>` and

`edge`
: returns the element `<edge>` of the default GXL document.

`gxl_presetting/3` does the same, but we can set or receive the attribute `id` of the elements `<graph>`, `<node>`, and `<edge>`. If the attribute `id` is not set, then it is generated using

```
gensym(id_, ID).
```

The attributes `x_pos` and `y_pos` of a node element are always set to random integers between 0 and 300.

Example 6.7 (Retrieving GXL Presettings)

```
?- gxl_presetting(head, id, Gxl).

Gxl =
   gxl:['xmlns:xlink':'http://www.w3.org/1999/xlink']:[
     prmtr:[label:'Default GXL', width:450, height:400,
            background:white, mouse_click:picture]:[],
     graph:[edgeids:true, edgemode:directed,
            hypergraph:false, id:id]:[]]

Yes
```

6.1.2 Layout Methods

In order to layout a graph, we have several possibilities: depending on the size of the graph, we can use internal layout algorithms, a built-in layout algorithm, which is offered by SWI-PROLOG, or external layout algorithms. The internal algorithms are especially useful for hierarchical graphs, e.g., trees. The external layout algorithms are for all other graphs and can be configured for certain layout purposes.

We have implemented the abstract method

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2).
```

which transforms a GXL document in depending on the XML configuration document `Actions`. We use this abstract method in Section 6.3 for further transformations. In order to layout a GXL document, we have to set the configuration document `Actions` as it is described in the following subsections. For each node, the layout algorithms add or change the x- and the y-position in the XML document `Gxl_1`. The resulting XML document is `Gxl_2`. This method is not only used for layouting. Other transformation are performed, too, depending on the XML configuration document `Actions`. In the following, we explain how to configure the XML document, in order to determine the layout algorithm, and we describe how to set possible parameters of the layout algorithm in the XML document.

The Internal GXL Graph Layouter

We have implemented some basic tree layout algorithms.

We call

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2)
```

whereby `Actions` is an XML document.

Example 6.8 (The Element <actions>)

```
<actions>
   <action name="layout">
      <gxl_layout mode="bfs" x_start="10" y_start="10"
         x_step="50" y_step="50" y_variance="10">
         <roots>
            <node id="id_1"/>
            ...
         </roots>
      </gxl_layout>
   </action>
</actions>
```

The corresponding DTD of the configuration looks as follows:

DTD:

```
<!ELEMENT action gxl_layout>
<!ATTLIST action
   name layout #required>

<!ELEMENT gxl_layout roots?>
<!ATTLIST gxl_layout
   mode (bfs|bfs_inv|dfs|dfs_inv|xpce|
      dot|twopi|neato|circo|fdp) #required
   x_start CDATA #required
   y_start CDATA #required
   x_step CDATA #required
   y_step CDATA #required
   y_variance CDATA #required>

<!ELEMENT roots node+>

<!ELEMENT node EMPTY>
<!ATTLIST node
   id CDATA #required>
```

The Element <gxl_layout> We can set the following attributes of this element:

`mode`
: selects between different layout algorithms. In order to execute one of the self implemented layout algorithms, `mode` can be `bfs`, `bfs_inv`, `dfs`, or `dfs_inv` (cf. Figure 6.2, 6.3). `bfs`, `bfs_inv` creates a tree layout using a *breadth first search* algorithm, and `dfs`, `dfs_inv` creates a tree layout using a *depth first search* algorithm. The additional suffix `_inv` means that the `x` and `y` values are interchanged. This effects that ones the graph runs from top to bottom, and others, the graph runs from left to right. The four algorithms use the subelements `<node>` of the element `<roots>`, if available, as starting nodes. Otherwise, the root nodes will be determined by the algorithms themselves. These layout algorithms are implemented in SWI-PROLOG.

We can use external layout algorithms, too. If the *Graph Visualization Software* `graphviz` [11] is installed, further possible values of `mode` are `dot`, `twopi`, `neato`, `circo`, and `fdp` (only for Unix based systems). Using the external layouter graphviz, the following steps are processed: first the GXL document will be transformed into the *digraph/dot* format, which `graphviz` accepts as input graph format. The digraph will be saved in a temporary file. After this, the layouter is called with the temporary file as input argument. The layouter calculates the layout and saves the new layout information in a second temporary file. The layout information has the *svg* format. We read this file, get the layout information and write them into the original GXL document, which we display in the picture. At the end of this section and in Section 6.1.2, we describe further layout methods.

`x_start`
: sets the x-position of the first node (only used by the self implemented layout algorithms).

`y_start`
: sets the y-position of the first node (only used by the self implemented layout algorithms).

`x_step`
: sets the x-distance to the next node (only used by the self implemented layout algorithms).

`y_step`
: sets the y-distance to the next node (only used by the self implemented layout algorithms).

`y_variance`
: The y-distances between the nodes are split into 6 intervals between the borders `y_step - 3 * y_variance` and `y_step + 3 * y_variance`. This effects that the text of nodes does not overlap each other, because it is staggered (only used by the self implemented layout algorithms).

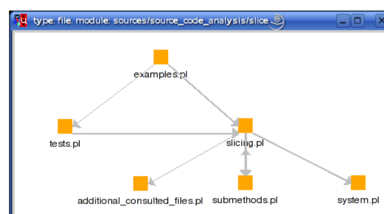


Figure 6.2: File Dependency Graph of the Module Slice: BFS Layout

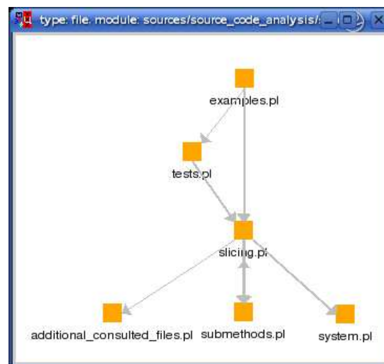


Figure 6.3: File Dependency Graph of the Module Slice: DFS Layout

The Element <roots> This element contains multiple node IDs, which will be used as initial nodes. If this element is missing, then we determine the roots automatically. The element `<roots>` is only used by the internal layout algorithms `bfs`, `bfs_inv`, `dfs`, or `dfs_inv`.

The SWI-PROLOG Layout Algorithm

After a graph is displayed in a picture, we can apply a further layout algorithm to the nodes of the picture, which does not use the underlying GXL document:

```
xpce_graph_layout(xpce, +Picture).
```

`xpce` is a layout algorithm implemented by Jan Wielemaker (cf. Figure 6.4). This layout algorithm is written in SWI-PROLOG. It only works on the node addresses of a picture. Therefore, the new coordinates of the nodes are not yet saved in the GXL document, but we have implemented the method `picture_to_gxl/3` to transfer the nodes and edges of a picture to a GXL document, including the node positions in the picture (cf. Section 6.4.2, page 200).

The call `xpce_graph_layout/2` can be used together with the other `mode` arguments described above. Using another argument than `xpce`, the call

```
xpce_graph_layout(+Mode, +Picture)
```

first writes the nodes and edges of a picture into a GXL document. Then it layouts the graph. After this, it clears the picture and at last, it visualizes the graph with the new layout in the picture. Summarized, `Mode` can be `bfs`, `bfs_inv`, `dfs`, `dfs_inv`, `xpce`, `dot`, `twopi`, `neato`, `circo`, and `fdp`.

The External Visualization Tools or Layouter

In order to use an external visualization tool, e.g `GXLGraphpad` [33], we can separate the pure GXL elements and the additional elements of our GXL document in an extra file.

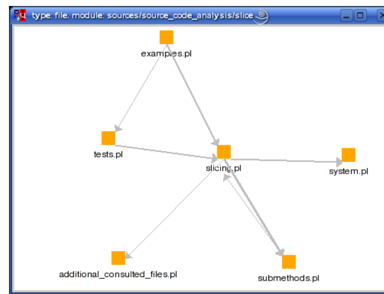


Figure 6.4: File Dependency Graph of the Module Slice: XPCE Layout

We receive two documents, one with pure GXL (GXL document) and a second with all elements of our GXL extension (layout document). The corresponding elements of each document are referenced by their IDs. This allows us to merge the two documents again later.

Separating Layout and GXL In order to separate the layout information from a GXL document, we call one of the following methods:

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2).
gxl_to_gxl(+Actions, +Gxl_1, -Layout, -Gxl_2).
```

In this case, the configuration `Actions` only contains the element `<action>` with the attribute `name="separate_gxl_and_layout"`.

Example 6.9 (The Element <actions>)

```
<actions>
   <action name="separate_gxl_and_layout"/>
</actions>
```

`Gxl_1`
: is a GXL document including our GXL extensions.

`Layout`
: is the resulting XML document containing only our GXL extensions of the document `Gxl_1`. In other words, this XML document contains only the `<prmtr>`-elements.

The DTD of this XML layout document looks as follows:

DTD:

```
<!ELEMENT gxl_layout gxl? layout*>

<!ELEMENT gxl prmtr>
<!ATTLIST gxl
   id #required>

<!ELEMENT layout prmtr>
<!ATTLIST gxl
   id #required>
```

Each element `<prmtr>` is a copy of the corresponding element `<prmtr>` of the GXL document. Each element pair contains the same IDs.

`Gxl_2`
: is a pure GXL document without any foreign elements, which can be used in external tools.

We can use this pure GXL document in an external visualization tool. If this tool is able to calculate a layout for the nodes and save this layout in a file in XML format, we can extract these layout information and add them to our layout document or to our original GXL document. The node positions of the original document will be replaced.

Merging Layout and GXL We can merge the layout document and the pure GXL document again:

```
gxl_to_gxl(+Actions, +Gxl_1, +Layout, -Gxl_2).
```

In this case, the configuration `Actions` contains the element `<action>` with the attribute `name="merge_gxl_and_layout"`.

Example 6.10 (The Element <actions>)

```
<actions>
   <action name="merge_gxl_and_layout"/>
</actions>
```

`Gxl_1`
: is a GXL document with or without our GXL extensions. We recommend to use the GXL document with our GXL extension, if we want to merge the layout information of an external layout tool. If we want to merge a layout document, which we generated ourself using `gxl_to_gxl/4`, we recommend to use the pure GXL document. Doing so, we avoid missing or double entries in the resulting GXL document.

`Layout`
is an XML document containing the node positions. We can process the layout information, which are generated from our separation method, or which are generated by `GXLGraphpad`, or which are in the *svg* format.

`Gxl_2`
is the merged GXL document.

6.2 Basic Methods for GXL Documents

We have implemented multiple methods, which operate on GXL documents. In this section, we describe the basic methods. These methods do not return a GXL document. The input of the method is a GXL document, and the results are either booleans or lists, containing the IDs of nodes and edges. In Section 6.3, we describe methods, which return modified GXL documents. These methods build on the methods of this section.

Extended Reading and Writing for GXL Documents

We have extended the methods of the field notation, which read or write an XML document into or from a file. In the following sections, we explain how to use the read and write methods together with GXL documents.

Reading a GXL Document from a File

In order to load a GXL document from a file and to represent the content in field notation, we use

```
dread(xml(+Attribute_Types), +File, -Gxl).
```

We specify the *attribute type* of an XML document using the argument

```
Attribute_Types = dtd:[a_1:V_1, a_2:V_2, ...]:[],
```

or

```
Attribute_Types = Alias.
```

Thereby, the argument `Attribute_Types` can either be in field notation, containing the specification itself, or it can be an alias of a predefined attribute type specification. For each attribute of an XML document, we specify the corresponding type in the following format (cf. Example 6.11). The attributes consists of a list of attribute name and type pairs.

```
Attribute_Name:Type.
```

Each pair of this list defines the type of an attribute of the read XML document.

Example 6.11 (Attribute Types)

```
<dtd size="int" prmtrs="list" font_style="sequence"/>
```

The default type of each attribute is `atom`. This means, if an *attribute type* of an attribute is not specified, it will be transformed to an `atom`. Thereby, *terms*, *lists*, *sequences* and *integers* will be set into *apostrophes*. We specify the following types:

`int`
: used in order to specify that the corresponding attribute value is an integer.

`term`
: used in order to specify that the corresponding attribute value is a term.

`list`
: used in order to specify that the corresponding attribute value is a list in squared brackets.

`sequence`
: used in order to specify that the corresponding attribute value is a sequence in parenthesis.

Example 6.12 (Attribute Types) We read the file `'graph.gxl'`, containing a GXL graph and parse the attributes into the corresponding types given in DTD:

```
?- File = 'graph.gxl',
   DTD = dtd:[size:int, color:term, font_style:sequence]:[],
   dread(xml(DTD), File, Gxl).

Gxl = gxl:[...]:[
   graph:[]:[
      node:[id:...]:[
         prmtr:[color:colour(35, 84, 84), size:24, ...]:[
            string:[font_style:(times, roman, 18)]:[...] ] ],
   ...

Yes
```

Otherwise, `color`, `size`, and `font_style` would be atoms.

```
?- File = 'graph.gxl',
   dread(xml, File, Gxl).

Gxl = gxl:[...]:[
   graph:[]:[
      node:[id:...]:[
         prmtr:[color:'colour(35, 84, 84)', size:'24', ...]:[
            string:[font_style:'(times, roman, 18)']:[...] ] ],
   ...

Yes
```

Instead of the XML document `Attribute_Types` we can use an alias of a predefined attribute specification, too. The corresponding attribute specification of each alias is defined in the file

```
source_code_analysis/common/alias_to_dtd.pl
```

and can be extended. The following aliases for attribute specifications already exist:

```
gxl_graph, gxl_presettings, gxl_config,
visur_config, visur_browser, config_arrow_types,
hierarchy_browser_config, alias_browser, file_history.
```

In order to load a GXL document, we usually use the alias `gxl_graph`.

Example 6.13 (Reading an XML Document) Reading an XML document, using the predefined attribute type specification `gxl_graph`.

```
?- dread(xml(gxl_graph), 'gxl_graph.gxl', Gxl).

Gxl = ...

Yes
```

Remark: A GXL document can be loaded into a PROLOG variable without using an attribute type specification:

```
dread(xml, +File, -Gxl).
```

But then, it is necessary that we convert the type of some attributes from `atom` to `int`, `term`, or `sequence`, ourself. We also have to pay attention to the fact that `dread/3` used without the argument `DTD_Types`, encloses the resulting term in squared brackets, e.g.,

```
Gxl = [xml:[...]:[...]]
```

instead of

```
Gxl = xml:[...]:[...].
```

Remark: We can split large XML documents into an XML *main document* and several XML *sub-documents*. We refer to the XML sub-documents using the attribute `ref`. `ref` contains the name of the file, containing the XML sub-document. In the XML main document, the XML element with the attribute `ref` will be completely replaced by the XML sub-document.

This is useful, if we use several, different XML documents for configuring diverse tools, but which have some common elements. If we want to share a certain part of the configuration in all XML documents, we spare to replicate the same parts of the XML

documents again and again. Even if we want to use the same part of an XML document several times in one XML document, it is useful just to reference the repeating part. Thereby, the referenced filename is either an absolute path or a path relative to the path of the XML main document.

`dread` only includes further referenced documents, if it is used together with the argument `Attribute_Types`.

Example 6.14 (XML Documents with References) We have two XML documents. The XML document `XML_Main` contains all main elements which are not shared with other XML documents. The XML document `XML_Sub` contains elements, which are shared from several other XML documents or which can be found several times in an XML document. Using the method `dread/3` together with the argument `DTD_Types` embeds the XML document `XML_Sub` in the XML documents `XML_Main` at the corresponding position.

XML document `XML_Main`

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<config>
  <gxl_layout .../>
  <mouse_clicks ref="mouse_clicks.xml"/>
  ...
</config>
```

XML document `XML_Sub`

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<mouse_clicks>
  <mouse_click alias=...>
  ...
  </mouse_click>
  ...
</mouse_clicks>
```

Resulting XML document:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<config>
  <gxl_layout .../>
  <mouse_clicks>
    <mouse_click alias=...>
    ...
    </mouse_click>
    ...
  </mouse_clicks>
  ...
</config>
```

Writing a GXL Document into a File

In order to save a GXL document in a file, we use

```
dwrite(xml_2, +File, +Gxl).
```

Instead of the argument `xml_2`, other values are possible, too, which are specialized to save other kinds of documents. But only `dwrite/3` used with the argument `xml_2` saves a GXL document in a proper way.

6.2.1 Correctness of GXL Documents

Before using an imported GXL document, it is recommendable to check the correctness of the GXL document. It is also recommendable to check a GXL document before exporting it. Therefore, we have implemented several methods, which check a GXL document for obvious faults, which could cause problems during processing or visualizing a GXL document.

Verifying IDs

We use the node- and edge-IDs of a GXL document in order to generate a corresponding XPCE address of a node- or edge object of a picture. These XPCE object-addresses have to be atomic. For this reason, the IDs of a GXL document in PROLOG have to be atomic, too. Otherwise, there will arise problems sending a GXL document to a picture. The method

```
check_atomic_ids_in_gxl(+Gxl)
```

checks, if the IDs of a GXL document are atomic or not. This method searches for the attributes `id`, `from`, and `to` of the elements `<node>`, `<edge>`, and `<graph>` and checks, if the corresponding values are atomic.

Locating Inconsistencies of Nodes or Edges

We have implemented methods to retrieve extraordinary nodes or edges. Extraordinary nodes are nodes, which are referenced by an edge in the graph, but which are not contained in the graph. Extraordinary edges are the edges, which contain references to not existing nodes in the graph. Finally, we search for all single nodes, which are not referenced by any edge, too.

Locating Missing Nodes

It is possible that there exists edges, which are connected to nodes, which are not included in the GXL document. Such a invalid GXL document may be imported from another graph tool. Our algorithms can treat such GXL documents. We just write a warning to the console. But if we want to export such an invalid GXL document, e.g., in order to use it together with another tool, we have to delete these edges. The method call

```
gxl_to_missing_nodes(+Gxl, -Missing_Nodes)
```

detects all node IDs, which are referenced by an edge, but which are not existing in the GXL document. In Section 6.3.2, we describe a method, which generates a new GXL document without edges that reference a missing node.

Locating Edges with Missing References In order to retrieve the edges, which reference not existing nodes in a GXL document, we have implemented the method

```
gxl_to_edges_with_empty_reference(+Gxl, -Edges).
```

In Section 6.3.2, we describe a method, which deletes these edges from the GXL document and generates a new GXL document without these edges.

Locating Single Nodes Nodes in a GXL document, which are not referenced from any edge are single nodes. In order to find the IDs of the single nodes, we call

```
gxl_to_single_nodes(+Gxl, -Single_Node_IDs).
```

`Single_Node_IDs` contains a list of the single node IDs of the GXL document `Gxl`. In Section 6.3.2, we describe a method call, which generates a new GXL document without single nodes.

6.2.2 Classifying Nodes and Edges

The methods of this section classify the nodes and edges of a GXL document into diverse types.

Classifying Nodes

We classify the nodes of a GXL document into five different types. Some of our methods described later are based on this classification. The method

```
gxl_to_node_types(+Gxl,
      -Fathers, -Embedded, -Not_Embedded,
      -Deduced_Nodes, -D_Node_Fathers).
```

generates five sets, each set representing a certain class of node types. The sets contain the corresponding node IDs of the GXL document.

`Gxl`
: is the GXL document, which nodes we want to classify.

`Fathers`
: Nodes can have *embedded graphs*, containing further nodes, edges and graphs. In this set, all node IDs are listed, which are not contained in an embedded graph of a node but which contain themselves an embedded graph (in other words, these nodes are in the XML structure on the top level and have an embedded graph).

`Embedded`
contains all node IDs, which are contained in an embedded graph of a node.

`Not_Embedded`
contains the nodes, which have no embedded graphs and which are not embedded themselves in an embedded graph (only the nodes, which are in the XML structure on the top level are examined, if they have not an embedded graph).

The nodes of the sets `Fathers`, `Embedded` and `Not_Embedded` are disjunct. The union of these sets are all nodes of the GXL document. The union of the sets `Fathers` and `Not_Embedded` are all nodes, which are shown in a GXL document, if we call the method `gxl_to_picture/3`. The embedded nodes are only shown in a picture, if we call a corresponding method. Such a method is described in Section 6.3.4.

`Deduced_Nodes`
are all nodes, which are referred from an edge. They can be the source of an edge (edge attribute `from`), or the target of an edge (edge attribute `to`). It may be the case that the corresponding node does not exist in the GXL document. The set `Missing_Nodes` mentioned in Section 6.2.1 is a subset of this set.

`D_Node_Fathers`
are the father nodes of the deduced nodes `Deduced_Nodes`, which are in an embedded graph of a node. These set is always equal to, or it is a subset of the set `Fathers`.

Example 6.15 (Node Classification)

```
?- Gxl =
   gxl:[...]:[
     graph:[...]:[
       node:[id:n1]:[
         graph:[...]:[
           node:[id:n2]:[...], node:[id:n3]:[...],
           edge:[from:n2, id:e2, to:n3]:[...]]],
       node:[id:n4]:[...], node:[id:n5]:[...],
       edge:[from:n1, id:e1, to:n6]:[...]]],

   gxl_to_node_types(Gxl,
      Fathers, Embedded, Not_Embedded,
      Deduced_Nodes, D_Node_Fathers).

   Fathers = [n1],
   Embedded = [n2, n3],
   Not_Embedded = [n4, n5],
   Deduced_Nodes = [n1, n2, n3, n4, n6],
   D_Node_Fathers = [n1]

Yes
```

Classifying Tree-, Forward-, Back- and Cross-Edges

We can divide edges into *tree*, *forward*, *back* and *cross edges*. The following description defines these kinds of edges [32]:

- a tree edge is an edge in a DFS tree,
- a back edge connects a vertex to an ancestor in a DFS tree; a self-loop is a back edge,
- a forward edge is a non-tree edge that connects a vertex to a descendant in a DFS tree,
- a cross edge is any other edge in a graph; it connects vertices in two different DFS trees or two vertices in the same DFS tree neither of which is the ancestor of the other.

The method

```
gxl_to_edge_types(+Mode, +Gxl,
   -Tree, -Forward, -Back, -Cross)
```

determines the tree-, forward-, back- and cross-edges of a graph in a GXL document. If `Mode` is `id` or `1`, each output list contains the edge IDs. If `Mode` is `v_w` or `2`, each output list contains the elements in the format `source_id-target_id`. The implemented algorithm only classifies the edges correct, iff there exists at most *one* edge between each pair of nodes.

Remark: The type of an edge depends on the *tree edges*. A set of tree edges is not always well-defined. Depending of the algorithm, which determines the tree edges there may exist different sets with tree edges.

In Section 6.3.3, we describe a method, which generates a new GXL document with highlighted tree-, forward-, back-, and cross-edges.

6.2.3 Conversion between Ugraphs and GXL Documents

We are able to visualize graphs in diverse formats, e.g., *Ugraphs*, or graphs given as a list of *vertices* and *edges*. We have implemented diverse methods for converting between different graph formats and to a GXL.

Nodes and Edges to GXL If we just want to display some nodes and edges, we first have to transform the nodes and edges into corresponding GXL elements and construct a proper GXL document. The call

```
vertices_and_edges_to_gxl(+Vertices, +Edges, -Gxl)
```

transforms a list of nodes and edges into a GXL document. `Vertices` is a list of nodes, `Edges` is a list of edges, and `Gxl` is the resulting GXL document. An element of the list `Vertices` can be an incomplete `<node>` element, or it can have one of the following structures:

- (1): `Node_ID`,
- (2): `Node_ID-Color`,
- (3): `Node_ID-Label-(X, Y)`,
- (4): `Node_ID-Label-Symbol-Color`,
- (5): `Node_ID-Label-Symbol-Color-Size`,
- (6): `Node_ID-Label-Symbol-Color-Size-(X, Y)`.

For each node, these parameters can be merged to a simplified XML node element and used in the list `Vertices`, too:

```
<node id=Node_ID color=Color label=Label
   x=X y=Y symbol=Symbol size=Size/>
```

An element of the list `Edges` can be an incomplete `<edge>` element or it can have one of the following structures:

- (2): `V_ID-W_ID`,
- (3): `V_ID-W_ID-Color`,
- (5): `Edge_ID-V_ID-W_ID-Color-Arrows-Pen`,
- (7): `Edge_ID-V_ID-W_ID-Color-Arrows-Pen-Second_Arrow`.

For each edge, these parameters can be merged to a simplified XML edge element and used in the list `Edges`, too:

```
<edge from=V_ID to=W_ID color=Color id=Edge_ID
   arrows=Arrows pen=Pen second_arrow=Second_Arrow/>
```

The variables have the following meaning:

`Node_ID`
: is the ID of a node.

`Color`
: is the color of a node or an edge. The value of the color is analogous to the background color (cf. attribute `background`, page 147).

`Label`
: is a list of atoms, defining the label of a node or edge.

`(X, Y)`
: is the position of a node.

`Symbol`
: is the symbol of a node.

`Size`
: is the size of a node.

`V_ID`
: is the ID of source node of an edge.

`W_ID`
: is the ID of target node of an edge.

`Arrows`
: determines the arrows of an edge. It can be `none`, `first`, `second` or `both` (cf. DTD of an edge, Section 6.1.1, page 152).

`Pen`
: is thickness of a line.

`Second_Arrow`
: is a reference to the attribute `alias` of the element `<arrow>` of an XML configuration document. There, we define the appearance of an arrow. The value can be composed of letters and integers.

Example 6.16 (Vertices and Edges to GXL)

```
?- Vertices = [n1-'node 1'-box-colour('#FF0000'), n2],
   Edges = [n1-n2],
   vertices_and_edges_to_gxl(Vertices, Edges, Gxl).

Gxl = gxl:[...]:[prmtr:[label:'Default GXL', ...]:[],
         graph:[id:id_1 ...]:[
            node:[id:n1]:[
               prmtr:[color:colour('#FF0000'), ...]:[...] ],
            node:[id:n2]:[...],
            edge:[from:n1, id:id_2, to:n2]:[
               prmtr:[arrows:both, ...]:[]]]]

Yes
```

Missing elements or attributes of a GXL node or edge will be added from the default GXL settings (cf. Section 6.1.1). Thereby, the method

```
gxl_element_to_element_complete(+E_1, -E_2)
```

is called to complete a node or an edge element.

GXL to Nodes and Edges In order to convert a GXL document into nodes and edges we call

```
gxl_to_vertices_and_edges(
   +Mode, +Gxl, -Vertices, -Edges).
```

The parameter `Mode` determines the format of the nodes and edges. Mode has the format `Mode = Vertex-Edge`.

`Vertex`
can have one of the following values:

- `-1`
  effects that the vertices are merged to the following simplified XML structure

  ```
  <node id=ID label=Label symbol=Symbol
     color=Color size=Size x=X y=Y/>.
  ```

  The advantage is, that we can extend the attributes or the content of this structure without changing the methods using this structure already. Using the following `Vertex` parameters `1`, `6`, we cannot change the structure without adapting corresponding methods. The advantage of the parameters `1`, `6` is, that the resulting structure is quickly written and easy to handle.
- `0`
  effects that the nodes will not be extracted from the GXL document. The list `Vertices` will be empty. This is useful, if we only want to extract the edges.
- `1`
  effects that the vertices only consist of the `ID`.
- `6`
  effects that the vertices have the structure
  `ID-Label-Symbol-Color-Size-(X, Y)`.

`Edge`
can have one of the following values:

- `-1`
  effects that the vertices are merged to the following simplified XML structure

  ```
  <edge id=ID from=V_ID to=W_ID color=Color
     arrows=Arrows pen=Pen weight=Weight/>.
  ```

Analogous above, the advantage is, that we can extend the attributes or the content of this structure without changing the methods using this structure already.

- `0`
  effects that the edges will not be extracted from the GXL document. The list `Edges` will be empty. This is useful, if we only want to extract the nodes.
- `1`
  effects that the edges only consist of the `ID`.
- `2`
  effects that the edges have the structure
  `V_ID-W_ID`.
- `6`
  effects that the edges have the structure
  `ID-V_ID-W_ID-Color-Arrows-Pen`.
- `7`
  effects that the edges have the structure
  `ID-V_ID-W_ID-Color-Arrows-Pen-Weight`.

The variable `ID` is the ID of an edge. The other variables are already explained above.

Ugraph to GXL PROLOG contains the graph library *ugraphs*. This library supports several graph operation, e.g., determining the strong components of a graph. We can convert a Ugraph into a GXL document calling the method

```
ugraph_to_gxl(+Ugraph, -Gxl).
```

Example 6.17 (Ugraph to GXL)

```
?- Ugraph = [a-[b, c], b-[c], c-[]],
ugraph_to_gxl(Ugraph, Gxl).

Gxl =
  gxl:['xmlns:xlink':'http://www.w3.org/1999/xlink']:[
    prmtr:[label:'Default GXL', width:450, height:400,
           background:white, mouse_click:picture]:[],
    graph:[edgeids:true, edgemode:directed,
           hypergraph:false, id:id1_8]:[
      node:[id:a]:[
        prmtr:[color:grey, handles:default,
               mouse_click:node, size:small,
               symbol:no_symbol, x_pos:282, y_pos:164]:[
          string:[bubble:a,
                  font_style:(times, roman, 12)]:[a]]],
      node:[id:b]:[
```

```
      prmtr:[color:grey, handles:default,
             mouse_click:node, size:small,
             symbol:no_symbol, x_pos:271, y_pos:89]:[
        string:[bubble:b,
                font_style:(times, roman, 12)]:[b]]],
    node:[id:c]:[
      prmtr:[color:grey, handles:default,
             mouse_click:node, size:small,
             symbol:no_symbol, x_pos:57, y_pos:144]:[
        string:[bubble:c,
                font_style:(times, roman, 12)]:[c]]],
    edge:[from:a, id:id1_12, to:b]:[
      prmtr:[arrows:both, color:black, pen:1,
             mouse_click:edge, first_arrow:first_arrow,
             second_arrow:second_arrow, weight:1]:[
        string:[font_style: (times, roman, 12),
                bubble:'Default Edge']:[]]],
    edge:[from:a, id:id1_13, to:c]:[
      prmtr:[arrows:both, color:black, pen:1,
             mouse_click:edge, first_arrow:first_arrow,
             second_arrow:second_arrow, weight:1]:[
        string:[font_style: (times, roman, 12),
                bubble:'Default Edge']:[]]],
    edge:[from:b, id:id1_14, to:c]:[
      prmtr:[arrows:both, color:black, pen:1,
             mouse_click:edge, first_arrow:first_arrow,
             second_arrow:second_arrow, weight:1]:[
        string:[font_style: (times, roman, 12),
                bubble:'Default Edge']:[]]]]]

Yes
```

GXL to Ugraph We convert a GXL document into a Ugraph, using the following method

```
gxl_to_ugraph(+Gxl, -Ugraph).
```

`Gxl`
is the GXL document written in field notation.

`Ugraph`
is the corresponding Ugraph.

6.3 Transformation Methods for GXL Documents

We support different transformations on GXL documents, e.g., methods to validate GXL documents, to layout graphs, to color nodes and edges, to fade out nodes and edges, to

determine the strongly connected components, and generally, to transform one GXL document into another GXL document. These transformation algorithms have as input argument a GXL document. After processing this GXL document, we return a new GXL document. The transformation can be easily determined in an XML configuration document, which is always passed as the first argument of the transformation method. Therefore, we extended the abstract method

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2),
```

which we already described in Section 6.1.2 in parts. This method operates on GXL documents and transforms a GXL document into a new GXL document. This method is configured by an XML document, or field notation respectively. Using an XML document makes it easy to configure the method individually. The arity of this method is, except once, always three. The XML configuration document is passed as the first argument to the PROLOG method.

Depending on the XML configuration document `Actions`, a certain transformation will be applied to the GXL document `Gxl_1`. Each transformation is specified by the element `<action>` of the XML document `Actions`.

Example 6.18 (The Element <action>)

```
<actions>
  <action name=Action/>
</actions>
```

The content of the element `<action>` can contain further attributes, or, respectively, this element can contain further subelements, which are necessary for the corresponding transformation. It is possible that the XML configuration document `Actions` contains several elements `<action>`, each determining a certain transformation; these transformations will be executed successively on the GXL document.

Example 6.19 (Iterative Transformations) The following XML configuration document determines three transformations, which will be processed iteratively.

```
<actions>
  <action name=Action_A/>
  <action name=Action_B/>
  <action name=Action_C/>
</actions>
```

The corresponding field notation looks as follows:

```
actions:[]:[
  action:[name:Action_A]:[],
  action:[name:Action_B]:[],
  action:[name:Action_C]:[] ].
```

The DTD of this XML document looks as follows:

DTD:

```
<!ELEMENT config action*>

<!ELEMENT action CDATA|EMPTY>
<!ATTLIST action
   name validate|
        change_ids|
        delete|
        delete_missing_references|
        delete_single_nodes|
        delete_prmtr_tags|
        layout|
        separate_gxl_and_layout|
        merge_gxl_and_layout|
        set_color|
        edge_weight|
        call_weight|
        delete_light_weighted_edges|
        delete_colored_edges|
        scc|
        highlight_scc|
        merge_nodes|
        delete_loops|
        delete_embedded_graphs|
        highlight_tfbc_edges #required>
```

Depending on the attribute `name` of the element `<action>`, the element `<action>` can have further subelements. We explain these subelements in the corresponding sections, where we describe the transformation of the GXL document, in detail. The result of the transformation of the GXL document `Gxl_1` is given in the GXL document `Gxl_2`.

6.3.1 Completing and Correcting GXL Documents

In this section, we describe how to configure the method `gxl_to_gxl/3` in order to generate a valid GXL document. Missing parts of our GXL extensions are added to the corresponding elements of GXL document.

Furthermore, we explain how to configure the method, in order to check if the used IDs in a GXL document have the correct type, so that the GXL document can be used by other tools. In both cases, a new GXL document is generated, which contains the necessary changes.

Creating a valid GXL Document

GXL documents can be retrieved from any source and used in PROLOG, e.g., they can be imported from another graph processing tool, which uses a GXL document as output format. After a GXL document has been imported from somewhere, we have the ability to validate the document. We have implemented a method in order to receive a valid GXL

document, which we can use together with our implemented PROLOG methods. For this purpose, we have to add our GXL extensions, which we describe in Section 6.1, to the original GXL document.

If we want to create a GXL document ourself using only a few attributes and elements, we can use this method or its sub-method in order to generate a complete GXL document, too.

Nevertheless, the rough structure of the original GXL document has to be valid. This means that the `<node>` and `<edge>` elements have to be embedded in the `<graph>` elements and the `<graph>` elements have to be embedded in either the `<node>`, `<edge>` or `<gxl>` elements. We only add missing attributes and our extensions (written in the element `<prmtr>`) to the incomplete elements of the original GXL document. The call

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2)
```

used with the configuration of `Actions`

```
<actions>
   <action name="validate"/>
<actions>
```

transforms an arbitrary, incomplete GXL document to a valid GXL document. Then, it can be used together with all of our GXL methods. This method checks, if the IDs are atomic, too (cf. Section 6.2.1). If an ID is not atomic, it transforms the ID to an atomic ID.

All added default values will be retrieved from the the default GXL settings (cf. Section 6.1.1). The method `gxl_to_gxl/3` calls recursively the method

```
gxl_element_to_element_complete(+E_1, -E_2),
```

which completes the following elements, if they are not complete, on the basis of the default GXL settings:

```
<gxl>, <graph>, <node>, <edge>.
```

`E_1` is in field notation, which begins with `node`, `edge`, `graph` or `gxl`. This method only completes the current XML element `E_1`. It will not check further, embedded elements in the content of the element `E_1`.

Example 6.20 (Element Completion)

```
?- gxl_element_to_element_complete(node:[]:[], Node).

Node = node:[id:id_1]:[
  prmtr:[color:grey, handles:default, mouse_click:node,
    size:small, symbol:circle, x_pos:224, y_pos:68]:[
      string:[bubble:'Default Node',
        font_style: (times, roman, 12)]:[default, node]]]

Yes
```

Changing IDs

Some GXL tools cannot treat all of our used IDs, even if they are atomic. In order to change all node IDs, edge IDs and referenced IDs into IDs, which can be used by other tools, we call

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2)
```

with the configuration of `Actions`

```
<actions>
   <action name="change_ids"/>
<actions>
```

This will generate for each ID a new ID, using `gensym(id_, ID)`. After changing the node IDs, we change the corresponding references (source- and target-node IDs) of the edges, too.

6.3.2 Deleting Nodes and Edges

In the following sections, we describe how to delete nodes and edges. We have implemented multiple possibilities to determine the properties of nodes and edges, which we want to delete.

Deleting Nodes or Edges using IDs

In order to delete nodes or edges of a GXL document, we call the method

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2)
```

using the configuration of `Actions`:

```
<actions>
   <action name="delete">
      <node id="id_10"/>
      ...
      <edge id="id_20"/>
      ...
   </action>
</actions>
```

The DTD looks as follows:

DTD:

```
<!ELEMENT action node* edge*>
<!ATTLIST action
   name delete #required>

<!ELEMENT node EMPTY>
```

```
<!ATTLIST node
   id CDATA #required>

<!ELEMENT edge EMPTY>
<!ATTLIST edge
   id CDATA #required>
```

Each `<node>`- or `<edge>` element with the corresponding ID in the GXL document will be deleted. The tag names `node` or `edge` are arbitrary. Only the ID `id` is important. Even if the tag name is `edge`, and the ID is the ID of a node, the corresponding node in the GXL document will be deleted.

Deleting Edges with Missing References

In the GXL standard, it is not allowed that an edge references a node, which is not contained in the GXL document itself. Even, all referenced nodes must be embedded in the same element `<graph>` of the GXL document. Our tool just writes a messages to the console, if such an edge exists. Some GXL visualization tools cannot handle such GXL documents. The call

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2)
```

together with the configuration

```
<actions>
   <action name="delete_missing_references"/>
</actions>
```

deletes all edges in a GXL document, which reference a non existing node in this document. Additionally, we have implemented the method

```
gxl_to_gxl(+Actions, +Gxl_1, -Edges, -Gxl_2),
```

in order to receive the IDs of the edges, which contain not existing node-IDs (cf. Section 6.2.1).

Deleting Light Weighted Edges

Using

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2)
```

together with

```
<actions>
   <action name="delete_light_weighted_edges" limit=Limit/>
</actions>
```

deletes all edges representing not more than `Limit` calls. The weight of an edge is readout from the attribute `weight` of the element `<edge>` in the GXL document.

Deleting Colored Edges

If we want to delete the edges that have a certain color, we call

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2)
```

together with the configuration

```
<actions>
   <action name="delete_colored_edges" color=Color/>
</actions>
```

Deleting Single Nodes

The predicate

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2).
```

used together with the configuration

```
<actions>
   <action name="delete_single_nodes"/>
</actions>
```

deletes all nodes in a GXL document, which are not referenced by an edge in this document. Additionally, we have implemented the method

```
gxl_to_gxl(+Actions, +Gxl_1, -Single_Nodes, -Gxl_2),
```

because sometimes, we need the IDs of the single nodes, e.g., in order to display these nodes in a table of single nodes.

Deleting Loops in Graphs

If we want to delete loops in GXL graphs, we call

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2)
```

with the configuration

```
<actions>
   <action name="delete_loops"/>
</actions>
```

A loop in a graph is an edge, which has the same source and target node. This configuration deletes the edges in a GXL document, which causes loops in a graph. Embedded edges, which cause loops in a picture will be deleted, too.

Example 6.21 (Deleting) We visualize a graph in a picture (cf. Section 6.4). The graph contains a node that contains a further, embedded graph. The nodes of the embedded graph are not visualized. Only the father node is shown in the picture. But, if the embedded graph contains an edge, which references two nodes contained in the embedded graph, an edge *from* the father node *to* the father node is drawn. We delete this loop, if we apply the transforming method mentioned above.

Deleting Embedded Graphs

If we want to delete all embedded graphs and change corresponding edges, which have either their source or their target node in an embedded graph, we call the method

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2).
```

with the configuration

```
<actions>
   <action name="delete_embedded_graphs"/>
</actions>
```

The edges, which are outside of an embedded graph and which have either their *source* or their *target node* in an *embedded graph* will be changed. The node ID of the embedded node will be replaced by the father node ID of the embedded graph. This can cause multiple edges with the *same* source and target node. Nevertheless, these edges will not be deleted, because they may be needed for any metric calculation. Multiple edges with the same *source* and *target* node are not disturbing, because they have different edge IDs. Only multiple edges, which have the same ID cause failure, when we visualize a graph.

Edges, which are inside an embedded graph have both – their source and their target node inside the embedded graph. These edges will be simply deleted.

Deleting the GXL Extensions

In order to export and use a GXL document together with other graph visualization tools, we have to delete our GXL extensions. The call

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2)
```

used with the configuration

```
<actions>
   <action name="delete_prmtr_tags"/>
</actions>
```

deletes all `<prmtr>` elements in the GXL document.

6.3.3 Highlighting Nodes and Edges

We have implemented multiple algorithms in order to highlight nodes or edges in dependency of their properties. In the following sections, we first introduce some basic highlighting methods. Later, we explain more complex highlighting methods.

Coloring Nodes and Edges

In order to color nodes or edges, we call

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2)
```

with the configuration

```
<actions>
   <action name="set_color" color=Color mode=Mode>
      <node id="id_10"/>
      ...
      <edge id="id_20"/>
   </action>
</actions>
```

`Gxl_1` contains the original GXL document, and `Gxl_2` contains the changed GXL document. The configuration `Actions` has the following DTD:

DTD:

```
<!ELEMENT action node* edge*>
<!ATTLIST action
   name set_color #required
   color CDATA #required
   mode all_edges|
        all_nodes|
        IDs|
        incoming_edges|
        outgoing_edges|
        transitive_closure #required>

<!ELEMENT node EMPTY>
<!ATTLIST node
   id CDATA #required>

<!ELEMENT edge EMPTY>
<!ATTLIST edge
   id CDATA #required>
```

The Element <action> The attributes of the element `<action>` have the following meaning:

`mode`
: can be either

- `all_edges`
  in order to color all edge lines,
- `all_nodes`
  in order to color all node symbols,

- `IDs`
  in order to color the node symbols or edge lines contained in the content list of the element `<action>`,
- `incoming_edges`
  in order to color the edge lines, which are directed *to* one of the nodes contained in the content list of the element `<action>`,
- `outgoing_edges`
  in order to color the edges, which are directed *from* one of the nodes contained in the content list of the element `<action>`,
- `transitive_closure`,
  in order to color the edges of the transitive closure, starting with the nodes contained in the content list of `<action>`.

`color`
: determines the color of the edge lines or node symbols. The value of `color` is analogous to the attribute `background`, page 147.

The Elements <node> and <edge> The attribute `id` of the element `<node>` and `<edge>` contains the ID of GXL node or a GXL edge. If the attribute `mode` has the value `all_edges` or `all_nodes`, we do not need any `<node>` or `<edge>` elements containing GXL node- or GXL edge IDs.

Highlighting Important Edges

We can highlight the edges of a GXL document in dependency of their weight. In the first step we determine the weight of each edge, and then, in the second step, we allocate a color to each edge or we change the thickness of each edge in dependency of the weight. The following example configuration assigns each edge to a *weight class*.

Example 6.22 (Highlighting Importing Edges)

```
<actions>
   <action name="edge_weight">
      <weight_classes mode="color">
         <weight_class alias="few" color="yellow"
            min="0" max="5" fade_out="true"/>
         <weight_class alias="moderate"
            color="orange" min="6" max="25"/>
         <weight_class alias="a_lot"
            color="red" min="26"/>
      </weight_classes>
   </action>
</actions>
```

The weight of an edge is readout from the attribute `weight` of the element `<edge>` in the GXL document. The value of `weight` of each edge in the GXL document has to be generated before, using a separate algorithm. We have implemented an algorithm (cf. the end of this section), which determines the weight in dependency of the amount of calls, which an edge is representing. This algorithm generates a suitable configuration of *weight classes*, too. Each edge gets the color defined in the *weight class*.

Instead of *coloring* the edges, we can change the *thickness* of an edge depending on its weight. In this case, the edges are only classified into three groups: thin edges, which represent a small weight, medium edges, which represent an average weight, and thick edges, which represent a large weight.

The corresponding DTD looks as follows.

DTD:

```
<!ELEMENT action weight_classes>
<!ATTLIST action
   name edge_weight #required>

<!ELEMENT weight_classes weight_class>
<!ATTLIST weight_classes
   mode (color|thickness) #required>

<!ELEMENT weight_class EMPTY>
<!ATTLIST weight_class
   alias CDATA #required
   color fade_out|CDATA #required
   fade_out true|false #implied
   min CDATA #required
   max CDATA #implied>
```

The Element <weight_classes> This element has the attribute `mode` and contains the elements `<weight_class>`. These elements define diverse groups, in which we classify the edges. The attribute `mode` determines the method of the weighting. It can be

`color`,
: which changes the color of the edges in dependency of their weight and it can be

`thickness`,
: which changes the thickness of the edges in dependency of their weight.

The weight of an edge is specified by the attribute `weight` of an `<edge>` element in the GXL document.

The Element <weight_class> This element defines a group, into which we can classify the edges.

`alias`,
: is an alias name for this class.

`color`,
: determines the color of the edges belonging to this class. Possible values are analogously to the attribute `background`, page 147.

`fade_out`,
: can be `true` or `false` and determines, if the edges belonging to this class are shown or fade out.

`min`, `max`
: are integers and determine the weight borders. A missing attribute `max` sets the value to infinite.

In order to determine the weight in dependency of the amount of calls, which an edge represents, and to generate a suitable configuration of weight-classes we call

```
gxl_to_gxl(+Actions_1, +Gxl_1, -Actions_2, -Gxl_2)
```

together with the configuration

```
<actions>
   <action name="call_weight"/>
</action>
```

The edges must contain the call information. This information is added to the GXL document `Gxl_1`, e.g., when we generate the Rule/Goal graph (cf. Section 7.2.1). The GXL document `Gxl_2` contains the attribute `weight`, now. The configuration `Actions_2` can be used as input configuration of the method `gxl_to_gxl/3`, in order to generate a new GXL document, in which the the edges are highlighted.

Highlighting Tree-, Forward-, Back- and Cross-Edges

In Section 6.2.2, we introduced a method to determine the *tree*, *forward*, *back*, and *cross edges*. We use this method, in order to mark each edge type with an individual color. The call

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2)
```

used with the following configuration creates a new GXL document in which the *tree*, *forward*, *back* and *cross edges* are highlighted. The colors of the diverse edge types are defined in the configuration `Actions`.

Example 6.23 (Configuration)

```
<actions>
   <action name="highlight_tfbc_edges">
      <edge_colors>
         <edge_color type="tree_edge" color="blue"/>
         <edge_color type="cross_edge" color="green"/>
         <edge_color type="back_edge" color="yellow"/>
         <edge_color type="forward_edge" color="red"/>
      </edge_colors>
   </action>
</actions>
```

The corresponding DTD looks as follows:

DTD:

```
<!ELEMENT action edge_color>
<!ATTLIST action
   name highlight_tfbc_edges #required>

<!ELEMENT edge_colors edge_color*>

<!ELEMENT edge_color EMPTY>
<!ATTLIST edge_color
   type tree_edge|cross_edge|
        back_edge|forward_edge #required>
   color CDATA #required>
```

Example 6.24 (Coloring Tree, Forward, Back, and Cross Edges) In this example, we first determine the tree (black), forward (red), back (orange), and cross edges (green) of a given GXL graph `Gxl_1` and create the new graph `Gxl_2`. Then, we visualize the graph with the colored edges (cf. Figure 6.5).

```
?- Gxl_1 = gxl:[...]:[...],
   Actions = action:[]:[
      action:[name:highlight_tfbc_edges]:[
         edge_colors:[]:[
            edge_color:[type:tree_edge, color:black]:[],
            edge_color:[type:cross_edge, color:green]:[],
            edge_color:[type:back_edge, color:orange]:[],
            edge_color:[type:forward_edge, color:red]:[] ] ] ],
   gxl_to_gxl(Actions, Gxl_1, Gxl_2),
   gxl_to_picture(Gxl_2).

Yes
```

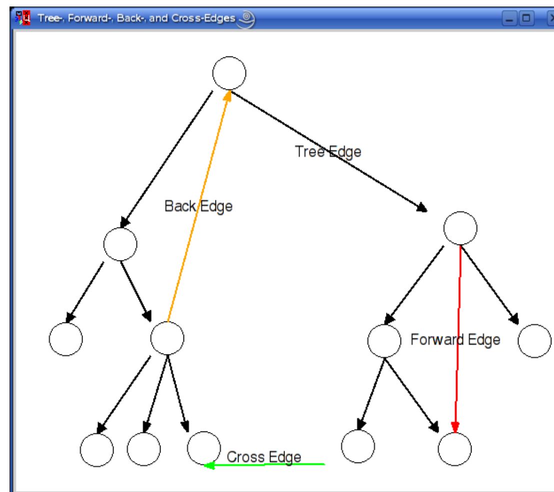


Figure 6.5: Tree-, Forward-, Back-, and Cross Edges

6.3.4 Merging Nodes

We can merge several nodes of a GXL document to *one merged node*. We generate a new node, which represents a chosen set of nodes. All edges, which are connected to the merged nodes, now are connected to the new node. This compacts a large graph and improves the visualization of a large graph containing a lot of nodes and edges. It is possible to configure the representing, merged node so that we can zoom into the node (cf. Section 6.3.4) in such a way that we can visualize the corresponding merged nodes and edges contained in the node. Depending on the method, which is called when we click on the node, a new picture opens and shows the merged nodes and edges. The method

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2).
```

merges the nodes, which IDs are listed in the configuration `Actions`, to a *merged node*. The appearance of the merged node is defined in the configuration `Actions`, too. The corresponding DTD of the configuration `Actions` looks as follows:

DTD:

```
<!ELEMENT action components>
<!ATTLIST action
   name merge_nodes #required>

<!ELEMENT components merged_node+>
<!ATTLIST components
   mode keep_uninvolved_edge_ids|
        change_uninvolved_edge_ids #required>
```

```
<!ELEMENT merged_node prmtr? nodes>

<!ELEMENT nodes node+>
<!ATTLIST node
   id CDATA #required>
```

Example 6.25 (Configuration)

```
<actions>
   <action name="merge_nodes">
      <components mode="keep_uninvolved_edge_ids">
         <merged_node>
            <prmtr color="red" mouse_click="embedded_node"
              size="large" symbol="honeycomb">
                 <string>Composite Node</string>
            </prmtr>
            <nodes>
              <node id="id_1"/>
              <node id="id_2"/>
            </nodes>
         </merged_node>
         ...
      </components>
   </action>
</actions>
```

The meaning of the elements and attributes is explained in the following.

The Element <components>

The element `<components>` contains the configuration of all merged nodes. Thereby `mode` determines, how to handle the edge IDs. It can be one of the following values:

`keep_uninvolved_edge_ids`
: generates a new GXL document, in which all source IDs and target IDs of all edges are kept unchanged.

`change_uninvolved_edge_ids`
: generates a new GXL document and changes the `from` and `to` IDs of the edges, which are moved into a single node and which refer to a node outside this single node to the ID of the main nodes.

Example 6.26 (Changing Node IDs) Consider we have six nodes n1, n2, n3, n4, n5, n6 and four edges n1-n2, n2-n3, n2-n5, n6-n1 (cf. Figure 6.6). The nodes n1, n2 are merged to the merged node m1, and the nodes n3, n4 are merged to the merged node m2. Using the mode `change_uninvolved_edge_ids`, the source and target IDs of the edges are changed into the following way:

```
n2-n3 -> m1-m2
n2-n5 -> m1-n5
n6-n1 -> n6-m1
```

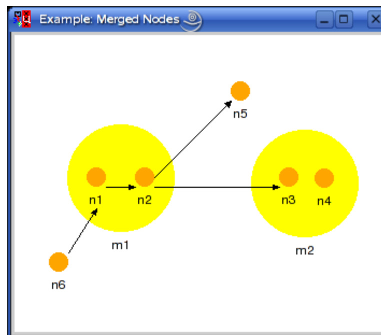


Figure 6.6: Merged Nodes

The Element <merged_node>

This element describes the appearance of the merged node. It contains the same elements and attributes as a normal GXL node. Missing elements or attributes will be added. The appearance is configured in the element `<prmtr>`.

The Element <prmtr>

This element is the same element of the GXL nodes, which we describe in the Section 6.1.1, page 148. Missing attributes will be added automatically.

The Elements <nodes> and <node>

The element `<nodes>` contains the IDs of the nodes, which we want to merge to one merged node. If this element only contains one node in the content, nothing will be done and the original node will be left unchanged.

Each element `<node>` contains one ID of a node, which we merge with the other nodes to a merged node. Thereby the attribute `id` is a reference to the same ID, which is used in the original GXL document.

Example 6.27 (Merging Nodes and Changing IDs) Consider the following GXL document. We have six nodes and four edges (cf. Figure 6.6). We want to merge node `n1`, `n2` and node `n3`, `n4`.

```
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
   <graph id="graph_1" edgeids="true"
     edgemode="directed" hypergraph="false">
      <node id="n1">...</node>
      <node id="n2">...</node>
      <node id="n3">...</node>
      <node id="n4">...</node>
      <node id="n5">...</node>
      <node id="n6">...</node>
```

```
      <edge from="n2" id="e1" to="n3"/>
      <edge from="n2" id="e2" to="n5"/>
      <edge from="n1" id="e3" to="n2"/>
      <edge from="n6" id="e4" to="n1"/>
   </graph>
</gxl>
```

In order to generate a suitable configuration, which merges the nodes, we call

```
?- generate_merge_configuration(list_2, Gxl_1,
      [[n1, n2], [n3, n4], [n5], [n6]], Actions),
   gxl_to_gxl(Actions, Gxl_1, Gxl_2).

Yes
```

We explain `generate_merge_configuration/4` below. This method generates the following XML configuration document.

```
<actions>
  <action name="merge_nodes">
    <components mode="change_uninvolved_edge_ids">
      <merged_node>
        <prmtr color="red" ...>...</prmtr>
        <nodes>
          <node id="n1"/>
          <node id="n2"/>
        </nodes>
      </merged_node>
        <merged_node>
          <prmtr color="red" ...>...</prmtr>
          <nodes>
            <node id="n3"/>
            <node id="n4"/>
          </nodes>
      </merged_node>
      <merged_node>
        <prmtr color="red" ...>...</prmtr>
        <nodes>
          <node id="n5"/>
        </nodes>
      </merged_node>
        <merged_node>
        <prmtr color="red" ...>...</prmtr>
        <nodes>
          <node id="n6"/>
        </nodes>
      </merged_node>
    </components>
  </action>
</actions>
```

At last, we call `gxl_togxl/3` using the new created XML configuration document and the GXL document as input. Then, we get the following GXL document, which contains the merged nodes.

```
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
   <prmtr .../>
   <graph id="graph_1" ...>
      <node id="n6">...</node>
      <node id="n5">...</prmtr>
      <node id="m_1">
         <prmtr color="red" ...>...</prmtr>
         <graph id="g_1" ...>
            <node id="n1">...</node>
            <node id="n2">...</node>
            <edge from="n1" id="e3" to="n2"/>
         </graph>
      </node>
      <node id="m_2">
         <prmtr color="red" ...>...</prmtr>
         <graph id="g_2" ...>
            <node id="n3">...</node>
            <node id="n4">...</node>
         </graph>
      </node>
      <edge id="e1" from="m_1" to="m_2">...</edge>
      <edge id="e2" from="m_1" to="n5">...</edge>
      <edge id="e4" from="n6" to="m_1">...</edge>
   </graph>
</gxl>
```

Creating a Configuration for Merging Nodes

In order to get a proper configuration `Actions`, containing the node IDs, which we want to merge, we have implemented the method

```
generate_merge_configuration(+Mode, +Gxl,
      +Components, -Actions).
```

which transforms a list of lists, each list containing node IDs, to a valid XML configuration document, which can be used as input of the method `gxl_to_gxl/3`. The arguments have the following meaning:

`Mode`
: In order to change between different configuration possibles, we use the argument `Mode`. We can define additional methods, which generate a suitable configuration. Up to now, `Mode` can have the value `list`, `list_2`, `bubble`, or `name`. Depending on this value, the merged nodes have different properties. In order to configure these properties, we edit the file `'gxl/generate_merge_config.pl'`.

`Gxl`
: is the original GXL document containing the graph.

`Components`
: is a list of lists. Each list represents a merged node and contains the node IDs, which we want to merge.

`Actions`
: contains a suitable configuration for the method `gxl_to_gxl/3`, if we want to merge nodes.

Merging nodes is, e.g., used in the method, which contracts the strongly connected components. First, we determine the IDs of all nodes belonging to a strongly connected components. After this, the nodes of each strongly connected component are merged to a merged node, which contains the original nodes and edges as an embedded graph. At last, the new GXL graph is generated.

Strongly Connected Components (SCC)

Merging nodes is used by the method, which detects the strongly connected components (scc) of a graph. Thereby, the elements of a scc are merged to one node. The result is a new GXL graph, in which each node represents a scc. In the following subsections, we explain how to detect the scc of a graph and how we can highlight the corresponding nodes, which belong to the same scc.

Determining the Strongly Connected Components In order to determine the scc, we call

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2),
```

using the configuration

```
<actions>
   <action name="scc"/>
</actions>
```

In the new graph, each scc-node contains an *embedded graph*. The members of this embedded graph are the nodes of the original graph, belonging to the same scc. We can configure the scc-nodes in a way, so that clicking on a scc-node opens a new picture, showing the embedded graph with the nodes of the scc.

We call `gxl_to_picture(Config_P, Gxl_2, Picture)` in order to visualize the new graph (cf. Section 6.4). The scc-node contains the nodes of the original graph, which belong to this scc. Using the following example configuration, we can click on a scc-node. A new picture opens and we see the embedded graph of the scc-node.

Example 6.28 (Configuration for Embedded Graphs) An example of the corresponding configuration for the call `gxl_to_picture/3` looks as follows:

```
<config>
   <mouse_clicks>
      <mouse_click alias="embedded_node">
         <on_click
            button="middle"
            type="single"
```

```
            module="user"
            predicate="show_embedded_graph"
            prmtrs="[@1, @2]"
            message="prolog"/>
      </mouse_click>
   </mouse_clicks>
</config>
```

Using this configuration, a single click on a scc-node with the middle mouse button calls the method `show_embedded_graph/2`. This method is implemented in the following way:

Implementation:

```
1   show_embedded_graph(Picture, Node_Address) :-
2      get(Picture, xml, GXL),
3      get(Picture, config, Config_P),
4      xpce_address_to_gxl_id(Node_Address, Node_ID),
5      GXL_Node := GXL/_/node::[@id=Node_ID],
6      Sub_Graph := GXL_Node/graph,
7      gxl_presetting(head, gxl:Attr_1:_),
8      Gxl = gxl:Attr_1:[Embedded_Gxl_Graph],
9      gxl_to_picture(Config_P, Gxl, _Picture_1).
```

Highlighting the Strongly Connected Components In order just to identify the scc of a graph, we mark the edges and nodes belonging to the same scc with the same symbol and the same color. The nodes are not merged together to one node, which represent the scc. The call

```
gxl_to_gxl(+Actions, +Gxl_1, -Gxl_2)
```

with the following configuration changes the symbol and color of each node and the color of each edge belonging to the same scc, so that all nodes and edges of a scc have the same symbol and color.

Example 6.29 (Configuration for Coloring SCC)

```
<actions>
   <action name="highlight_scc"/>
      <symbols scc_size="medium">
         <symbol name="circle"/>
         <symbol name="box"/>
      </symbols>
      <colors>
         <color name="black" scc="true"/>
         <color name="green" scc="true"/>
      </colors>
   </action>
</actions>
```

This is the corresponding DTD of the configuration.

DTD:

```
<!ELEMENT symbols symbol*>
<!ATTLIST symbols
   scc_size CDATA #required>

<!ELEMENT symbol EMPTY>
<!ATTLIST symbol
   name CDATA #required>

<!ELEMENT colors color*>

<!ELEMENT color EMPTY>
<!ATTLIST color
   name (black|green|red|orange|CDATA) #required
   scc (true|false) #required>
```

The Elements <symbols> and <symbol> These elements specify the available symbols for the strongly connected components. The attribute `scc_size` of the element `symbols>` determines the symbol size of all strongly connected components.

The attribute `name` of the subelement `<symbol>` is the name of an available symbol, which is implemented in the PROLOG file `gxl/symbols.pl`. Only the listed symbols are used, when we automatically generate strongly connected components. New symbols can be implemented in the file `gxl/symbols.pl`, too.

The Elements <colors> and <color> These element contain all available colors for the strongly connected components. Each subelement `<color>` specifies a color. The subelement `<color>` has the following attributes:

`name`
: is the name of a available color,

`scc`
: can be `true` or `false` and determines, if the color can be used for a symbol of a strongly connected component. Only the `<color>` elements with the attribute value `true` are used for the symbols of strongly connected components, when we automatically generate the graph.

The product of the available symbols and the amount of colors has to be greater than the existing strongly connected components of a graph, so that each strongly connected component can be assigned to a distinct pair of color and symbol.

6.4 The Picture Class for GXL Graphs

We have implemented a method, which visualizes a graph contained in a GXL document in an XPCE picture and which adds the GXL document as an object to the picture. Addi-

tionally, we pass an XML configuration document and add this XML configuration document as an object to the picture. Using the XML configuration document, we configure nodes, edges, pop-up menus and mouse click interactions.

Furthermore, we have implemented a method, which extracts a graph, visualized in an arbitrary picture. We export this graph into a GXL document. Using the conversion methods of Section 6.2.3, we convert the GXL document into the graph format, which we prefer, and vice versa.

6.4.1 Visualization of GXL Graphs and Ugraph

The following describes the methods to visualize graphs in a picture. We can even add further vertices and edges to an existing picture, which already visualizes a graph. To reuse a picture, we have implemented a method, which deletes the content of an existing picture.

Clearing a Picture from Nodes and Edges

If we want to clear a picture and delete all nodes and edges visualized in the picture, we call the method

```
xpce_picture_clear(+Picture).
```

This frees all objects in the picture `Picture`. It is not sufficient to call the XPCE method

```
send(+Picture, clear).
```

Unfortunately, this will not free all objects in the picture. The left object-addresses may cause problems later. For this reason, we propose only to use the method mentioned above in order to clear a picture in a clean way.

Visualizing GXL Graphs and Ugraphs

In order to visualize a GXL graph, we call the method

```
gxl_to_picture(?Conf, +Gxl, ?Pic).
```

This method visualizes the graph in the picture with the XPCE address `Pic`. If the XPCE address `Pic` is unbound, then a new XPCE picture will be created and the XPCE address of the picture will be returned in the variable `Pic`.

`Conf` is an XML document which defines, e.g., the properties of nodes, edges, arrows, mouse click events, and pop-up menus. The following cases exists for the configuration `Conf`:

- We can use the empty configuration `Conf = config:[]:[]` which specifies no properties .

- If we send a GXL graph to an existing picture (the variable `Pic` is bound), and the passed variable `Conf` is unbound, then we use the existing configuration of the picture, if there already exists a configuration. Otherwise, we use a predefined default configuration. Depending if there exists a configuration of the picture, the variable `Conf` will be bound to the existing configuration or the default configuration.
- If we send a GXL graph to an existing picture (the variable `Pic` is bound), and the passed variable `Conf` is bound to an XML configuration document, then we overwrite the existing configuration of the picture with the new passed configuration.

We describe the configuration possibilities of the XML document `Conf` in Appendix B.

The variable `Gxl` contains the GXL document, including the GXL graph. This variable always has to be a GXL document.

Remark: Each XPCE object in an XPCE picture has an XPCE address. Using this XPCE address, we have access to an XPCE object. The XPCE address can be chosen randomly by the system, or we can determine the name of the XPCE address ourself.

For the nodes and edges of a GXL document, we determine the name of the XPCE address ourself. If a GXL graph is send to a picture using `gxl_to_picture/3`, the value of the attribute `id` of a node or edge is transformed to a valid XPCE address of the node or edge in the picture. To transform an ID to an XPCE address, we call

```
gxl_id_to_xpce_address(+Pic_Addr, +GXL_ID, -XPCE_Addr).
```

Summarized, the method `gxl_id_to_xpce_address/3` concatenates the address of the XPCE picture and the GXL ID of a node or edge to an atom. The inverted method

```
xpce_address_to_gxl_id(+XPCE_Addr, -GXL_ID)
```

removes the XPCE address of the picture, in order to get the ID, which is used in the GXL document.

Example 6.30 (GXL to XPCE Address)

```
?- gxl_id_to_xpce_address(@1234, id_1, XPCE_Addr).

XPCE_Addr = @'id_1-1234'

Yes

?- XPCE_Addr = @'id_1-1234',
   xpce_address_to_gxl_id(XPCE_Addr, GXL_ID).

GXL_ID = id_1

Yes
```

The first call concatenates the XPCE address of the picture and the ID. In the second call splits the result of the first call and gets the pure ID.

Adding Supplementary Nodes and Edges Sometimes, it is necessary to complete a visualized graph with further nodes and edges. For this reason, we can send an additional GXL document, containing further nodes and edges to an existing picture. It is not necessary that the additional GXL document contains the old GXL document of the picture. In the additional GXL document, we can even reference node IDs of the existing, visualized GXL document.

If we send an additional GXL document to an existing picture, only the nodes and edges of the additional GXL document will be added to the graph of the existing GXL document of the picture. Further arguments, contained in the passed GXL document are ignored. E.g., the label or the size of the picture will not be replaced by the arguments given in the passed GXL document.

Furthermore, it is not possible to add a new node or edge to a GXL document, if the GXL document already contains a node or edge with the same ID. Such nodes or edges will be ignored. All IDs of a GXL document have to be unique.

Example 6.31 (GXL to Picture) In this example, we send nodes and edges separately to a picture (cf. Figure 6.7). All GXL documents are incomplete. This does not matter, because the incomplete elements are completed automatically by the values of the default GXL document.

First, we pass the empty XML configuration document `C_1` and one node. The variable `Pic` is unbound. The method creates a new picture and the address of the picture is bound to the variable `Pic`.

```
?- C_1 = config:[]:[],
   Gxl_1 = gxl:[]:[graph:[]:[node:[id:a]:[]]],
   gxl_to_picture(C_1, Gxl_1, Pic).

Pic = @2702860/picture

Yes
```

Then we pass the short XML configuration document `C_2` and a second node to the same picture. The node is added to the GXL document of the picture and the former XML configuration document is replaced by the new one.

```
?- C_2 = config:[]:[gxl_layout:[mode:bfs]:[]],
   Gxl_2 = gxl:[]:[graph:[]:[node:[id:b]:[]]],
   gxl_to_picture(C_2, Gxl_2, @2702860).

Yes
```

After this, we pass the unbound variable `C_3` and a third node to the picture. We retrieve the current XML configuration document of the picture. Now, the unbound variable `C_3` is bound to the current XML configuration document.

```
?- Gxl_3 = gxl:[]:[graph:[]:[node:[id:c]:[]]],
   gxl_to_picture(C_3, Gxl_3, @2702860).

C_3 = config:[]:[gxl_layout:[mode:bfs]:[]]

Yes
```

At last, we use the method `gxl_to_picture/3` to send two edges to the picture. This method passes the default XML configuration document to the picture and replaces the existing XML configuration document. The two edges are added to the GXL document of the picture, and they are visualized in the picture (cf. Figure 6.7).

```
?- Gxl_4 = gxl:[]:[graph:[]:[
      edge:[from:a, to:b, id:d]:[],
      edge:[from:a, to:c, id:e]:[]]],
   gxl_to_picture(_, Gxl_4, @2702860).

Yes
```

If the label of a GXL node is not specified, then the ID of the node is used as label. If in addition the ID of a GXL node is not specified, then the label of the default node is used as label and a random ID is generated. The default GXL configuration is introduced in Section 6.1.1.

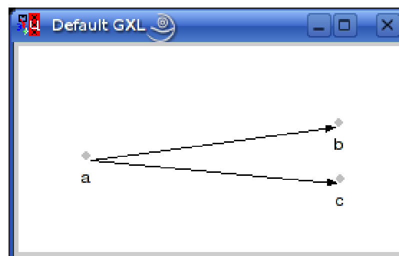


Figure 6.7: Sending Nodes and Edges Separately to a Picture

Shortcuts

In the following, we have combined methods to useful and often used shortcuts.

Vertices and Edges to Picture In order to send a graph which is given by a set of vertices `Vs` and a set of edges `Es` to a picture `Pic`, we combine the two methods

```
vertices_and_edges_to_gxl/3,
gxl_to_picture/3
```

to the method

```
vertices_and_edges_to_picture(+Mode, +Vs, +Es, ?Pic).
```

The argument `Mode` selects between different layout algorithms. The available layout algorithms are described in Section 6.1.2, page 156.

The list `Vs` contains the nodes and the list `Es` contains the edges. Each entry of the node list has to be atomic. It is used as label and as node ID in the GXL document and in the visualization. In order to define the symbol, size, color, and further arguments of a node or an edge, each element of these two lists can be a GXL-node or a GXL-edge element, too.

`Pic` can be unbound or it can be bound to an XPCE address, if we want to add nodes or edges to an existing picture.

Example 6.32 (Vertices and Edges to Picture) The following call creates a graph with three vertices and two edges (cf. Figure 6.8). In this case the label and the ID of the vertices are identical.

```
?- vertices_and_edges_to_picture(bfs, [a, b, c], [a-b, a-c], Pic).

Pic = @1234567/picture

Yes
```

Ugraph to Picture In order to layout and visualize a *Ugraph* in a picture, we combine the two methods

```
ugraph_to_gxl/2,
gxl_to_picture/3
```

to the method

```
ugraph_to_picture(+Mode, +Ugraph, ?Pic).
```

Analogously above, the argument `Mode` selects between different layout algorithms. The available layout algorithms are described in Section 6.1.2, page 156.

`Ugraph` contains a Ugraph. Each entry of the Ugraph has to be atomic. It is used as a label and as a node ID in the GXL document and in the visualization. Unlike above, GXL elements cannot be used instead of atomic elements.

`Pic` can be unbound or bound to an XPCE address, if we want to add nodes or edges to an existing picture.

Example 6.33 (Ugraph to Picture) In the following example we send a Ugraph to a XPCE picture. Then we layout the graph, and at last we send the graph to a picture `Pic` (cf. Figure 6.8):

```
?- ugraph_to_picture(xpce, [a-[b, c], b-[], c-[]], Pic).

Pic = @1234569/picture

Yes
```

Because of the variable `Pic` is unbound, a new picture is created and the XPCE address of this picture is returned in the variable `Pic`.

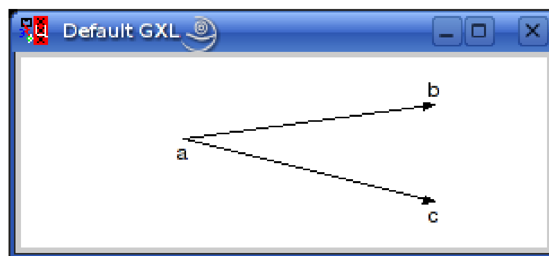


Figure 6.8: A Graph with Three Nodes in an XPCE-Picture

6.4.2 Reconstruction of Graphs from Pictures

Due to the fact that there exist actions, which work directly on the nodes and edges of an XPCE picture without using the underlying GXL document (e.g., if we rearrange nodes with the mouse-pointer), we have implemented a method to obtain a GXL document from any XPCE picture containing a graph.

Picture to GXL Graph Using the following method, we retrieve a GXL document and the corresponding configuration from a picture. If the picture contains already a GXL document, this GXL document is returned. Otherwise, a GXL document of the visualized graph in the picture is generated and returned.

```
picture_to_gxl(+Pic, -Gxl, -Config).
```

We look, if the picture contains a GXL and an XML configuration document as payload and adapt the changes made in the picture to the GXL document. If there exists no GXL document, then we use the default values in order to create a new GXL document. The default values of a GXL document (cf. Section 6.1.1) are stored in a file, which is associated to the alias `gxl_presettings`. If there exists an XML configuration of the picture, the XML configuration is returned. Otherwise, the empty configuration `config:[]:[]` is returned.

Remark: Using the methods `gxl_to_picture/3`, we attach the whole GXL document and the XML configuration document as payload to the XPCE object *picture*. If we want to obtain this payload, we call

```
get(+Picture, gxl, -Gxl),
get(+Picture, config, -Config_P).
```

Normally, there is no need to obtain the payload of a picture. Just, if we want to transform the nodes and edges of a picture back to a GXL document, (e.g if nodes have been moved) we look for this payload and use it as presetting. Then we can incorporate the changes made in the picture into this GXL document.

We have implemented further visualization methods by combining the method

```
picture_to_gxl/3
```

with the conversion methods of Section 6.2.3. Now, we introduce two combined methods, using these conversion methods.

Shortcuts

In the following, we have combined methods to useful and often used shortcuts.

Picture to Vertices and Edges In order to receive only the nodes and edges of the graph in a picture, we combine the methods

```
picture_to_gxl/3,
gxl_to_vertices_and_edges/4.
```

The resulting method

```
picture_to_vertices_and_edges(+Pic, -Vs, -Es)
```

returns all visualized nodes and edges of the picture `Pic`. The node list `Vs` contains the node IDs and the edge list `Es` contains the source and target node IDs of each edge.

Picture to Ugraphs In order to receive a Ugraph of the visualized nodes and edges in a picture, we combine the methods

```
picture_to_gxl/3,
gxl_to_vertices_and_edges/4
vertices_edges_to_ugraph/3.
```

The resulting method

```
picture_to_ugraph(+Pic, -Ugraph)
```

gives the required Ugraph.

Chapter 7

The Visualization Tool VISUR

In this chapter, we first explain the GUI of VISUR. Using the GUI, we can execute a lot of commands without using the command line of SWI-PROLOG. The GUI of VISUR is an example, which shows, how we can easily configure menus using XML documents and the XPCE programming techniques mentioned in Appendix A.

The case studies show how to visualize DTDs, XML documents, ER diagrams, PROLOG package call dependencies. Using VISUR, we generate cross reference and call dependency graphs.

7.1 The System VISUR

The GUI combines the methods of the PROSORE database, RAR, VISUR, and GXL and makes these methods easily accessible for the user. It is an application, which uses the XPCE programming techniques of Appendix A. We start the GUI calling

```
visur_gui.
```

This reloads the last used configuration files and the last used PROSORE database. Depending on the number of facts of the PROSORE database, this takes few seconds.

The GUI of VISUR has several sections (cf. Figure 7.1). First, we have the menu bar. Below the menu bar, we have the symbol bar, which contains shortcuts of some frequently used commands of the menu bar. The area below the symbol bar contains the hierarchy browser. The hierarchy browser visualizes the hierarchy tree of the actual loaded PROSORE database and contains pop-up menus, which, among other things, generate graphs. Each graph will be visualized in an own, new picture. On the right side of the hierarchy browser, we have a source code viewer, which shows the source code of the file marked in the hierarchy browser. The status bar is below the hierarchy browser and the source code viewer. The status bar contains information about system failures, e.g., if a configuration file could not be loaded.

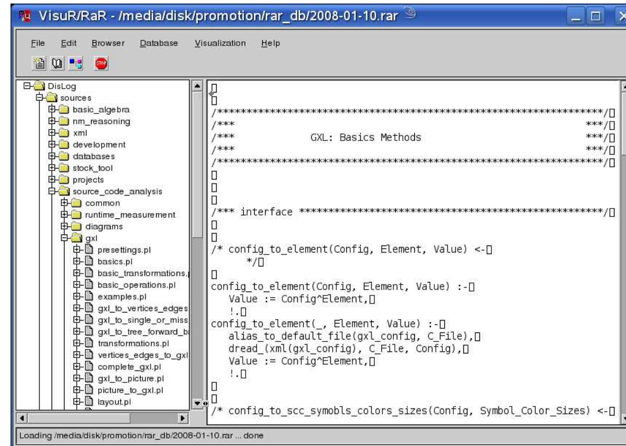


Figure 7.1: The GUI of VISUR

7.1.1 The Menu Bar

The menu entries are defined in the configuration file, which is associated to the alias `visur_menu_bar`. The menu entries can be changed by editing the configuration file or by loading another configuration file. Using the default configuration, the menu bar contains the 6 main menus *File*, *Edit*, *Browser*, *Database*, *Visualization*, and *Help*.

File

The menu *File* is divided into the following 6 submenus.

New Project resets all used variables and deletes the content of the PROSORE database. A new project can be started by choosing the source code files in the menu *Browser*.

Load Project... displays a file selection window for loading another PROSORE database. Typically, the extension of a PROSORE database is `.rar`, but the PROSORE database can be saved in XML, too. Then the extension will be `.xml`. If the PROSORE database is in the proprietary RAR format, loading and saving the database is faster than calculating the database new. If we load or save the database in XML format, loading or saving takes some time.

Reload Project shows a choice of all PROSORE databases, which have been loaded any time before. The items are order by the time of the last usage. If a new PROSORE database is saved, the path with name of the file is added on the top of the list. Clicking on an item clears the actual database without confirming and reloads the clicked PROSORE database. When we start the GUI, this list will be updated. The existence of all items in the list will be verified and non existing item will be deleted from the list.

Save Project saves the actual used PROSORE database. If the database was loaded from a file, the file will be overwritten with the PROSORE database without confirmation. If it is a new generated database which was not saved yet, a file selection windows will be shown, and we can determine a file name and location.

Save Project as... saves the PROSORE database with a new name. A file selection windows will be shown, and we can determine a file name and location. If the file already exists, we will be asked if we want to overwrite it.

The PROSORE database can be saved in the proprietary RAR format, or in XML. The format, in which the database will be saved depends on the file extension we choose in the file selection drop-down menu. Saving the database in the proprietary format is much faster than saving it in XML.

Exit will exit the GUI in a clean way. All facts of the PROSORE database will be retracted and the content of the used variables will be deleted.

Edit

The menu *Edit* contains the submenus *Configuration*, *Used Configuration Files*, and *Alias Browser*.

Configuration enables the user to load configurations. The configurations are XML documents with the file extension `.xml`. Starting the GUI of VISUR next time, the last used configuration file will be reloaded again. In order to modify the configuration, each configuration file can be loaded into the emacs editor of SWI-PROLOG.

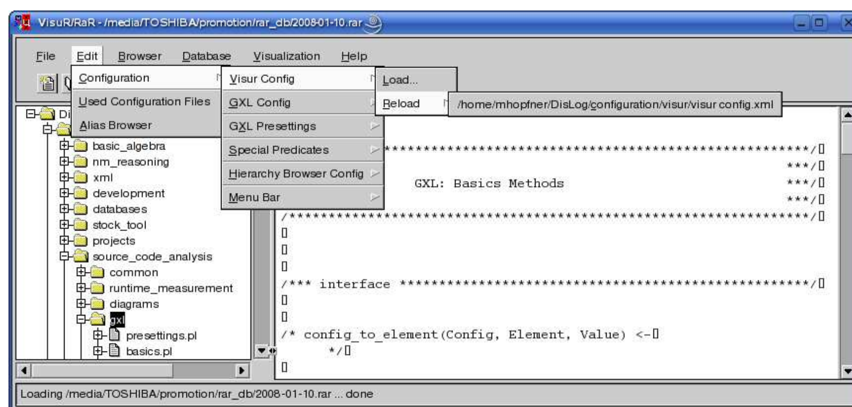


Figure 7.2: The Menu *Edit*

Used Configuration Files is called to see a table of the actual used configuration files.

Alias Browser calls the alias browser (cf. Appendix A.5).

Browser

Using the menu *Browser*, we can load a new hierarchy tree, we can generate the hierarchy tree of an arbitrary directory, or we can generate the hierarchy of DISLOG again. The hierarchy will be visualized in the browser automatically.

Database

The menu *Database* supports the handling with the PROSORE database. It contains the submenus *Generate Database*, and *Update Database*.

Generate Database calculates the PROSORE database in dependency of the hierarchy, which is shown in the hierarchy browser. First, the actual PROSORE database will be reset. Then all PROLOG files of the hierarchy tree shown in the hierarchy browser on the left side will be determined. After this, the files will be parsed and the source code will be transformed into XML and added as facts to the PROSORE database.

Update Database updates the PROSORE database incremental after some changes in the source code files have been made or new files were added to the hierarchy or deleted from the hierarchy.

Visualization

Dependency Graph... shows a form. We can choose a predicate or a file. We visualize the call dependency graph of the chosen predicate or file.

System Statistics (Table) shows the table about the number of units, modules, files, predicates etc.

Unit Statistics (Table) shows a table with a statistics about the predicates, rules and lines of source code of each unit.

File Statistics (Table) shows a table with a detailed statistics about the directives, outgoing and incoming calls, and the lines of source code of each file.

Dead Code (Table) shows a table with the dead code predicates.

Excluded Dead Code (Table) shows the table with the excluded dead code predicates.

Undefined Code (Statistics shows a table with the undefined predicates.

Undefined Code (Extended shows a table with detailed information about the undefined code.

Strongly Connected Components shows a table with a statistics about the strongly connected components.

Prolog Modules to files... shows a table with all PROLOG modules and the corresponding files, which belong to a PROLOG module

Distributed Predicates shows a table with all predicates, which are defined in more than one file.

Predicates Defined As... shows the multifile, distinguished and dynamic defined predicates.

Help

This menu just contains the menu *Version Info*.

The Symbol Bar

The symbol bar contains shortcuts for frequently used menus of the menu bar. These shortcuts are fixed and cannot be changed. The first shortcut loads a new PROSORE database and the second shortcut loads a new configuration file, which will be associated to the alias `visur_config`. The third shortcut loads a hierarchy in the hierarchy browser from a file, and the last shortcut exits the GUI.

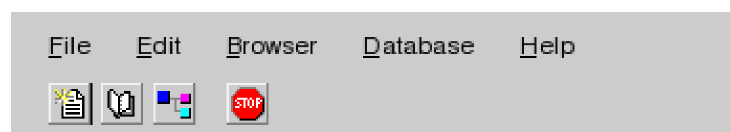


Figure 7.3: The Shortcuts in the Symbol Bar

7.1.2 The Hierarchy Browser

The hierarchy browser of the GUI displays the hierarchy tree. If we click the submenu *Generate Database* of the menu *Database*, the files contained in this hierarchy tree will be parsed.

For each item in the visualization of the hierarchy tree, individual pop-up menus and mouse click events can be easily configured in the file, which is associated to the alias

`visur_browser`. Clicking on an item of the tree executes diverse actions, or shows a pop-up menu, from which we can choose further commands. In the default configuration, most menu entries call the methods of the libraries RAR or VISUR. The pop-up menus of the hierarchy browser offers the possibility to generate diverse graphs, which are restricted to a sub-hierarchy. The pop-up menus and mouse click events are depending on the type of an item, e.g., double clicking on a filename opens the file in the syntax highlighting SWI-PROLOG editor emacs in an extra window.

The XPCE address of the hierarchy browser is stored in the global SCAV variable `visur_browser`. We retrieve the value of this variable calling

```
sca_variable_get(visur_browser, -Browser_Address).
```

A new tree can be loaded using the shortcut *Load hierarchy* in the symbol bar or the corresponding submenu of the menu *Browser*.

7.1.3 The Visualization Picture

Each graph will be visualized in a new picture. The graphs are generated in GXL format. We call `gxl_to_picture/3`, in order to visualize a graph. Each node or edge of the picture has individual pop-up menus and mouse click events. We can define a pop-up menu for the picture, too. It appears, if we click with the right mouse button on an empty place in the picture. In order to reconfigure the pop-up menus or mouse click events, we have to edit the file, which is associated to the alias `visur_gxl_config`.

Using the GUI

The GUI is adapted to analyze PROLOG source code, but the menus and executed methods can be changed by editing the corresponding XML document, which configurates the pull-down and pop-up menus. Now, we describe the default menus for analyzing PROLOG source code.

In order to start the GUI and to reload the last used project, we call

```
visur_gui.
```

We can either use the reloaded project, we can load another saved project, or we can create a new project (menu *File*, cf. Figure 7.4). To create a new project, we click *File→New Project*.

We need to load a hierarchy containing the location of all sources which we want to examine. To load a hierarchy into the hierarchy browser, we go into the menu *Browser* (cf. Figure 7.5). There, we can either create and store a new source location using *New Source Location...*, or we can reload a source location. The stored source locations are divided into the scopes DDK, PROLOG, JAML, PHP. The menu *Browser* offers the possibility to load an XML file into the hierarchy browser, too. This XML file has to be build up as described in Subsection 3.1.2. We can import the directory structure of the files system, too, using the menu *Select Directory...*.

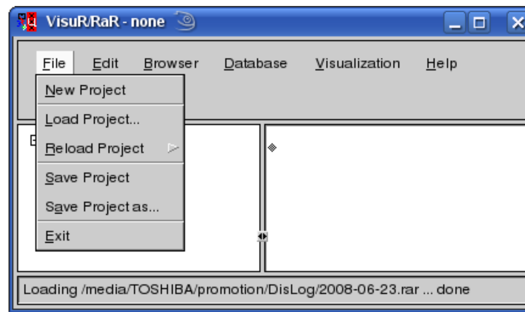


Figure 7.4: Menu File

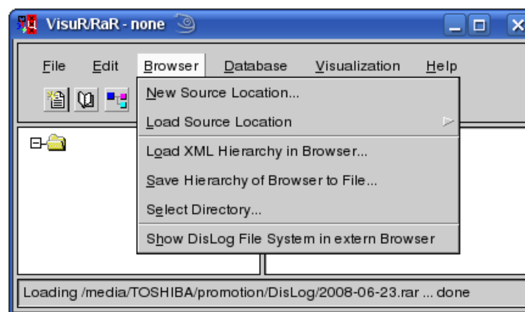


Figure 7.5: Menu Browser

After we have selected the sources, we have to call the menu *Database→Generate ProSoRe Database* (cf. Figure 7.6). This parses the PROLOG files into the PROSORE database. If a file has been modified, we update the PROSORE database by calling *Database→Update ProSoRe Database*.

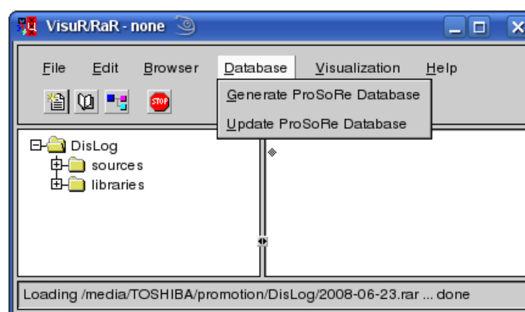


Figure 7.6: Menu Database

Now, we can choose one of the tables or graphs offered in the menu *Visualization* (cf. Figure 7.7).

The methods of the menu *Visualization* concern the whole sources. To restrict methods

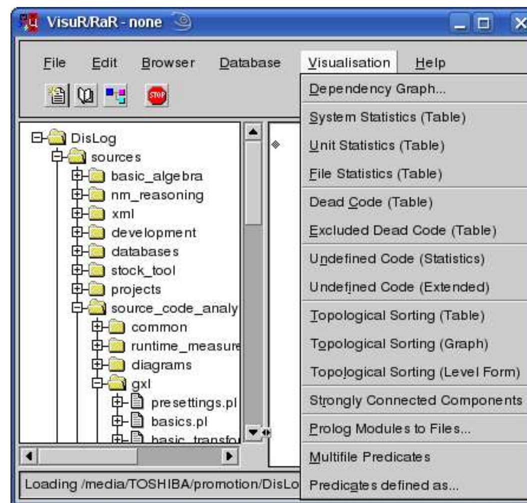


Figure 7.7: Menu Visualization

to a certain branch, we can choose an item of the hierarchy tree and click the right mouse button. A pop-up menu opens (cf. Figure 7.8) and we can choose the corresponding method.

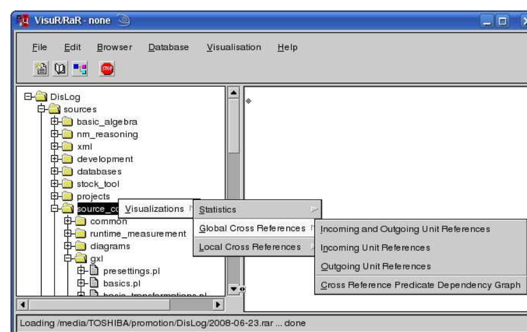


Figure 7.8: Pop-up Menu of the Hierarchy Browser

7.1.4 Plug-ins – Extending the GUI

The features of the GUI are determined by XML documents. They can be extended – comparable to plug-ins of other applications – just by adding further entries to the corresponding XML document. We can add or delete menu entries to the menu bar, the browser, the nodes and edges, and the picture. The default configuration of the GUI is configured for refactoring and visualizing PROLOG source code and for retrieving statistically information about the source code. The configuration is adapted to DISLOG.

To locate XML documents, the corresponding files are associated to certain aliases. These associations can be changed in order to use another configuration. Following aliases are used by the GUI of VISUR:

`visur_menu_bar`
: configures the pull-down menus of the menu bar. How to configure the menu bar in detail is described in Appendix A.3.

`visur_browser`
: configures the integrated hierarchy browser of VISUR. Here, we define the pop-up menus, mouse click events and node symbols of the hierarchy browser (cf. Appendix A.4).

`visur_gxl_config`
: contains the configuration of the GXL graph visualizations and defines beside other parameters, the pop-up menus and mouse click events of the nodes, the edges and the picture of a graph (cf. Appendix B).

Example 7.1 (Configuration of the Menu Bar) The following part of an XML document configures the menu *File* of VISUR. The first submenu is *New Project*, which calls the predicate `new_project/1` in PROLOG. After this menu item, there will be drawn a line (`end_group="on"`). Further menus and calls can be added by adding further `menu` elements.

```
<menus name="Visur Menus">
  <menu name="File">
    <menu name="New Project"
      predicate="new_project"
      prmtrs="[@1]" message="prolog" end_group="on"/>
  ...
  </menu>
...
</menus>
```

7.2 Case Studies

VISUR can be used in a wide area of source code visualization and reasoning. Having a suitable XML representation of the source code, e.g., PROLOGML, JAML [12] or PHPML [14], and having adapted basic methods, we apply methods for visualization, statistics, code reasoning, and refactoring. In Subsection 7.2.1, we generate cross reference graphs between different packages. In order to visualize dependency graphs containing meta calls, we defined the extended Rule/Goal graph. Figure 7.9 shows, e.g., the meta-call predicate *findall/3*, which is called by `calls_uu/2` and by `calls_uu_reduce/3`. Predicates belonging to the group of meta-call predicates are able to call other predicates in their arguments. The predicate-calls of the meta-call predicates are more interesting

than the predicates, which are called in the rule of a meta-call predicate, so we extended the rule/goal graph in order to include such calls. Subsection 7.2.2 shows a visualization

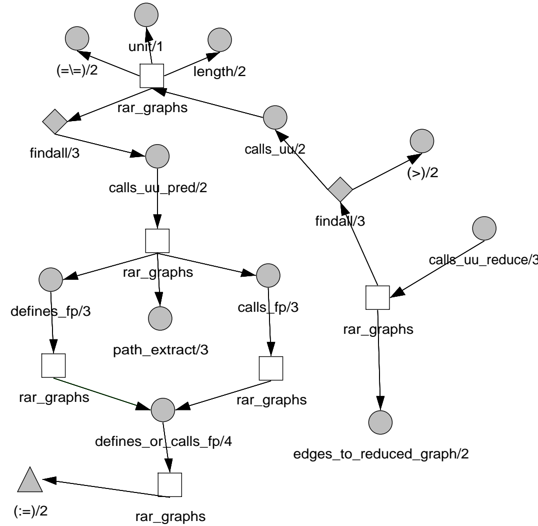


Figure 7.9: Extended Rule/Goal Graph in VISUR

of rule based knowledge represented by XML documents. Finally, in Subsection 7.2.3, we visualize ER diagrams, a DTD in the hierarchy browser and we show how to visualize HTML references.

7.2.1 Cross Reference Graphs for Packages

In this section, we describe the method, which creates graphs. We can create cross reference graphs, and rule/goal graphs. Often, we have to limit the source code, which we want to visualize, in order to create a well arranged graph. Otherwise, the graph would contain too much nodes and edges, so that it would not anymore be informative. In order to get informative graphs and to minimize the amount of nodes and edges in the graph, we restrict the whole source code to the part of the interesting source code, which we then visualize.

In order not to create too large graphs, we use on the one hand a tree in order to choose branches of interest. Each branch of the tree contains a part of the source code. Usually, the source code is hierarchically ordered, which complies with the branches of the tree. The source code of a branch normally belongs to the same programming scope.

On the other hand, we can determine the granularity of the visualization. After we have chosen a branch of the source code tree, we decide, between which packages we want to visualize the dependencies. For the DDK, we have, e.g., the possibility to visualize the dependencies between *units*, *modules*, *files*, or *predicates*. Thereby, a unit contains several modules, a module contains several files, and a file contains several predicates. Figure 7.10 shows the cross references between the units of DISLOG. In order to organize

units or modules hierarchically, we can query the responsible predicates, which causes a call from one module to another module. Knowing these predicates, we can consider moving the predicates into another module.

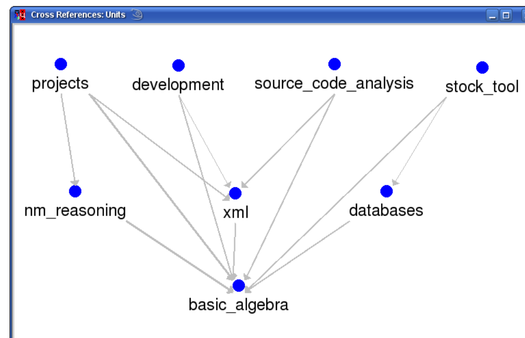


Figure 7.10: Unit Dependency Graph of DISLOG in VISUR

In order to create a graph, we call

```
package_to_gxl(+Config, Predicate_Groups, +Tree,
               +Type, +Package, -Gxl),
```

or just

```
package_to_gxl(+Type, +Package, -Gxl).
```

Depending on the value of the passed arguments, this creates a cross reference graph, a rule/goal graph or an inverse rule/goal graph of the actual parsed system. This method works closely together with the methods of the PROSORE database, in order to generate a visualization of the dependencies between packages or predicates. This means that the corresponding source code has to be parsed and asserted as facts, before calling one of the graph creation methods.

Using the method `create_graph/3` just calls the method `create_graph/6`. The missing parameters

```
Config, Predicate_Groups, Tree
```

are determined before as default values. We specify the graph and the appearance of the graph with the parameters, described in the following. After we created the GXL graph `Gxl`, we visualize the graph using `gxl_to_picture/3` (cf. Section 6.4.1).

The Configuration of the Node Symbols We use the XML document `Config` in order to define the appearance of the node symbols. Thereby, the nodes are divided into different groups. For each group, we define the symbol, the color and the size of the nodes, belonging to the group.

The predicate groups are defined in the configuration file, which is associated to the alias `predicate_groups`. There, we define which predicate, respectively method, belongs to which group. Additionally, we define some further visualization settings in this XML document, too. E.g., we define which nodes are duplicated or hidden in the visualization of a GXL graph (cf. Section 3.2.6). The DTD of the XML document as like follows:

DTD:

```
<!ELEMENT config node_symbols>

<!ELEMENT node_symbols node_symbol+>

<!ELEMENT node_symbol EMPTY>
<!ATTLIST node_symbol
   group type CDATA #required
   class type CDATA #required
   symbol (rhombus|triangle|
      circle|box|honeycomb|
      no_symbol|text_in_box|
      text_in_ellipse|CDATA) #required
   size (small|medium|large|CDATA) #required
   color (yellow|blue|red|green|
      black|white|orange|...) #required>
```

We saved an XML document containing a default configuration for the method mentioned above and we associated this file to the alias `visur_config`. We can load the content of this file easily into a PROLOG variable using `alias_to_fn_triple/2`.

Example 7.2 (Configuration of create_graph/6) The following XML document contains a fragment of the configuration `Config` of the method `create_graph/6`.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<config>
  <node_symbols>
    <node_symbol group="rule_node" class="rule_node"
      symbol="box" size="medium" color="white"/>
    <node_symbol group="built_in" class="dislog"
      symbol="triangle" size="medium" color="yellow"/>
  ...
  </node_symbols>
</config>
```

The element `<node_symbols>` contains the visualization settings for the predicate groups. The element contains several `<node_symbol>` elements. The attributes of the `<node_symbol>` element contain the settings for the visualization.

`group`, `class`
These two attributes are references. They are used to configure the node symbol of

the predicates belonging to the predicate group with the same attributes `group` and `class` of the XML document `Predicate_Groups` (cf. Section 3.2.6).

`symbol`
determines the node symbol, which is shown in a visualization. Possible values are:

```
no_symbol, text_in_box, text_in_ellipse,
rhombus, triangle, circle, box, honeycomb.
```

`size`
determines the size of the node symbol. Possible values are

```
small, medium, large,
```

or all other size aliases, which are defined in the XML configuration document of the visualization method `gxl_to_picture/3`.

`color`
determines the color of the node symbol. Possible values are, e.g.,

```
green, red, yellow, blue, ...
```

It can be either an alias name or a RGB value in hexadecimal format. RGB values have the format `colour(HEX)`, e.g., `colour('#FF0000')` for red.

The Predicate Groups are defined in the XML document `Predicate_Groups`. This document is already described in Section 3.2.6.

The Tree reflects the hierarchical structure of the examined source code. In the DDK, this matches with the file system structure. Each branch of the tree is equal to a directory of the file system of the DDK. Thereby, the top level, respectively the root of the tree, is called `system`. Further levels are `sources`, `libraries`, `units`, `modules`, and `files`. Often, we just choose a part of the source code to visualize in a graph. These parts are restricted to the source code, contained in a branch or this tree, respectively, a subdirectory containing the source code.

The Parameter Type selects the graph, which we want to create. Using the PROSORE database and the corresponding RAR methods, we retrieve the dependencies between predicates, files and in general, packages. Thereby, the considered source code is limited to the selected branch of the tree.

`incoming`
creates a graph, containing the incoming cross references between the chosen package `Package` and all other packages of the source code. Thereby,

```
Package = Level:Path
```

The granularity of the visualization is the same as `Level`. E.g., if `Level` is `module`, we visualize all modules, which contain a method calling a method of the package `Package`.

Example 7.3 (Graph Creation) In order to show all modules of the DDK, which call the module `gxl` (cf. Figure 7.11), we call

```
?- Package = module:'sources/source_code_analysis/gxl',
   visur:create_graph(incoming, Package, Gxl),
   gxl_to_picture(_, Gxl, _).

Yes
```

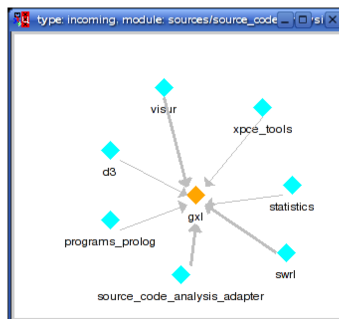


Figure 7.11: Incoming Calls of the Module Gxl

`outgoing`
analogous above, this creates a graph, containing the outgoing cross references between the chosen package `Package` and all other packages of the source code.

Example 7.4 (Graph Creation) In order to show all modules of the DDK, which are called from the module `gxl` (cf. Figure 7.12), we call

```
?- Package = module:'sources/source_code_analysis/gxl',
   visur:create_graph(outgoing, Package, Gxl),
   gxl_to_picture(_, Gxl, _).

Yes
```

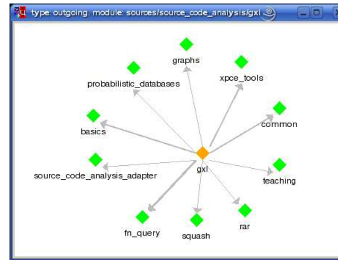


Figure 7.12: Outgoing Calls from the Module

`in_out`
: This combines the first two graphs mentioned above. This means we generate a graph which contains all calls from outside to the current package and all calls from the current package to the outside packages.

Example 7.5 (Graph Creation) Figure 7.13 shows all calls between the module `gxl` and the other modules of the DDK.

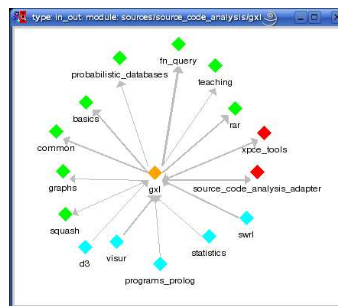


Figure 7.13: Incoming and Outgoing Calls of the Module

The three graphs mentioned above visualize cross references of the whole available source code *from* and *to* the package `Package = Level:Name`. Thereby, the granularity is always the same as it is specified with `Level`. The next values of `Type` create graphs, which are limited to the source code contained in the package `Level` and `Path`. Calls outside this package are ignored.

`file`
: visualizes the cross references of all *files* of the package `Package`.

Example 7.6 (Graph Creation) In order to show the file dependency graph of all files in the module `gxl` (cf. Figure 7.14), we call

```
?- Package = module:'sources/source_code_analysis/gxl',
   visur:create_graph(file, Package, Gxl),
   gxl_to_picture(_, Gxl, _).

Yes
```

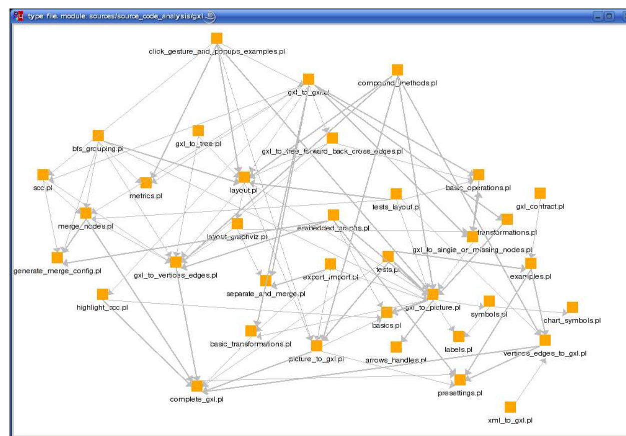


Figure 7.14: File Dependency Graph of the Module GXL

`module`
: visualizes the cross references of all *modules* of the package `Package`.

`unit`
: visualizes the cross references of all *units* of the package `Package`.

`rule_goal`
: creates a *rule/goal* graph of the predicates contained in the package `Package`.

`goal`
: creates a *predicate dependency* graph of the predicates contained in the package `Package`.

Package contains the part of the system, respectively branch of the tree, on which we want to look at. The package is divided into the level and the path.

```
Package = Level:Path
```

The level and the path have to be contained in the hierarchy, which we have passed, or it has to be a head of a rule in the format

```
predicate:(Module:Predicate)/Arity.
```

Using the DDK, `Path` contains the path beginning from the root `~/Dislog/` of the tree.

Example 7.7 (Package) Possible values of `Package` of the DDK are, e.g.,

```
unit:'sources/source_code_analysis',
module:'sources/source_code_analysis/gxl',
file:'sources/source_code_analysis/visur/graphs.pl'
predicate:'(visur:rule_goal_graph)/5'.
```

The Result: GXL The method `create_graph/6` creates a graph, containing the wished dependencies of the examined part of the source code. The output is a graph which is written in GXL. The method `create_graph/6` uses the PROSORE database and the methods of the PROSORE database in order to create graph.

Example 7.8 (Graph Creation) This example shows the source code of a method, which parses an arbitrary PROLOG file and creates either a rule/goal graph, or a goal graph, depending on the value of `Type`.

Implementation:

```
1  prolog_file_to_dependency_graph(Type, File_1) :-
2     memberchk(Type, [rule_goal, goal]),
3     tilde_to_home_path(File_1, File_2),
4     rar:parse_file_to_rar_database(File_2),
5     visur:create_graph(Type, file:File_2, Gxl),
6     gxl_to_picture(_, Gxl, _).
```

First, we check, if the argument `Type` is either `rule_goal` or `goal` (line [2]). Then, if the *Unix* shortcut tilde ~ occurs in the filename, we replace it with the home path of the current user (line [3]). After this, we parse the PROLOG source code file and we add the content to the PROSORE database (line [4]). At last, we create the graph and visualize the graph in an XPCE picture (lines [5, 6]).

We call

```
?- prolog_file_to_dependency_graph(rule_goal, 'basics.pl').

Yes
```

in order to view the rule/goal graph of the file `basics.pl`.

Example 7.9 (Graph Creation) Having the method above, we can extend it with some further GXL actions, using the method `gxl_to_gxl/3`. In the following example, we write a method which additional deletes loops and single nodes. The single nodes are visualized in an additional table. At last, we add a weighting method and a layouting method to the method. The weighting method changes the thickness of the edges in dependency of the weight of an edge. The layouter uses a bfs algorithm to position the nodes.

Implementation:

```
create_graph_visualize_graph(Type, Package) :-
   alias_to_fn_triple(visur_config, Conf),
   alias_to_fn_triple(predicate_groups, PG),
   sca_variable_get(source_tree, Tree),
   create_graph(Conf, PG, Tree, Type, Package, Gxl_1),

   Actions_1 = actions:[]:[
      action:[name:delete_loops]:[] ],
   Actions_2 = actions:[]:[
      action:[name:delete_single_nodes]:[] ],
   Actions_3 = actions:[]:[
      action:[name:layout]:[
         gxl_layout:[mode:bfs, x_start:10, y_start:10,
            x_step:50, y_step:50, y_variance:10]:[]]],

   gxl_to_gxl(Actions_1, Gxl_1, Gxl_2),
   gxl_to_gxl(Actions_2, Gxl_2, Single_Nodes, Gxl_3),
   gxl_to_gxl(Actions_3, Gxl_3, Gxl_4),
   gxl_edge_weight(Gxl_4, Gxl_5),
   gxl_to_picture(_, Gxl_5, _),
   singletons_to_table('Single Nodes', Single_Nodes).
```

We assume that the source code already is contained in the PROSORE Database. This has to be done before, using the parsing method described in Subsection 3.2.2.

We present statistical information about calls between different modules or units in tables, too. Figure 7.15 shows the number of calls from one module to another module.

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 7.15: Calls across Modules

7.2.2 Visualizing Rule Based Knowledge

[54] introduces a declarative approach for querying and visualizing rule based knowledge represented by XML documents. The extension and maintenance of large rule-based systems is a complex task. For instance, the deletion of a redundant or an incorrect rule is often very difficult to perform, since (transitive) dependencies of this rule are not obvious at first sight. The visualization of knowledge can be used, e.g., in the following scenarios:

Restructuring knowledge: If the expert wants to remove or modify an existing rule, then it is helpful to inspect all depended knowledge objects (e.g., constrained findings and inferred solution objects) and rules, respectively. A visual view of the dependencies can simplify the inspection of the knowledge base.

Validating knowledge: The visual inspection of knowledge can be also helpful during the validation of the knowledge systems' reasoning behavior. Then, the visualization of the (transitive) derivation graph of a solution object defined by its deriving rules can assist the expert during a debugging session.

Examination of knowledge design: The rule base is visualized as a graph with knowledge objects (i.e., findings, solutions) represented as nodes and rules depicted by corresponding edges. Then, the design of the knowledge base can be simply analyzed by viewing the graph structure. Results of this analysis are domain dependent. E.g., a sub-graph connected to the remaining graph structure only by one node is an indicator for vulnerable knowledge design, since a part of the implemented knowledge depends on a single object.

Besides the examples given above there exist many other applications for the visualization of rule bases. Figure 7.16 shows a transitive derivation tree using VISUR. This example, shows, e.g., that the diagnosis P181 is directly derived by two rules (i.e., Rfb1831, Rfb1822).

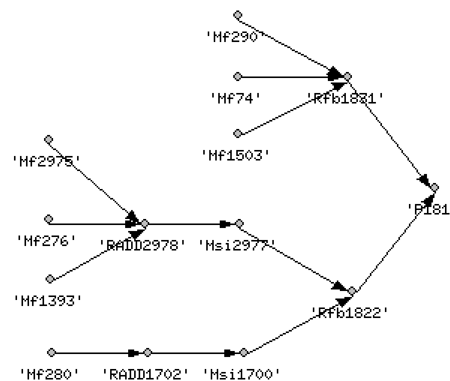


Figure 7.16: Transitive Derivation Tree of the Diagnosis *P181*.

7.2.3 Visualizing Other Graphs

Further examples of visualizations are, e.g., ER Diagrams, DTDs and HTML documents. In the following, we give some ideas how to visualize the dependencies of such document types.

ER Diagrams

In [20], we have shown how to visualize ER diagrams (cf. Figure 7.17). Now, the ER visualization method has been adapted to GXL. Thereby, we changed the data input format of an ER diagram to XML. It looks like the following example.

Example 7.10 (File "er_diagram.xml")

```
<er_diagram>
  <entity name="Lieferant">
    <attributes>
      <attribute name="Lieferantennr" description="VARCHAR(8)"/>
      ...
    </attributes>
    <primary_key>
      <key name="Lieferantennr"/>
    </primary_key>
  </entity>

  <relationship name="Bestellung_von_Einkaufswaren">
    <entities>
      <entity name="Lieferant" cardinality="n"/>
      ...
    </entities>
    <attributes>
      <attribute name="Lieferantennr" description="CHAR(9)"/>
      ...
    </attributes>
  </relationship>
  ...
</er_diagram>
```

The DTD of the XML document looks as follows:

DTD:

```
<!ELEMENT er_diagram entity* relationship*>

<!ELEMENT entity attributes primary_key>
<!ATTLIST entity
   name type CDATA #required>

<!ELEMENT attributes attribute+>

<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
   name type CDATA #required
   description type CDATA #required>

<!ELEMENT primary_key key+>
```

```
<!ELEMENT key EMPTY>
<!ATTLIST key
   name type CDATA #required>

<!ELEMENT relationship entities attributes>
<!ATTLIST relationship
   name type CDATA #required>

<!ELEMENT entities entity+>

<!ELEMENT entity EMPTY>
<!ATTLIST entity
   name type CDATA #required
   cardinality type CDATA #required>
```

Calling the method

```
er_diagram_to_gxl(+Config, +File, -Gxl)
```

consults the file `File`, which contains the description of the ER diagram. We transform it into GXL using the parameters defined in the configuration `config`, which define the symbols and size of entities, attributes, and relationships. Now, having a GXL document, we visualize the ER diagram, using the call `gxl_to_picture/3`.

Figure 7.17: Visualization of an ER Diagram by VISUR

DTDs

We visualize DTDs in a graph or a hierarchy browser. In the hierarchy browser, recursive elements of the DTD are marked with a special symbol. Figure 7.18 shows a part of a DTD in the hierarchy browser and Figure 7.19 shows a part of a DTD as a graph. We call

```
visur:dtd_to_tree(+File, -Tree)
```

in order to create a suitable tree `Tree` of a DTD file `File`, which we can visualize with the hierarchy browser. We call

```
visur:dtd_to_browser(+File), or
visur:dtd_to_browser(+File, -Frame)
```

in order to visualize a DTD direct in the DTD browser. We call

```
visur:dtd_to_gxl(+File, -Gxl)
```

to create a graph of a DTD, which we can visualize using `gxl_to_picture/3`.

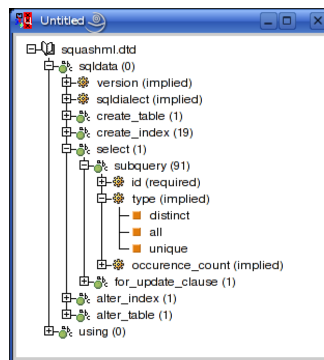


Figure 7.18: A DTD visualized in the Hierarchy Browser

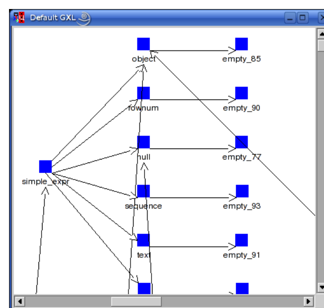


Figure 7.19: A DTD visualized in an XPCE Picture

Dependencies of HTML References

To visualize HTML references, we have to replace the basic predicate `calls/2` of the PROSORE. We do not need a local database of facts storing the dependencies in an XML representation. We can use the world wide web as database. Calling

```
calls(+Url_1, ?Url_2),
```

the following steps have to be done:

- the HTML document of the URL `Url_1` has to be downloaded
- the HTML document has to be parsed for tags, such as

  ```
  <a href="...">...</a>
  ```

  or equivalent tags.
- the result has to be bound to the variable `Url_2`.

As there do not exist symmetric links in the world wide web (we do not know, which document is linked by another document), we do not need the reverse database which uses the facts `is_called_by/6` of the PROSORE.

Additionally, we can check if the linked websites exist or not and mark not existing websites in the visualization. To avoid endless searches of websites, which are transitively linked from one site, the implementation of the transitive closure should be restricted to a maximal depth of linked websites. This can be done by an additional parameter, which stops `calls/2` after a certain number of iterations.

The hierarchy browser can be used to list a number of URLs, whose content can be visualized in an additional window.

Part IV

Conclusions

This thesis deals with the management and analysis of source code, which is represented in XML. Using the elementary methods of the XML repository, we access, change, update, and save the XML source code representation. We reason about the source code, refactor source code and we visualize dependency graphs for call analysis. The visualizations and source code analyses are helpful for refactoring. We visualize dependencies between files, modules, or packages. Using source code visualizations and call spectra, we try to structure the source code in order to get a system, which is easily to comprehend, to modify and to complete. We have developed sophisticated methods to slice the source code in order to obtain a working package of a large system, containing only a specific functionality.

The basic methods, on which the visualizations and analyses are built on can be changed like changing a plug-in. The advantage is, that we can reuse the visualization methods in order to handle arbitrary source code representations, e.g., JAML, PHPML, PROLOGML. Dependencies of other context can be visualized, too, e.g., ER diagrams, or website references.

The tool SCAV supports source code visualization and analyzing methods. The design of SCAV is built up in several layers. Adapting the first layer which contains the source code representation in an XML repository and the basic query methods, we can use SCAV to visualize dependencies of arbitrary coherences. The plug-in ability makes SCAV to an easily and widely usable tool. Using SCAV, we calculate statistics which we present in tables, and graphs. The library GXL for PROLOG and the picture class for GXL graphs is used to represent graphs. The library GXL enables us to execute diverse graph operations, e.g., we can classify, highlight or delete nodes and edges, we can save and reload graphs or transform graphs into different representations. Using the picture class, we can visualize GXL graphs. Thereby, we are able to configure diverse pop-up menus and mouse click events. The SCAV GUI supports nearly all available methods. This GUI can be configured individually. It can be easily extended by suitable plug-ins.

SCAV can be combined with further tools. We can join the micro analyses tools of Dian Dochev and Michael Müller with the macro analyses of SCAV. Moreover, we can extend SCAV with further reasonings on the XML structure of the repository.

Part V

Appendix

Appendix A

XPCE Programming Techniques

Using SWI-PROLOG, it is complex to define mouse click events and pull-down or pop-up menus. We have implemented a possibility to define mouse click events, pull-down and pop-up menus, which can be easily configured by just editing an XML document. Therefore, we have to apply methods to XPCE objects. Appendix A.1 describes how to apply a method to an XPCE object, in Appendix A.2 we describe how to generate mouse click events and in Appendix A.3, we describe how to generate pull-down and pop-up menus.

A tool using the XML techiques for defining pop-up menus is the hierarchy browser. We use the hierarchy browser to visualize hierarchies of the source code. We configure the hierarchy browser using XML documents. Menus can be changed easily by changing the corresponding XML document. The hierarchy browser is explained in Appendix A.4.

We use XML documents to configure methods. It would be tedious to specify the location of an XML document each time we need the document. Therefore, we use an alias-file association in order to access files in a fast way without specifying the location of the file. We describe, how we associate files with aliases in Appendix A.5. The alias browser is an application of the hierarchy browser and is used to comfortably administrate the aliases.

A.1 Applying Methods to Objects of a Picture

We can apply a method to XPCE objects. This is needed, e.g., in order to get the new node positions of a graph, which is modified interactively by the user. E.g., the method `picture_to_gxl/3` (cf. page 195) generates a new GXL document of a graph visualized in an XPCE picture. This method retrieves from each node object of the XPCE picture the coordinates, and saves them into the GXL document. The following lines of source code show how to apply a method to node or edge objects of a picture.

Implementation:

```
apply_method_to_objects_in_pic(Type, Pic, Method) :-
   member(Type, [device, connection]),
   send(Pic?graphicals, for_all,
      if(message(@arg1, instance_of, Type),
         message(@prolog, Method, @arg1))).
```

`Type`
: is either `device` or `connection`. If we want to apply the method on a box, rhombus, circle or in general a *compound object*, we use `device`. If we want to apply the method on an edge, a line or in general a *connection* between two node, we use `connection`.

`Pic`
: is the XPCE address of the picture and `Method` is the method, which we want to apply to the objects, e.g., `writeln/1`. `@arg1` contains the XPCE address of the object, e.g., a node or an edge.

`Method`
: is the method, which will be executed, e.g., `writeln/1` in order to write the XPCE address of the node or an edge.

If we have the XPCE address of an object in the picture, we can receive the properties of this object using the predicate `get/3` (cf. page 201). Further information about `get/3` can be obtained in the XPCE manual [63].

Example A.1 (Writing XML Payload)

```
write_xml_payload(Address) :-
   get(Address, xml, Xml),
   fn_to_xml(Xml).

write_payload_of_nodes_and_edges_in_pic(Pic) :-
   send(Pic?graphicals, for_all,
      if(message(@arg1, instance_of, device),
         message(@prolog, write_xml_payload, @arg1))),
   send(Pic?graphicals, for_all,
      if(message(@arg1, instance_of, connection),
         message(@prolog, write_xml_payload, @arg1))).
```

In this example, we search for each XPCE address contained in the picture. If the corresponding class of an XPCE object belongs to `device`, which is in our visualizations equal to a node, we call the method `write_xml_payload/1`, together with the argument `@arg1`. The argument `@arg1` contains the XPCE address of the object.

The method `write_xml_payload/1` queries for the the XML payload of the object and pretty prints the XML payload using `fn_to_xml/1` to the PROLOG console. We do the same with the class `connection`, which represents the edges in a graph picture.

A.2 Generating Mouse Click Events

We have implemented an easy method to define mouse click events for graphical XPCE objects. We just call the following method together with the XPCE address of the object, a list of variables, and an XML configuration document in order to make an XPCE object accessible for mouse clicks events. Clicking on such a graphical XPCE object, the corresponding method, which we defined for a certain event in the XML configuration document, will be executed. The method

```
xml_click_action_to_xpce(+Address, +Variables, +XML)
```

adds the mouse click events, defined in the XML document `XML`, to the XPCE object, which belongs to the XPCE address `Address`.

`Address`
: is the XPCE address of an existing object to which we want to add one or more mouse click events with corresponding methods. The mouse click events and methods are defined in the XML document `XML`.

`Variables`
: is a list of variables, which are needed by a method, which is called after a mouse click event. These variables are usually only available at runtime. It may be a dynamic XPCE address of a frame, a browser window, a node, or a menu item. The process flow looks as follows: first a mouse click event is triggered. Then the corresponding method, which is defined in the XML document `XML` is called. The XML document contains a list of parameters the method needs. This can be a fixed value or it can be a dynamic variable, which is only available at runtime. In the XML document `XML`, we reference dynamic variables contained in the list `Variables` by using the attribute `prmtrs` and the shortcuts `@1`, `@2`, `@3` .... Thereby, the shortcut `@1` in the XML document references the first variable of the list `Variables`, `@2` references the second variable, `@3` references the third variable and so on.

`XML`
: contains the definition of the mouse click events, the corresponding called methods and the necessary parameters.

The DTD of the XML document `XML`, looks as follows:

DTD:

```
<!ELEMENT on_click EMPTY>
<!ATTLIST on_click
   button (left|middle|right) #required
   type (single|double|triple) #required
   module CDATA "user"
   predicate CDATA #required
   prmtrs CDATA #required
   message (prolog|class) "prolog">
```

`button`
: This attribute defines the mouse button. Valid values are `left`, `middle` and `right`.

`type`
: This attribute defines, if the button has to be clicked `single`, `double`, or `triple` times, in order to trigger a certain mouse click event.

`module`
: We can call predicates of PROLOG modules, too. This attribute determines the PROLOG module of the called predicate. The default value of the module is `user`.

`predicate`
: determines the predicate, which is called after we triggered a mouse click event.

`prmtrs`
: The value of the attribute `prmtrs` is a list, which contains the fixed arguments of the predicate `predicate` and the references to the dynamic variables of the list `Variables`, which are only available at runtime. If we want to call a dynamic argument, we use the aliases `@1`, `@2`, ..., e.g., if `prmtrs` has the value `[@1]`, the first element of the list `Variables`, which is passed by creating the mouse click events using `xml_click_action_to_xpce/3`, is chosen. In the list `prmtrs`, only atomic elements are allowed. The list may not contain unbound elements.

`message`
: determines the namespace, in which the called predicate is executed. `message` is either `prolog` or it is `class`. The default value of `message` is `prolog`. If `message` has the value `prolog`, the call is executed in the normal namespace of PROLOG.

  If `message` has the value `class`, the call is executed in the namespace of the class, which is given in the first argument of the list `Prmtrs`. This means that it must be a call of a method contained in this class. Using the value `class`, there must always exist at least one argument in the list `Variables` with the XPCE address of the class, which is referenced in the attribute `prmtrs`.

Example A.2 (Mouse Click Events)

```
?- new(Picture, picture),
   XML_1 = on_click:[button:left, type:double,
       module:prolog, predicate:emacs, prmtrs:[@1],
       message:prolog]:[],
   XML_2 = on_click:[button:right, type:single,
       module:prolog, predicate:writeln, prmtrs:[@2],
       message:prolog]:[],
   Variables = ['file.txt', 'Hello, again!'],
```

```
    xml_click_action_to_xpce(Picture, Variables, XML_1),
    xml_click_action_to_xpce(Picture, Variables, XML_2),
    send(Picture, open).

Yes
```

First, we instantiate a new picture `Picture`. Then, we define two mouse click events: in the argument `XML_1`, we define that after double clicking the left mouse button on the picture, the internal PROLOG editor emacs is called with the first argument of the list `Variables`. In the argument `XML_2`, we define that after clicking the right mouse button once on the picture, `writeln` is called with the second argument of the list `Variables`. The text `Hello, again!` is written to the PROLOG console. Analogous examples for pop-up menus follow in Appendix A.3.

Remark: In VISUR, we generate the mouse click events for the picture, the nodes and the edges passing the following list of runtime variables to the corresponding XPCE object:

```
Variables = [Picture, Object, Button, Type, XML_Code]
```

`Picture` is the address of the visualizing picture, `Object` is either the address of the picture, or the address of the node, or the address of the edge. `Button` and `Type` specifies the mouse click event and `XML_Code` contains the corresponding XML part of the node or picture.

This means, that using a method of visur, we can define in an XML document, that one or more of the runtime variables above are passed to a method after a click gesture is triggered. In the XML document, we have to use the shortcuts `@1`, `@2`, `@3`, `@4`, and `@5` in the list `prmtrs`.

A.3 Generating Pull-down and Pop-up Menus

Analogous to the mouse click events, we have implemented an easy method to define pull-down and pop-up menus. These menus can be appended to any graphical XPCE object. We define the menus in an XML document. This XML document contains the methods, which we want to be executed, too. Doing so, we have the advantage that we can very easily change menus and methods, even during runtime.

First of all, we have to create a container for the pop-up menus. We have the choice between two types of containers. We have either the possibility to create a pull-down menu for a menu bar, which we can use in an application. Or, we can create a pop-up menu, which we can append to a graphical XPCE object, e.g., a node or an edge.

Depending on how we want to use the menus, we generate an instance of one of the following XPCE classes as menu container. If we want to create a container for a pop-up menu, in order to use the menu for a graphical XPCE object, we call

```
new(C, popup).
```

If we want to create a menu bar for an application, we call

```
new(C, menu_bar).
```

Having a proper container instance, our method

```
xml_menu_to_xpce(+C, +Variables, +XML)
```

appends the menu entries defined in the XML document `XML` to the container `C`. Now, we can append this pop-up to any graphical XPCE object or we can use it for a menu bar. The examples below show how to create a menu and how to append the pop-up to a XPCE object. But first of all, we explain the arguments of the method above. These arguments are equivalent to the arguments of the method, which generates mouse click events.

`Popup`
: is the XPCE address of the instance of a pop-up-class or a menu-bar-class.

`Variables`
: is equivalent to `Variables` of the *mouse click event* method in Appendix A.2. It is a list of variables, which is needed by the called predicate, and which are only available at runtime. This can be, e.g., an XPCE address of a frame, a browser window, a node, or a menu item. We refer to these variables using the attribute `prmtrs` in the XML document `XML`. There, we determine, which variable of this list is passed to a called predicate. For further details about the list `Variables` cf. Appendix A.2.

`XML`
: contains the menu structure, the corresponding called methods and the necessary arguments.

The DTD of the XML document, which contains the menu structure, looks as follows:

DTD:

```
<!ELEMENT popup menu*>

<!ELEMENT menu menu*>
<!ATTLIST menu
   name CDATA #required
   end_group (on|off) #implied
   module CDATA #implied
   predicate CDATA #implied
   prmtrs CDATA #implied
   message (prolog|class) #implied>
```

The attributes of the element `<menu>` mean the following:

`name`
: determines the visible name of a menu item. This name is displayed, when we open a menu. The name is highlighted, when we move the mouse over the menu name. If the menu name is highlighted, we can click on the menu name. Either the corresponding method is called, or if the menu item contains further submenus, a window for the submenu is opened. This window opens automatically, if we rest on a menu item, too.

`end_group`
: determines, if a separator will be drawn below a menu item. Separators can be used to group menu items, which belong to the same scope.

`module`, `predicate`, `prmtrs`, `message`
: These attributes have the same meaning as described for the mouse click event method in Appendix A.2. We can omit these attributes in menus, which contain further submenus.

Example A.3 (Pull-down and Pop-up Menus) This XML configuration document is used to generate the menu bar shown in Figure A.1:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<popup>
   <menu name="File">
      <menu name="Print to File..."
            module="visur" predicate="picture_to_eps_file"
            prmtrs="[@3]" message="prolog" end_group="on"/>
      <menu name="Exit"
            module="user" predicate="destroy"
            prmtrs="[@1]" message="class" end_group="off"/>
   </menu>
   <menu name="Edit">
     <menu name="Write"
            module="user" predicate="writeln"
            prmtrs="[@2]" message="prolog" end_group="on"/>
   </menu>
</popup>
```

We save this XML document in the file `menu.xml`. In order to use it in SWI-PROLOG, we read the XML document into a PROLOG variable, using `dread/3` (cf. Section 6.2). The XML document is represented in PROLOG using the field notation.

```
?- new(Frame, frame),
   new(Dialog, dialog),
   new(Menu_Bar, menu_bar),
   new(Pic, picture),
   new(Popup, popup),
```

```
    send(Dialog, append, Menu_Bar),
    send(Dialog, append, Pic),
    send(Frame, append, Dialog),
    send(Pic, popup, Popup),

    dread(xml(dtd:[prmtrs:list]:[]), 'menu.xml', XML),
    xml_menu_to_xpce(Menu_Bar, [Frame, 'Hello!', Pic], XML),
    xml_menu_to_xpce(Popup, [Frame, 'Hello!', Pic], XML),

    send(Frame, open).

Yes
```

First, we generate a new frame, a dialog, a menu bar, a picture, and a pop-up. Then we append the menu bar and the picture to the dialog. The dialog is appended to the frame and the pop-up is added to the picture. After we created the proper container class for the menu bar and the pop-up, we add the items to the container. Therefore, we read the XML document using `dread/3`. Due to the fact that the attribute `prmtr` of the xml document contains a PROLOG list, we give in the DTD the corresponding value. At last, we add the menu items to the menu bar and the pop-up and open the frame. Now, we have at the top of the frame a menu bar, and we see a pop-up menu by clicking the left mouse button on the picture.

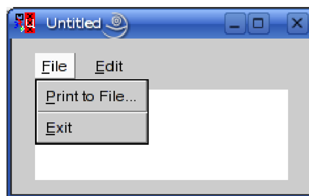


Figure A.1: A Menu Bar Build with xml_menu_to_xpce/3

Note that the variables `Frame` and `Pic` have to be bound before we can pass it in the arguments of `xml_menu_to_xpce/3`.

The generated menu contains the menus *File*, and *Edit*. The menu *File* contains the submenus *Print to File...* and *Exit*. Now, e.g., if we click on the submenu *Print to File...* the method

```
picture_to_eps_file(Picture)
```

will be executed and if we click on the submenu *Exit* the call

```
send(Frame, destroy)
```

will be executed.

Remark: In VISUR, we create the menu bar, pop-up menus for the picture, the nodes and the edges passing the following list of runtime variables to the corresponding XPCE object:

```
Variables = [Picture, XML_Code, Object]
```

`Picture` is the address of the visualizing picture, `Object` is either the address of the picture, the address of the node, or the address of the edge. `XML_Code` contains the corresponding XML part of the node, edge or picture.

This means we can define an XML document in this way, that the dynamic variables mentioned above are be passed to a method. We do this in the list `prmtrs` using the shortcuts `@1`, `@2`, `@3`.

A.4 The Hierarchy Browser

We have implemented a hierarchy browser to visualize trees. The tree has to be passed as XML document to the hierarchy browser. Using the configurable menu bar, pop-up menus, and the mouse click events, we can easily attach methods to the hierarchy browser, which will be executed after a mouse click event has been triggered. Each node of the visualized tree is represented by an item in the hierarchy browser. For each item we can configure two icons (*expanded*, *collapsed*), a certain pop-up, which opens when a right mouse click is done on the item and a method, which is called by a mouse click on the item. We integrated a hierarchy browser in the VISUR GUI in order to visualize the hierarchy of the loaded files (cf. Figure A.2), to choose the part of source code, which we want to visualize, and to execute predefined methods on parts of the chosen source code. But the hierarchy browser can be used separately, too (cf. Figure A.3). An example of a stand alone application, e.g., is the *alias browser*, which we describe in Appendix A.5.

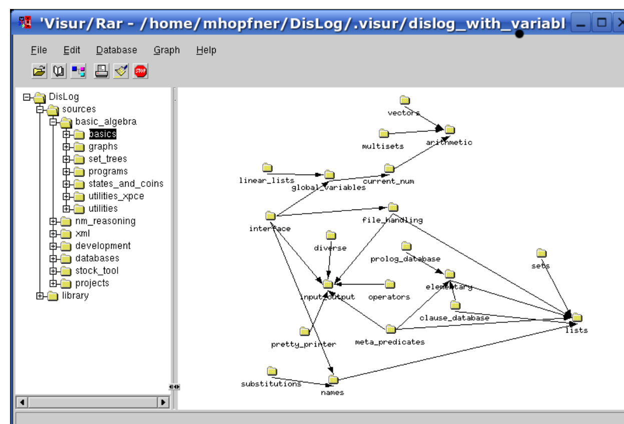


Figure A.2: Hierarchy Browser and File Dependency Graph of the Module basics

In the following, we describe how to create an instance of the hierarchy browser and how to integrate the hierarchy browser to an application. Furthermore, we explain the configuration possibilities of the hierarchy browser, e.g., how to configure pop-ups, menu bars and mouse click events. At last, we describe the DTD of the input trees of the hierarchy browser.

Creating the Hierarchy Browser

To create a new instance of the hierarchy browser or to send a tree to an existing hierarchy browser, we call

```
hierarchy_to_browser_ddk(+Config, +Tree, ?Browser).
hierarchy_to_browser_ddk(+Tree, ?Browser).
```

The argument `Config` configures the hierarchy browser. The configuration is written in an XML document. The argument `Tree` contains the tree, which we want to visualize. The argument `Browser` contains the XPCE address of the hierarchy browser.

It is not necessary to send a configuration to the browser. If we do not send a configuration, the empty configuration `config:[]:[]` will be used, or, if the hierarchy browser was instantiated with a configuration document before, the existing configuration document will be used. This may be, e.g., the case, if we send a new tree to an existing browser, in order to update the visualization.

In some cases, it is not possible to integrate the hierarchy browser into an existing GUI using the predicate `hierarchy_to_browser_ddk`. Instead, we can create an instance of the hierarchy browser, using the following call, too.

```
new(Browser, hierarchy_browser_ddk(+Config, +Tree)).
```

This makes it possible, to append the browser to a XPCE class, e.g., `frame`, `dialog` or `picture`.

Example A.4 (Creating the Hierarchy Browser) We arrange a hierarchy browser left of a picture, and open the picture together with the appended hierarchy browser.

```
?- new(Pic, picture),
   new(Browser,
      hierarchy_browser_ddk(config:[]:[], tree:[]:[])),
   send(Browser, left, Pic),
   send(Pic, open).

Pic = @2653629/picture,
Browser = @2709331/hierarchy_browser_ddk

Yes
```

In order to actualize the visualized tree in the hierarchy browser, we just need the XPCE address of the hierarchy browser. We call

```
?- hierarchy_to_browser_ddk(tree:[...]:[...], Browser).

Yes
```

and the new tree is visualized in the hierarchy browser.

The Configuration of the Hierarchy Browser

We have saved an example of the hierarchy browser configuration in an XML document. This XML document or file, respectively, is associated to an alias. We access and use the file without specifying the path and name of the file each time it is used. The file is associated to the alias `hierarchy_browser_config`. The association can be changed at any time by the user (cf. Appendix A.5). The DTD of the hierarchy browser configuration as like follows:

DTD:

```
<!ELEMENT config icons? nodes? mouse_clicks?>

<!ELEMENT icons icon*>

<!ELEMENT icon EMPTY>
<!ATTLIST icon
   alias CDATA #required
   file CDATA #required>

<!ELEMENT nodes node*>
<!ATTLIST nodes
   open CDATA #implied
   close CDATA #implied>

<!ELEMENT node EMPTY>
<!ATTLIST node
   alias CDATA #required
   open CDATA #required
   close CDATA #required>

<!ELEMENT mouse_clicks mouse_click*>

<!ELEMENT mouse_click on_click* popup*>
<!ATTLIST mouse_click
   alias CDATA #required>
```

The Elements <icons> and <nodes> These two elements are used for configuring the visualization of the items, respectively nodes, in the hierarchy browser. The element `<icons>` contains several `<icon>` elements. Each `<icon>` element contains the attributes `alias` and `file`.

The attribute `file` contains the filename and path of an existing picture, or more precisely, the filename and path of an existing icon. It can be a *jpg*, *bmp*, *png*, or *xpm* file. An icon is referenced from the `<node>` elements in this configuration via its attribute `alias`.

The element `<nodes>` contains several `<node>` elements. Each `<node>` element has the attributes `alias`, `open`, and `close`. The tag name of a node in the visualized tree is a reference of the attribute `alias` of a `<node>` element.

The attributes `open` and `close` are references to the attribute `alias` of an `<icon>` element. They define the icon of an expanded and collapsed node in the visualization. The attribute `open` defines the icon of a node in the tree, which is expanded, and the attribute `close` defines the icon of a node in the tree, which is collapsed.

The element `<nodes>` has the attributes `open`, and `close`, too. The values of these attributes are on the one hand the default values of `<node>` elements, with missing `open` or `close` attributes.

On the other hand, they are the default icons of nodes of a tree, for which a corresponding reference of a `<node>` element does not exist in the configuration. Corresponding elements are elements for which the tag name of a node in the tree matches with the attribute `alias` of a `<node>` element in the configuration.

Example A.5 (Icons and Nodes)

```
<config>
  <icons>
    <icon alias="open_1" file="opendir.xpm"/>
    <icon alias="open_2" file="text.xpm"/>
    <icon alias="close_1" file="closedir.xpm"/>
    <icon alias="close_2" file="edit.xpm"/>
  </icons>
  <nodes open="opendir.xpm" close="closedir.xpm">
    <node alias="system" open="open_1" close="close_1"/>
    <node alias="file" open="open_2" close="close_2"/>
  </nodes>
</config>
```

The Element <mouse_clicks> This element defines the entries of a pop-up menu of an item and the methods, which will be executed by a certain mouse click event. We describe the complete DTD of this element and the corresponding methods, which generate the pop-up menus and the mouse click events in Appendix A.2 and A.3.

We append mouse click events and pop-up menus to the items using the following arguments. These arguments can be retrieved from an application again and can be used by a called method:

- For methods called out from a *menu*, the sequence of the available arguments is

```
[Browser, Item]
```

- for methods called out from a *mouse click event*, the sequence of the available arguments is

```
[Browser, Item, Button, Type, XML_Code]
```

Thereby, `Browser` and `Item` contain the corresponding XPCE addresses. The necessary arguments, which we want to pass to a method can be defined with the attribute `prmtr` in the configuration. We use the alias `@Nr`, e.g., `[@1, @2, ...]` in order to reference one of these arguments.

Example A.6 (Configuration of Mouse Clicks)

```
<config>
  <icons>
    <icon alias="opendir" file="opendir.xpm"/>
  </icons>
  <nodes open="opendir.xpm" close="closedir.xpm">
    <node alias="system" open="opendir" close="closedir"/>
  </nodes>
  <mouse_clicks>
    <mouse_click alias="file">
      <on_click button="left" type="double" module="visur"
                predicate="emacs" prmtrs="[@2]"/>
      <popup>
        <menu name="Visualizations" end_group="off">
          <menu name="Global Cross References" end_group="on">
            <menu name="Incoming and Outgoing References"
                  module="visur" predicate="visualize_graph"
                  prmtrs="[@2, in_out]" end_group="off"/>
            <menu name="Incoming File References"
                  module="visur" predicate="visualize_graph"
                  prmtrs="[@2, incoming]" end_group="off"/>
            <menu name="Outgoing File References"
                  module="visur" predicate="visualize_graph"
                  prmtrs="[@2, outgoing]" end_group="off"/>
        </menu>
      </popup>
    </mouse_click>
  </mouse_clicks>
</config>
```

The DTD of the Tree

The hierarchy browser visualizes trees, which are written in an XML document. Now, we explain the DTD of this XML document. In the following DTD, the tag name `node` is arbitrary and can be any other name each time it is used in the tree, too.

DTD:

```
<!ELEMENT node node*>
<!ATTLIST node
   name CDATA #implied
   mouse_click CDATA #implied
   open CDATA #implied
   close CDATA #implied>
```

The elements of the hierarchy tree can contain additional attributes, too. E.g., the tree of the PROSORE database, which is visualized in the hierarchy browser of VISUR, additionally contains the attributes `date` and `path`.

```
<!ATTLIST node
   ...
   date CDATA #implied
   path CDATA #implied>
```

Example A.7 (Hierarchy Tree of the DDK) The following example shows a part of the file system tree as we use it for the hierarchy of the DDK.

```
<system name="DisLog">
   <unit name="basic_algebra" mouse_click="unit">
      <module name="basics" mouse_click="module">
         <file name="increment.pl" path="..."/>
         <file .../>
   ...
</system>
```

Each element of the tree can have further nested elements. The tree can have any depth. The tag names are arbitrary. It can be any name, e.g., using the hierarchy of the DDK, the tag names are `system`, `sources`, `unit`, `module`, `file`, and `predicate`. The attributes of the DTD have the following meaning:

`name`
: The value of `name` is displayed in the hierarchy browser. These are the item names in the visualization. If this attribute is not present, the tag name of the element will be displayed in the hierarchy browser.

`mouse_click`
: This is an optional attribute of each tree element. Using this attribute, we determine the predicate, which is called if a mouse click event is triggered. We define the pop-up menu of an item, too.

  This attribute is a reference to the attribute `alias` in the configuration (cf. Appendix A.4). The XML configuration document specifies among other things, what action has to be made, if a mouse click event is triggered. For the tree of the DDK, e.g., we configured that a click on the right mouse button shows a pop-up menu with diverse submenus, and a double click with the left mouse button on a file shows the content of the file in the PROLOG editor *emacs*.

`open`, `close`
: The attributes `open` and `close` determine the icons, which are shown in the hierarchy browser. The values contain complete file-paths to icons. Icons are displayed beside the label of a node. They can be configured in dependency of the node status. If the node is expanded the icon from the attribute `open` is displayed and if the node is collapsed, the icon of the attribute `closed` is displayed.

Example A.8 (The Configuration of the Hierarchy Tree) We show an abstract part of our solar system in the hierarchy browser. The XML configuration document defines the icons for the items in the browser. We use the pictures `opendir.xpm` and `closedir.xpm` as default icons of a node.

For the node `solar_system`, we will use the icons `binocular` and `hierarchy`. For the nodes `sun` and `planet`, we will use the default icons, except for the node `planet` with the attribute `name="Earth"`. For this node, we define the individual icon `ghost` in the hierarchy tree.

We define the pop-up menu and the predicate, which we want to be called by a double click on the left mouse button on the item `solar_system`, too.

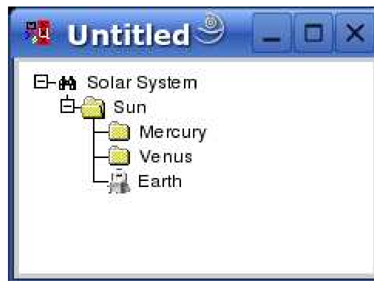


Figure A.3: The Hierarchy Browser

```
?- Config =
   config:[]:[
     icons:[]:[
       icon:[alias:a, file:'binocular.xpm']:[],
       icon:[alias:b, file:'hierarchy.xpm']:[] ],
     nodes:[open:'opendir.xpm', close:'closedir.xpm']:[
       node:[alias:solar_system, open:a, close:b]:[] ],
     mouse_clicks:[]:[
       mouse_click:[alias:solar_system]:[
         on_click:[button:left, type:double, call:view_data,
           prmtrs:[@1], message:prolog]:[],
         popup:[]:[
            menu:[name:'Data', end_group:on]:[
              menu:[name:'Population', call:population,
                prmtrs:[@1], message:prolog, end_group:off]:[]
            ]]]] ],
```

```
    Tree =
    solar_system:[name:'Solar Systems']:[
      sun:[name:'Sun', mouse_click:solar_system]:[
        planet:[name:'Merkur', mouse_click:solar_system]:[],
        planet:[name:'Venus', mouse_click:solar_system]:[],
        planet:[name:'Earth', mouse_click:solar_system,
           open:'ghost.xpm', close:'ghost.xpm']:[]] ],
    hierarchy_to_browser_ddk(Config, Tree, _).

Yes
```

Example A.9 (GXL to Hierarchy-Tree) We convert a GXL document into an XML hierarchy tree by calling

```
gxl_to_tree(+Gxl, -Tree).
```

This method creates an XML document, which only contains the nodes of a GXL document. The document is hierarchically structured in the following way: if two nodes of the GXL document are connected with a directed edge, then we embed the second node as XML element in the XML element of the first node. Additionally, the `<node>` elements of the GXL document are transformed: from each GXL node, only the ID is extracted and used as tag name in the new XML document.

The GXL document

```
<gxl>
  <graph ...>
    <node id="n_1">...</node>
    <node id="n_2">...</node>
    <edge id="e_1" from="n_1" to="n_2">...</edge>
  </graph>
</gxl>
```

will be transformed into

```
<tree>
  <n_1>
    <n_2/>
  </n1>
</tree>
```

The resulting document can be visualized by the hierarchy browser.

```
?- Gxl = gxl:[...]:[graph:[...]:[
      node:[id:n_1]:[...],
      node:[id:n_2]:[...],
      edge:[id:e_1, from:n_1, to:n_2]:[...] ]],
   gxl_to_tree(Gxl, Tree),
   hierarchy_to_browser_ddk(Tree, Browser).
```

```
Tree =
  tree:[]:[
    n_1:[]:[
      n_2:[]:[] ] ],
Browser = @2676802/hierarchy_browser_ddk.

Yes
```

We can visualize this tree using the hierarchy browser (cf. Figure A.4).

Figure A.4: The Hierarchy Browser Visualizing a GXL Document

A.5 The Alias Browser for the File History

In the PROSORE database and the libraries RAR, VISUR and GXL, we use XML documents as configuration files. In these libraries, each configuration file is not addressed directly via its path and filename, but each configuration file is addressed via an *alias*. We use an alias for the filename instead of using hard links for a filename.

Using an alias, we are much more flexible. In order to change the path of a configuration file, which is used several times in the library, we only need to change the associated path of the corresponding alias. Doing so, all relevant parts of the whole source code are changed simultaneously; we need not manually change the path in the whole library several times. We can change the path at runtime, too, so that immediately a new configuration is used.

Using an alias has another advantage, too. We do not only associate *one* filename with an alias. In order to get a history of the *last used files*, an alias is associated to a *list* of files. This list is organized as a stack. The last file, which we put on the stack will be the first one, which we retrieve, when we query for the *file history* of an alias. Additionally, we are able to associate each alias with a *default file*. If a changed configuration does not work, we can access the default file. Often, the default file is queried, too, if entries in a configuration file are missing. The default file never should be removed of the file history.

All associated file names and aliases are stored in an XML file. We explain the DTD and how to retrieve the location of this file in this section. Furthermore, we explain the command line tools for managing aliases. We have implemented the *alias browser*, a graphical user interface, in order to view the existing aliases, and to administrate the

aliases and the file history in a comfortable way. Using the alias browser, we can execute the command line methods for managing aliases.

Retrieving Associated Files and Content

All aliases and corresponding files are saved in an XML document. The following commands extract the queried information of this XML document. Each method, which queries the corresponding files of an alias only shows existing files. We call

```
alias_to_files(+Alias, -Files)
```

in order to retrieve the list of corresponding and existing files of an alias. If we just want to retrieve the last added file of the file history, we call

```
alias_to_file(+Alias, -File).
```

To each alias, we can define a default file. This file is used, e.g., if all other files in the file history are deleted. If we want to retrieve the default file of an alias, we call

```
alias_to_default_file(+Alias, -File).
```

Often, the content of the files we use is written in XML. We can query an associated file name of an alias, and read it into a PROLOG variable. The PROLOG variable represents the XML document in field notation.

Example A.10 (Retrieving Associated Files)

```
?- alias_to_file(Alias, File),
   dread(xml(dtd:Alias), File, Xml).

Yes
```

We combined these two steps in one step. In order to read the content of the last associated file directly into field notation, we call

```
alias_to_fn_triple(+Alias, -FN_Triple).
```

This and the next method only work, if the content of the file is XML. If we want to read the *default* file into field notation, we call

```
alias_to_default_fn_triple(+Alias, -FN_Triple).
```

In order to read a file into field notation in a correct way, the two methods above check, if there exists a corresponding DTD for this alias. If there exists a corresponding DTD, it will be used by `dread/3` to read the file into field notation (cf. Section 6.2). If there does not exist a DTD alias, the XML document is read without any DTD.

In order to retrieve the associated DTD of an alias, we call

```
alias_to_dtd(+Alias, -DTD).
```

The DTD definition associated to an alias is used to read a file, which content is XML, in a proper way into field notation. In order to read an XML document, we call `dread/3`. The associated DTDs of the aliases are defined in the file `alias_to_dtd.pl`. There, further DTD definitions can be added, too.

Adding an Alias-File Association

In order to associate a file to an alias, we call

```
add_file_to_alias(+Alias, +File).
```

This method associates a file to the alias, including the whole file path. If the file exists in the file history of this alias, yet, it will be deleted from the file history and it will be added to the top of the file history, again. If the alias does not exist, yet, the alias will be new created, and the file will be added to the file history.

The file will only be added to the associated file history, if the file exists. This will be proofed before. Adding a file to the file history of an alias deletes all non existing files in the associated file history of this alias.

Often, we want to add files to the file history of an alias, which have a relative path to the executed system DDK. If we want to add a file to an alias, which has a relative path, we call

```
add_file_to_alias(+Alias, +DDK_Var:File).
```

Thereby, `DDK_Var` is the name of a variable, which can be queried using

```
dislog_variable_get/2.
```

Valid values of `DDK_Var` are, e.g.,

```
home, source_path, sca, configuration.
```

The prefix and suffix are not concatenated and saved as absolute path in the file history. The relative variable `DDK_Var`, containing the prefix of the path, and the suffix `File` of the path are saved separately in the file history. and will be concatenated only at runtime each time it is needed.

A file can be associated to an alias as default file. A file is marked as default, if it has the attribute `type="default"` in its attribute list (cf. the DTD of the file history). In order to add a default file, we call either

```
add_default_file_to_alias(+Alias, +File),
```

or

```
add_default_file_to_alias(+Alias, +DDK_Var:File).
```

Each alias can only have one default file. Adding a further default file causes that the existing default file will be replaced by the new one.

Deleting an Alias-File Association or an Alias

If we only want to delete one alias-file association, we call

```
delete_file_from_alias(+Alias, +File).
```

Files, which are marked as default file, should not be deleted, because these files are absolute necessary. They can be replaced by another default file, but this is not advisable for aliases used by the PROSORE database and the libraries RAR, VISUR and GXL. Deleting a default file causes faults in all methods, which ask for the default file!

In order to delete a complete alias including all associated files, we call

```
delete_alias(+Alias).
```

Analogous to the method `delete_file_from_alias/2`, we should not completely delete an alias, which contains default files of the PROSORE database and the libraries RAR, VISUR and GXL.

The File History and its DTD

The file history is an XML document. In the file history, all aliases and their corresponding files and default files of the aliases are written. The filename of the file history can be retrieved calling

```
history_file(-File).
```

In order to read the file history into a PROLOG variable using field notation, we call

```
history_file_to_fn_triple(-FN_Triple).
```

Example A.11 (File History) The following XML document shows an example of a part of the file history.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<file_history>
   <aliases>
      <alias name="gxl_config">
        <file name="config.xml" path="config.xml"/>
        <file name="config_2.xml type="default"
           prefix="home" path="config_2.xml" />
        ...
      </alias>
      ...
   </aliases>
</file_history>
```

The DTD of the file history looks as follows:

DTD:

```
<!ELEMENT file_history aliases>

<!ELEMENT aliases alias*>

<!ELEMENT alias file*>
<!ATTLIST alias
   name CDATA #required>

<!ELEMENT file EMPTY>
<!ATTLIST file
   type default|normal|not_existing #implied
   path CDATA #required
   prefix CDATA #implied>
```

The file history can be visualized using the alias browser, too (cf. next section). In order to visualize the file history, we have to extend the DTD of the file history.

The Alias Browser

We have implemented a GUI, which simplifies the handling with aliases (cf. Figure A.5). All methods mentioned above are integrated in the alias browser. The alias browser is based on the hierarchy browser and on the methods for generating pop-up menus and mouse click events. The pop-up menus and mouse click events are defined in the file, which is associated to the alias `alias_browser`. In order to execute the alias browser, we just call

```
alias_browser.
```

First, the alias browser retrieves the XML document of the file history by calling the method `history_file/1`. The alias browser uses an extended DTD for the element `<file>` of this XML document, so in the next step, we add additional attributes to the elements `<file>`.

Using an extended DTD for the alias browser allows us to generate an individual name for the visualization of a file in the browser. And, second, we can allocate a proper symbol for default files, normal files and non existing files. The extended information is added to the XML document during the runtime of the alias browser.

DTD:

```
<!ATTLIST file
   mouse_click CDATA #implied
   open CDATA #implied
   close CDATA #implied
   name CDATA #implied
   type default|normal|not_existing #implied
   path CDATA #required
   prefix CDATA #implied>
```

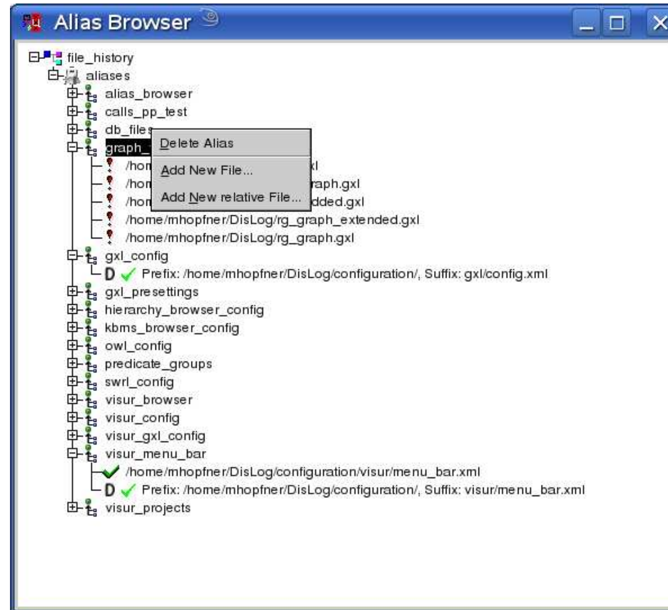


Figure A.5: The Alias Browser

We visualize an element in the alias browser in dependency of the element and the properties of the element. The root element is called <file_history> and it is visualized using the left icon of Figure A.6.

Figure A.6: The Root, Aliases and Alias Icons

The root element contains the element <aliases>. This element contains the aliases of the ProSoRe database and the libraries RaR, VisuR and GXL. It is visualized using the middle icon of Figure A.6. Each <alias> element contains the files associated to this alias. This list is a file history. The element <alias> is visualized using the right icon of Figure A.6.

Each <file> element is visualized using one of the following three icons of Figure A.7. If a file is contained in the file history and if the file exists in the file system, we use the first icon. If the file is additional defined as the *default* file of this alias, we use the second icon. And at last, if the file is contained in the file history, but it does not exists in file system, we use the third icon.

Each item has its individual pop-up menu. The item *aliases* only contains a pop-up menu, with which we can add a new alias to the XML document. An *alias*-item has a pop-up menu, with which we can either delete the corresponding alias, including the corresponding file history, or with which we can add a new file to the alias.

Figure A.7: The File Icons of the Alias Browser

A *file*-item has several different pop-up menus: if it is a normal file, we can view it in the PROLOG emacs editor, or we can delete it from the file history. If it is the default file, we only can view the file in the PROLOG emacs editor. And if it is a non existing file, we can delete it from the file history.

Appendix B

The XML-Based Configuration of the Picture Class for GXL Graphs

The visualization method `gxl_to_picture/3` is configured by an XML document. In the XML configuration document the properties and the behavior of nodes, edges, arrows, mouse click events, pop-up menus and so on can be easily and individually configured.

We have prepared a default XML configuration document. This default XML configuration document is used by the method `gxl_to_picture/3`. If the user does not specify a configuration, then the default XML configuration document is automatically used. In the following sections, we describe how to obtain the default XML configuration document and we describe the possible entries. We specify the XML configuration document by a DTD and give some examples.

B.1 The Default Configuration for Graph Pictures

The default XML configuration document is saved in a file. This file is an XML document, which contains all necessary parameters and configuration possibilities used by the visualization method. The file is associated to an alias, in order to get the content of this file quickly into a prolog variable. The alias of this file is `gxl_config` (cf. Appendix A.5). We retrieve the system path of this file by calling the method

```
alias_to_default_file(gxl_config, -File).
```

Based on this default configuration, we retrieve missing parameters, which are needed in order to visualize a graph in an XPCE picture. In Appendix D.4, we present a part of this default XML configuration document.

The easiest way to get a valid configuration for the visualization method is to load the *default* XML configuration document into a variable using

```
alias_to_default_fn_triple(gxl_config, -Config).
```

On the basis of this configuration, we make changes and adapt the configuration. In the simplest case, the configuration `Config` has the value `<config/>`. This configuration

always works. Using this configuration, no pop-ups and mouse click events are configured. If possible, missing configuration elements of incomplete configurations will be automatically retrieved from the default XML configuration document mentioned above. These elements are added to the configuration, if they are needed.

B.2 The DTD of the XML Configuration Document

The XML configuration document contains the elements

```
<sizes>, <handles>, <arrows>, <mouse_clicks>
```

for configuring the behavior of mouse clicks, pop-up menus and the visualization of nodes and edges. The following XML fragment is an example of this configuration:

Example B.1 (XML Default Configuration)

```
<config>
  <sizes>
    <size alias="small" points="6"/>
    ...
  </sizes>
  <arrows> ... </arrows>
  <handles> ... </handles>
  <mouse_clicks>
    <mouse_click alias="individual">
      <menu ...> ... </menu>
      <on_click ...> ... </on_click>
    </mouse_clicks>
  </mouse_clicks>
</config>
```

First, we just summarize all parts of the configuration DTD and explain the elements and attributes of the XML configuration document. The DTD consists of the following main elements, which are explained in detail in the subsections below.

DTD:

```
<!ELEMENT config sizes? handles? arrows? mouse_clicks?>

<!ELEMENT sizes size*>

<!ELEMENT size EMPTY>
<!ATTLIST size
   alias CDATA #required
   points CDATA #required>

<!ELEMENT arrows arrow*>
```

```
<!ELEMENT arrow EMPTY>
<!ATTLIST arrow
   alias CDATA #required
   pen (line|CDATA) #required
   width CDATA #required
   length CDATA #required
   color (line|CDATA) #required
   fill_pattern (nil|white_image|black_image) #required
   style (open|closed) #required>

<!ELEMENT handles handle_group*>

<!ELEMENT handle_group handle*>
<!ATTLIST handle_group
   alias CDATA #required

<!ELEMENT handle EMPTY>
<!ATTLIST handle
   direction (in|out) #required>
   pos (top|bottom|left|right|
        top_left|top_right|
        bottom_left|bottom_right) #required
   reference (symbol|text) #required
   factor CDATA #required>

<!ELEMENT mouse_clicks mouse_click*>

<!ELEMENT mouse_click on_click* popup?>
<!ATTLIST mouse_click
  alias CDATA #required>

<!ELEMENT on_click EMPTY>
<!ATTLIST on_click
  button (left|middle|right) #required
  type (single|double|triple) #required
  module CDATA "user"
  predicate CDATA #required
  prmtrs CDATA #required
  message (prolog|class) "prolog">

<!ELEMENT popup menu*>

<!ELEMENT menu menu*>
<!ATTLIST menu
   name CDATA #required
   end_group (on|off) #implied
   module CDATA #implied
   predicate CDATA #implied
   prmtrs CDATA #implied
   message (prolog|class) #implied>
```

The Elements <sizes> and <size>

These elements define aliases for the sizes of the node symbols. Each alias is assigned to an integer. In a GXL document, we can set the size of a node symbol either using an alias, whose corresponding integer value is defined in the XML configuration document, or we use directly an integer. The corresponding DTD of the XML configuration document looks as follows:

DTD:

```
<!ELEMENT sizes size*>

<!ELEMENT size EMPTY>
<!ATTLIST size
   alias CDATA #required
   points CDATA #required>
```

The element `<size>` contains the following attributes:

`alias`
: This attribute is an alias for the size. The alias is referenced from each element `<node>` of a GXL document. We can add further aliases for different sizes, too. The following aliases are defined in the default XML configuration document:

    `small, medium, large.`

`points`
: A GXL node contains the attribute `size`, which refers to the attribute `alias` mentioned above. The value of the attribute `points` is an integer, which determines the size of a node symbol in a picture.

Example B.2 (The Element Sizes)

```
<sizes>
   <size alias="small" points="6"/>
   <size alias="medium" points="18"/>
   <size alias="large" points="36"/>
</sizes>
```

The Elements <arrows> and <arrow>

The element `<arrows>` specifies the appearance of the arrows of an edge. If we visualize edges in an XPCE picture, there are several possibilities to draw arrows at the beginning and at the end of an edge. Figure B.1 shows several examples. We draw open or closed, filled or blank arrows. In the XML configuration document, the attribute `alias` of an `<arrow>` element is referenced from the attributes `first_arrow` or `second_arrow` of a GXL edge, which is defined in a GXL document.

Example B.3 (The Element Arrows)

```
<arrows>
   <arrow alias="arrow_1"
      width="15" length="20"
      fill_pattern="nil" color="black"
      style="open" pen="1"/>
</arrows>
```

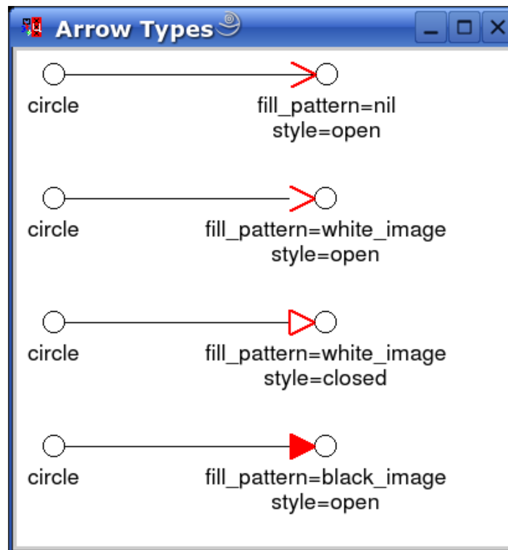


Figure B.1: Different Arrow Types and their Parameters

The corresponding part of the DTD looks as follows:

DTD:

```
<!ELEMENT arrows arrow*>

<!ELEMENT arrow EMPTY>
<!ATTLIST arrow
   alias CDATA #required
   pen (line|CDATA) #required
   width CDATA #required
   length CDATA #required
   color (line|CDATA) #required
   fill_pattern (nil|white_image|black_image) #required
   style (open|closed) #required>
```

`alias`
: This alias is referenced by the attribute `first_arrow`, and `second_arrow` of an edge in a GXL document. The arrows of the referencing edge are configured, as it is defined in the element `<arrow>` of the XML configuration document.

`pen`
: is an integer which determines the thickness of the arrow lines in pixel. `pen` can also have the value `line`. Then, the arrow lines will have the same thickness as the underlying line between the two connected nodes.

`width`
: is an integer which determines the width of the arrow in pixel.

`length`
: is an integer and determines the length of the arrow in pixel. Together with the attribute `width`, we determine if an arrow is large, small, wide or slight. The ratio between `width` and `length` determines, if the arrow is wide or slight.

`color`
: This attribute determines the color of the arrow. The value of the color is analogous to the background color (cf. attribute `background`, page 147). `color` can also have the value `line`. Then, the arrow will have the same color as the underlying line between the two connected nodes.

`fill_pattern`
: The value of `fill_pattern` can be `nil`, `white_image` or `black_image` (for an example, cf. Figure B.1).

- `nil:`
  the triangle of the arrow is not filled with anything. The edge line goes into the triangle and is complete visible.
- `white_image:`
  the edge line just goes to the beginning of the triangle.
- `black_image:`
  the triangle of the arrows is filled with the color of the triangle lines.

`style`
: The value of `style` can be `open` or `closed` (for an example, cf. Figure B.1).

- `open:`
  the triangle of the arrow is open, the third line of the triangle is missing.
- `closed:`
  the triangle of the arrow is closed, the triangle has three lines. This means that the two endings of the arrow lines are connected with each other.

The Element <handles>, <handle_group> and <handle>

In a visualization, an edge can only be connected to certain, predefined points of a node. Such a predefined point is called *handle*. A node can have many handles. The handles

of a node can be anywhere in or outside the node symbol. They are user-defined. The element `<handles>` defines the individual anchor points of a node, to which an edge is connected. Each subelement `<handle_group>` of the element `<handles>` defines multiple anchors. The attribute `alias` of the subelement `<handle_group>` is referenced by the attribute `handles` of a node in a GXL document (cf. Section 6.1.1, page 148). The node will be visualized using this anchor settings.

Example B.4 (Handles)

```
<handles>
   <handle_group alias="default">
      <handle pos="left" reference="text"
            direction="in" factor="100"/>
      <handle pos="top" reference="symbol"
            direction="in" factor="100"/>
      <handle pos="left" reference="text"
            direction="out" factor="100"/>
      ...
   </handle_group>
   <handle_group alias="node"
      <handle pos="left" reference="text"
            direction="out" factor="100"/>
</handles>
```

This part of the DTD looks as follows.

DTD:

```
<!ELEMENT handles handle_group*>

<!ELEMENT handle_group handle*>
<!ATTLIST handle_group
   alias CDATA #required

<!ELEMENT handle EMPTY>
<!ATTLIST handle
   direction (in|out) #required>
   pos (top|bottom|left|right|
        top_left|top_right|
        bottom_left|bottom_right) #required
   reference (symbol|text) #required
   factor CDATA #required>
```

In our visualization, a node is composed of a *symbol part* and below the symbol part, a *text part*. The anchor points can be in the middle or at the border of the symbol part, or at the border of the text part. We can configure the anchors for incoming and outgoing edges (cf. Figure B.2).

If there is no handle referenced in a GXL node element, then the settings of the *default* handle group are used. Each handle group consists of multiple handles. Each handle

defines an anchor point of the node, to which an edge can be connected. The attributes of the element `<handle>` have the following meaning:

`pos`
: Each `<handle>` element configures one anchor point. There exist eight anchor points. We can set an anchor in the middle of the *left*, *right*, *top* and *bottom* side, or at the *top* and *left*, *top* and *right*, *bottom* and *left*, *bottom* and *right* corner. We use the values `left`, `right`, `top`, `bottom`, `top_left`, `top_right`, `bottom_left`, and `bottom_right` for this attribute.

`reference`
: can be `text`, or `symbol`. If it is `text`, the anchor points are arranged around the text field of a node. If it is `symbol`, the anchor points are arranged around the symbol of a node.

`factor`
: determines the percentage distance to the point zero of of the coordinate system. The point zero of the coordinate system is in the middle of the symbol. If the attribute `factor` has the value `100`, and the attribute `reference` has the value `symbol`, the anchor for the edge will be at the border of the symbol. If `factor` has the value `0`, the anchor will be in the middle of the symbol.

`direction`
: determines, if we configure the anchor for outgoing or incoming edges. The value can be `out` or `in`.

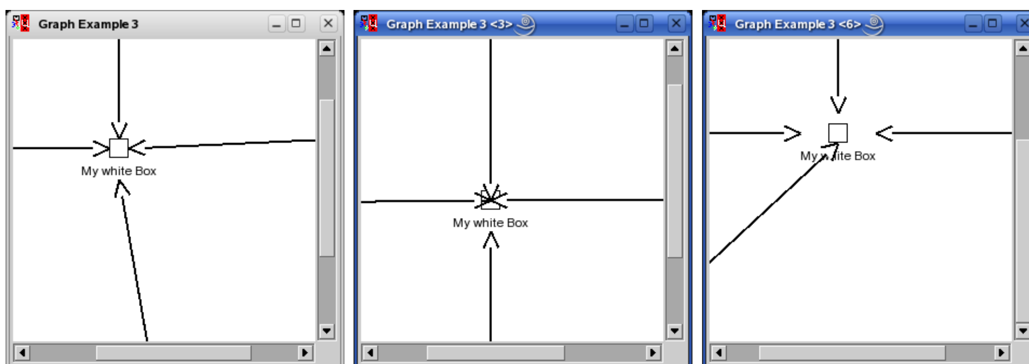


Figure B.2: Different Handles

The Element <mouse_clicks>

The element `<mouse_clicks>` determines, which action will be executed by a mouse click on a node, an edge or the background of the picture. It also determines the pop-up menu of a node, an edge or a picture. Each action or pop-up menu can be individually

defined, e.g., we can determine which mouse button we have to click and if we have to click once, or double.

The corresponding part of the DTD looks as follows:

DTD:

```
<!ELEMENT mouse_clicks mouse_click*>

<!ELEMENT mouse_click on_click* popup?>
<!ATTLIST mouse_click
  alias CDATA #required>
```

The attribute `alias` is referenced by the attribute `mouse_click` of a node, or an edge of a GXL document or by the XML configuration document itself. The pop-up menus and on click events defined in the element `<mouse_click>` will be attached to the corresponding XPCE node object, edge object or picture.

As the element `<mouse_clicks>` is used in the configuration of other applications, e.g., the *hierarchy browser*, too, we describe the rest of the DTD and the possible use in Appendix A.2.

Remark: It is important to know that the method `gxl_to_picture/3` passes multiple information to an XPCE object (node object, edge object or picture). Each XPCE object stores this information as payload in an internal structure. The following information is passed to an XPCE object:

- the XPCE address of the picture,
- depending on whether it is about an XPCE node- or an edge-object, the corresponding GXL part of the node or edge, and the XPCE address of the node or edge.

This information can be retrieved again in a corresponding method, which is called by a *mouse click* or chosen from a *pop-up menu*. For more precise information about pop-up menus and mouse click events see Appendix A.2.

Appendix C

Writing Plug-ins

We can write additional plug-ins for SCAV, with which we can analyze and visualize additional dependencies. To query dependencies, we can either use the PROSORE database and fill the database with corresponding constraints. Or, we can adapt the basic interface method `select/4`. This method connects the PROSORE database and the analysis and visualization methods of SCAV. `select/4` is the basic method and all other methods of SCAV are built on this method. We do not necessarily have to use the PROSORE database. Additionally, we can adapt the methods for analysis and visualization, too, in order to retrieve specific results. The calls to these methods can be configured in the configuration of the browser.

The following examples show, how to create a plug-in to manage the ancestor relations (cf. Appendix C.1), and how to create a plug-in to manage JAVA dependencies (cf. Appendix C.2).

C.1 Ancestor Relations Plug-in

This example shows two alternatives to create a plug-in for the ancestor relations. The ancestor relations, and the call dependencies, respectively, are based on PROLOG facts. Both alternatives use the browser to select a subset of the persons. This offers the possibility to choose only the relations of a certain city and to visualize them, or to visualize the relations between the cities, regions, countries. Therefore, we need to built a hierarchy tree and a configuration file for the browser of SCAV. The hierarchy tree and the pop-up menus of the browser can be adapted. In the examples, we generate a corresponding XML document for the browser (cf. Appendix C.1.3), which defines the hierarchy tree. A second XML document defines the pop-up menus of the browser.

C.1.1 The Ancestor Facts

We want to write a plug-in to visualize the family circumstances (cf. Figure C.1), which are represented in the following facts:

```
ancestor('Birgit', 'Werner').
ancestor('Jochen', 'Werner').
ancestor('Werner', 'Anna').
ancestor('Elke', 'Anna').
ancestor('Anna', 'Gutrun').
ancestor('Anna', 'Anke').
ancestor('Hans', 'Gutrun').
ancestor('Hans', 'Anke').
```

To be able to select only parts of the family, we assign each person to a city by the following facts:

```
living_in('Kitzingen', 'Birgit').
living_in('Kitzingen', 'Jochen').
living_in('Kitzingen', 'Werner').
living_in('Kitzingen', 'Elke').
living_in('Hollywood', 'Anna').
living_in('Frankfurt', 'Hans').
living_in('Frankfurt', 'Gutrun').
living_in('Frankfurt', 'Anke').
```

For the hierarchy browser, we need a hierarchical structure (cf. Figure C.3). The examples use the location of the cities on the earth:

```
belongs_to(planet:'Earth', continent:'Amerika').
belongs_to(planet:'Earth', continent:'Europe').
belongs_to(continent:'Amerika', country:'USA').
belongs_to(country:'USA', region:'Florida').
belongs_to(region:'Florida', city:'Hollywood').
belongs_to(continent:'Europe', country:'Deutschland').
belongs_to(country:'Deutschland', region:'Bayern').
belongs_to(country:'Deutschland', region:'Hessen').
belongs_to(region:'Hessen', city:'Frankfurt').
belongs_to(region:'Bayern', city:'Kitzingen').
```

These facts can be written into a separate file. There, we can consult these facts into the PROLOG system. If these facts are written within the plug-in file, they should be defined `dynamic`, so that the plug-in file can be reloaded.

```
:- dynamic
      ancestor/2,
      living_in/2,
      belongs_to/2.

:- retractall( ancestor(_, _) ).
:- retractall( belongs_to(_, _) ).
:- retractall( living_in(_, _) ).
```

The following plug-ins show, how to adapt SCAV to visualize the dependencies above.

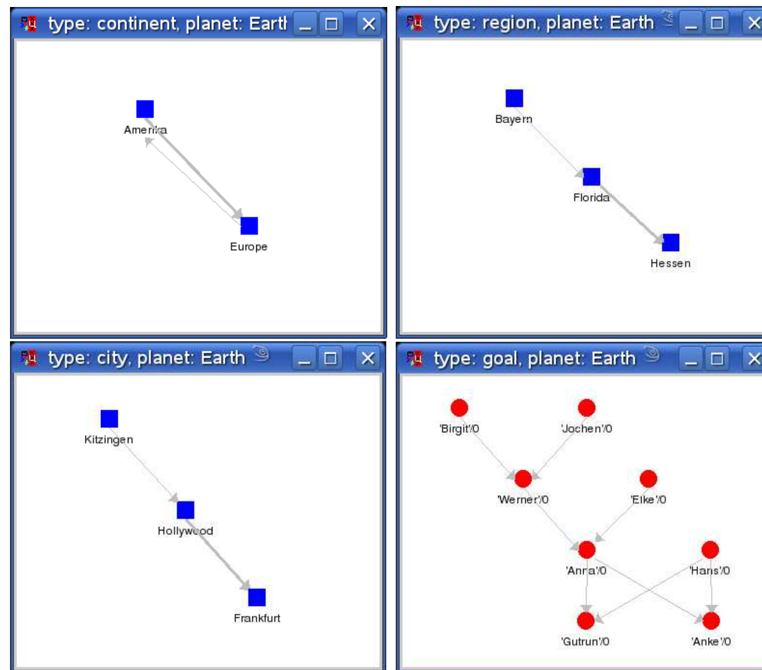


Figure C.1: The Continent, Region, City and Ancestor Relations

C.1.2 Plug-ins for SCAV

Both alternatives use the same hierarchy tree generated in Appendix C.1.3. The first alternative stores the facts statically in the PROSORE database. Plug-ins doing so are easier to implement. The second alternative changes the basic interface method `select/4`. The advantage is, that the database can be dynamically programmed. We can query a changed database without updating the PROSORE database of SCAV.

A plug-in is loaded either by the menu

```
File -> Load Plug-in ...
```

of SCAV, or by the PROLOG command line

```
load_plugin(+Plugin_File).
```

To be able to load another plug-in, each predicate of the plug-in file has to be defined `dynamic` in the plug-in file. Further on, we first have to retract all predicates in the file. The default plug-in of the PROSORE database looks like the following and is written in the file `plugin_select_prosore_db.pl`. It is loaded automatically during the loading of the DDK.

```
:- dynamic select/4.

:- retractall( rar:select(_, _, _, _) ).
```

```
select(Type, V1, V2, Object) :-
   sca_namespace_get(DB),
   hash_term(V1, H1),
   hash_term(V2, H2),
   Structure =.. [Type, H1, H2, V1, V2, Object, DB],
   call(rar:Structure).
```

Methods, which call `select/4`, use the `Type` values

```
rule, is_called_by, leaf_to_path.
```

Depending on the part of the PROSORE database, which should be queried, another value of `Type` is used. A new definition of the method `select/4` must contain a definition for each value of this parameter.

C.1.2.1 Filling the PROSORE Database

This plug-in inserts an adapted version of the ancestor relations to the PROSORE database. The source code analysis and the visualization methods work on the PROSORE database in the same way as for PROLOG source code.

Implementation:

```
1  :- dynamic plugin_rule/2.
2
3  :- retractall( plugin_rule(_, _) ).
4
5  plugin_rule(City, Rule) :-
6     ( ancestor(P, _)
7     ; ancestor(_, P) ),
8     living_in(City, P),
9     findall( atom:[module:user, predicate:Anc, arity:0]:[],
10       ancestor(P, Anc),
11       Ancestors ),
12    Rule =
13       rule:[]:[
14          head:[]:[
15             atom:[module:user, predicate:P, arity:0]:[] ],
16          body:[]:Ancestors ].
```

Line [1] defines the method `plugin_rule/2` as `dynamic`, and line [3] retracts former editions of this method. Lines [5-16] transform the `ancestor/2` and `living_in/2` facts to the PROLOGML syntax used by the PROSORE database.

Loading this plug-in by calling the method `load_plugin/1` inserts `rule` facts to the PROSORE database. Thereby, the method `plugin_rule/2` is called and backtracking is used. The database sources are the `ancestor/2`, and `living_in/2` facts. All other necessary facts of the PROSORE database, e.g., the `is_called_by` facts and the `leaf_to_path` facts are automatically generated by the method `load_plugin/1`.

Example C.1 (Facts for the Ancestor Relations) The following PROSORE database facts are created. The example only shows the corresponding facts for `Birgit`.

```
rar:rule(A, B, C, D, E, F).

A = 4977515,
B = 4252908,
C = (user:'Birgit')/0,
D = 'Kitzingen',
E = rule:[]:[
      head:[]:[
        atom:[arity:0, module:user, predicate:'Birgit']:[]],
      body:[]:[
        atom:[arity:0, module:user, predicate:'Werner']:[]]]
F = standard

Yes

rar:is_called_by(A, B, C, D, E, F).
A = 8671375,
B = 4252908,
C = (user:'Werner')/0,
D = 'Kitzingen',
E = (user:'Birgit')/0,
F = standard

Yes

rar:leaf_to_path(A, B, C, D, E, F).

A = 4252908,
B = 5433036,
C = 'Kitzingen',
D = nv,
E = leaf_to_path:[level:city, name:'Kitzingen']:[
      leaf_to_path:[level:region, name:'Bayern']:[
        leaf_to_path:[level:country, name:'Deutschland']:[
          leaf_to_path:[level:continent, name:'Europe']:[
            leaf_to_path:[level:planet, name:'Earth']:[]]]]]
F = standard
```

Example C.2 (The Pop-up Menus of the Hierarchy Browser) In order to create the *continent*, *region*, *city*, or *ancestor* dependency graphs (cf. Figure C.1), we created a corresponding XML document (cf. Appendix C.1.3), defining the pop-up menus (cf. Figure C.2). Choosing an item of the pop-up menu calls the method defined in the XML document.

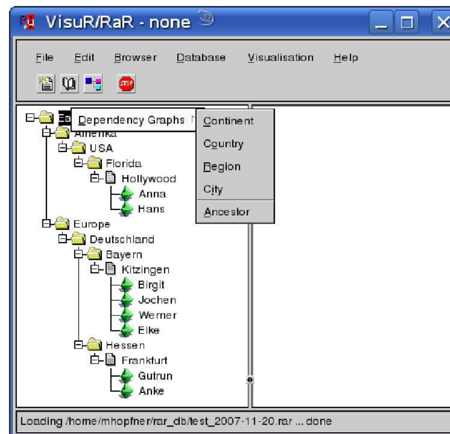


Figure C.2: The Hierarchy Browser and the Pop-up Menu

C.1.2.2 Changing the Interface Method of the Database

This example retracts the standard method `select/4` of the ProSoRe database and adds an adapted method to SCAV. The adapted method has to simulate the behavior of `select/4`. In particular, we have to simulate `select/4` for the parameters

```
rule, is_called_by, leaf_to_path.
```

The following plug-in adapts the basic interface method `select/4` to the ancestor relation facts.

Implementation:

```
1  :- dynamic
2        select/4,
3        select_rule/3,
4        create_path/2.
5
6  :- retractall( rar:select(_, _, _, _) ).
7  :- retractall( rar:select_rule(_, _, _) ).
8  :- retractall( rar:create_path(_, _) ).
9
10 select(rule, (user:I_A)/0, I_B, Object) :-
11    setof( (I_A, I_B, Object),
12       select_rule(I_A, I_B, Object),
13       Rs ),
14    member( (I_A, I_B, Object), Rs ).
15
16 select(is_called_by, (user:I_B)/0, I_A_2, (user:I_A_1)/0) :-
17    ancestor(I_A_1, I_B),
18    living_in(I_A_2, I_A_1).
19
20 select(leaf_to_path, I, nv, Object) :-
```

```
21    findall( I,
22       living_in(I, _),
23       Is_1 ),
24    list_to_ord_set(Is_1, Is_2),
25    member(I, Is_2),
26    create_path(city:I, Object).
27
28 select_rule(I_1, I_2, Object) :-
29    ( ancestor(I_1, _)
30    ; ancestor(_, I_1) ),
31    living_in(I_2, I_1),
32    findall( atom:[module:user, predicate:Anc, arity:0]:[],
33       ancestor(I_1, Anc),
34       Ancs ),
35    Object = rule:[]:[head:[]:[
36                atom:[arity:0, module:user, predicate:I_1]:[]],
37                      body:[]:Ancs ].
38
39 create_path(L_2:N_2, XML_2) :-
40    belongs_to(L_1:N_1, L_2:N_2),
41    create_path(L_1:N_1, XML_1),
42    XML_2 = leaf_to_path:[level:L_2, name:N_2]:[XML_1].
43 create_path(planet:N_1, XML) :-
44    XML = leaf_to_path:[level:planet, name:N_1]:[].
```

Lines [1-4] define the used methods dynamic, so that they can be retracted. Lines [6-8] retract the used methods of this plug-in. This is necessary to clean the memory, so that the same plug-in can be reloaded a second time again. Lines [10-14] and lines [28-37] define the method `select/4` for the parameter `rule`. Thereby, the method `select_rule/3` queries the `ancestor/2`, and `living_in/2` facts and transform them into a suitable PROLOGML syntax. In lines [16-18], we define the method `select/4` for the parameter `is_called_by`. The `ancestor/2` and `living_in/2` facts are transformed into the suitable PROSORE database syntax. At last, in lines [20-26] and lines [39-44], we define the method `select/4` for the parameter `leaf_to_path`. The `living_in/2` and `belongs_to/2` facts are adapted to the syntax of the PROSORE database.

C.1.3 Generating the Hierarchy Tree

The hierarchy tree assigns the persons to diverse continents, countries, regions and cities (cf. Figure C.3). The attribute list of the root element contains the name of the lowest level, in this case `city`.

The hierarchy tree has to be defined by the method `tree_for_browser/1`. Loading a plug-in, this method is called and the retrieved hierarchy is shown in the browser.

Implementation:

```
1  :- dynamic
2        xml_browser_config/1,
3        tree_for_browser/1,
4        tree_for_browser/2.
```

```
5
6  :- retractall( xml_browser_config(_) ).
7  :- retractall( tree_for_browser(_) ).
8  :- retractall( tree_for_browser(_, _) ).
9
10  tree_for_browser(XML_2) :-
11     findall( A,
12        ( belongs_to(A, _),
13          not( belongs_to(_, A) ) ),
14        Roots ),
15     list_to_ord_set(Roots, [Level:Name]),
16     tree_for_browser(Level:Name, XML_1),
17     XML_2 =
18        Level:[name:Name, path:Name, lowest_level:city]:XML_1.
19
20
21  tree_for_browser(L_1:Elt, XML_List) :-
22     findall( L_2:[name:D_2, path:D_2]:Sub_XML,
23        ( belongs_to(L_1:Elt, L_2:D_2),
24          tree_for_browser(L_2:D_2, Sub_XML) ),
25        XML_List_A ),
26     findall( person:[name:P, path:P]:[],
27        living_in(Elt, P),
28        XML_List_B ),
29     append(XML_List_A, XML_List_B, XML_List).
30
31  xml_browser_config('browser_cfg.xml').
```

Lines [1-4] define the predicates `dynamic`. This is necessary to be able to load another plug-in. Lines [6-8] retract former predicates, which are not necessary anymore. Lines [10-18] generate the hierarchy tree for the browser. In lines [11-15], we retrieve the root element of the tree. Therefore, the facts `belongs_to` are queried. Lines [16-29] generate the corresponding hierarchy for the browser, using the facts `belongs_to`. Line [31] contains the location of the configuration file of the browser. This configuration defines the pop-up menu of the hierarchy and the methods called by selecting a pop-up menu. The configuration is described in the following.

The following XML document contains the configuration of the browser. It can be extended by further pop-up menus, called methods, and other icons for nodes.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>

<config>
  <icons>
    <icon alias="opendir" file="16x16/opendir.xpm"/>
    <icon alias="closedir" file="16x16/closedir.xpm"/>
    <icon alias="person" file="16x16/forward.xpm"/>
  </icons>
  <nodes open="16x16/opendir.xpm" close="16x16/closedir.xpm">
    <node alias="person" open="person" close="person"/>
  </nodes>
  <mouse_clicks>
    <mouse_click alias="planet">
```

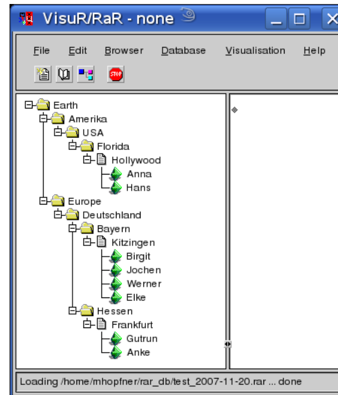


Figure C.3: The Hierarchy Tree of the Ancestor Relations

```
          <popup>
            <menu name="Dependency Graphs" end_group="on">
              <menu name="Continent" predicate="visualize_graph"
                    module="visur" prmtrs="[@2, continent]"
                    message="prolog" end_group="off"/>
              <menu name="Country" predicate="visualize_graph"
                    module="visur" prmtrs="[@2, country]"
                    message="prolog" end_group="off"/>
              <menu name="Region" predicate="visualize_graph"
                    prmtrs="[@2, region]" message="prolog"
                    module="visur" end_group="off"/>
              <menu name="City" predicate="visualize_graph"
                    prmtrs="[@2, city]" message="prolog"
                    module="visur" end_group="on"/>
              <menu name="Ancestor" predicate="visualize_graph"
                    prmtrs="[@2, goal]" message="prolog"
                    module="visur" end_group="off"/>
            </menu>
          </popup>
        </mouse_click>
        <mouse_click alias="continent">...</mouse_click>
        ...
      </mouse_clicks>
    </config>
```

We can define an individual pop-up menu for each browser node. The first `menu` element of this XML document adds the pop-up menu *Dependency Graphs* containing the submenus *Continent*, *Country*, *Region*, *City*, and *Ancestor* to the browser node beginning with the tag `planet`. Further menus, e.g., for nodes beginning with the tag `continent` are equivalent.

C.2 JAVA Plug-in

We want to insert several JAVA source code files into the PROSORE database. Therefore, this plug-in uses the JAVA XML representation JAML [12]. We assume, that each JAVA source code file is represented by a corresponding JAML file and we assume that we know the path to each JAML file. The JAML sources are parsed and transformed into a suitable syntax, which can be inserted into the PROSORE database. We can use the given directory structure of the file system for the hierarchy tree.

The following plug-in uses the equivalent technique of the example described in Appendix C.1.2.1. This means, instead of redefining the method `select/4`, we insert the transformed JAML sources into the PROSORE database. The advantage is, that we can quickly access the data. After we inserted the content into the PROSORE database, we do not need to access the JAML files on the hard disk any more.

Implementation:

```
1  :- dynamic
2        plugin_rule/2,
3        tree_for_browser/1,
4        jaml_source_code_files/1,
5        xml_browser_config/1.
6
7  :- retractall( plugin_rule(_, _) ).
8  :- retractall( tree_for_browser(_) ).
9  :- retractall( jaml_source_code_files(_) ).
10 :- retractall( xml_browser_config(_) ).
11
12 xml_browser_config('browser_cfg_jaml.xml').
13
14 plugin_rule(File, Rule) :-
15    jaml_source_code_files(Files),
16    member(File, Files),
17    dread_(xml, File, [XML|_]),
18    calls_cmcm(XML, C1:M1, _),
19    findall( atom:[module:C2, predicate:M2, arity:'']:[],
20       calls_cmcm(XML, C1:M1, C2:M2),
21       Calls ),
22    Rule =
23       rule:[module:C1]:[
24          head:[]:[
25             atom:[module:C1, predicate:M1, arity:'']:[]],
26          body:[]:Calls ].
27
28 calls_cmcm(Jaml, C1:M1, C2:M2) :-
29    calls_mm(Jaml, M1, M2),
30    owns_cm(Jaml, C1, M1),
31    owns_cm(Jaml, C2, M2).
32
33 tree_for_browser(T:A2:C) :-
```

```
34    Dir = 'Jaml/Javasrc_Jaml/xml/',
35    file_system_to_xml(Dir, T:A1:C),
36    A2 = [root:'', lowest_level:file, source_code:java|A1].
37
38 jaml_source_code_files(Files) :-
39    tree_for_browser(XML_Tree),
40    tree_to_files(XML_Tree, Files).
```

Lines [1-5] define the predicates of this plug-in `dynamic`. This is necessary to be able to load another plug-in. Lines [7-10] retract former predicates of this plug-in, which are not necessary anymore. Line [12] defines the location of the XML document for the browser configuration. Lines [14-25] define the method `plugin_rule/2`, which is called by the plug-in loading method `load_plugin/1` of SCAV. Line [15] retrieves the location of the JAML sources. Line [16] chooses a file of this list and line [17] reads the XML context into a PROLOG variable. Line [18] queries the call dependencies. Thereby, `calls_cmcm/3` (lines [28-31]) uses the methods defined in Section 4.1.3. Lines [33-36] generate the hierarchy tree for the browser. In this case, the hierarchy tree is based on the file system. Line [36] adds necessary attributes to the root element of the hierarchy tree. The method `tree_for_browser/1` is called by the method `load_plugin/1` of SCAV to add the hierarchy tree to the browser. The pop-up menu configuration defined by the method `xml_browser_config/1` (line [12]) is used. Lines [38-40] extract the filenames found in the hierarchy tree.

In order to configure the pop-up menus of the browser, we use the following XML document `browser_cfg_jaml.xml`.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<config>
  <icons>
    <icon alias="opendir" file="16x16/opendir.xpm"/>
    <icon alias="closedir" file="16x16/closedir.xpm"/>
    <icon alias="file" file="16x16/doc.xpm"/>
    <icon alias="predicate" file="16x16/forward.xpm"/>
  </icons>
  <nodes open="16x16/opendir.xpm" close="16x16/closedir.xpm">
    <node alias="dir" open="opendir" close="closedir"/>
    <node alias="file" open="file" close="file"/>
    <node alias="person" open="predicate" close="predicate"/>
  </nodes>
  <mouse_clicks>
    <mouse_click alias="dir">
      <popup>
        <menu name="Dependency Graphs" end_group="on">
          <menu name="Directory"
                predicate="xia_create_graph_visualize_graph"
                module="visur" prmtrs="[@2, dir]"
                message="prolog" end_group="off"/>
          <menu name="Ancestor"
                predicate="xia_create_graph_visualize_graph"
                module="visur" prmtrs="[@2, goal]"
                message="prolog" end_group="off"/>
        </menu>
      </popup>
```

```
    </mouse_click>
    <mouse_click alias="file">
      <on_click button="left" type="double" module="visur"
         predicate="view_with_emacs" prmtrs="[@1, @2]"/>
      <popup>
        <menu name="Dependency Graphs" end_group="on">
          <menu name="Goal"
                predicate="xia_create_graph_visualize_graph"
                module="visur" prmtrs="[@2, goal]"
                message="prolog" end_group="off"/>
        </menu>
      </popup>
    </mouse_click>
  </mouse_clicks>
</config>
```

Appendix D

Examples, Default Settings and Configurations

D.1 XML Document in Field Notation

The PROLOG rule

```
increment(X, Y) :-
  Goal = add(1, X, Y),
  call(Goal).
```

is represented in an XML document in the following way

```
<?xml version=’1.0’ encoding=’ISO-8859-1’ ?>
<rule module="user" operator=":-">
  <head>
    <atom module="user" predicate="increment" arity="2">
      <var name="X"/>
      <var name="Y"/>
    </atom>
  </head>
  <body>
    <atom module="user" predicate="=" arity="2">
      <var name="Goal"/>
      <term functor="add">
        <term functor="1"/>
        <var name="X"/>
        <var name="Y"/>
      </term>
    </atom>
    <atom module="user" predicate="call" arity="1">
      <var name="Goal"/>
    </atom>
  </body>
</rule>
```

The corresponding representation in PROLOG using field notation looks as follows:

```
rule:[module:user, operator:’:-’]:[
   head:[]:[
      atom:[module:user, predicate:increment,
         arity:2]:[
         var:[name:’X’]:[],
         var:[name:’Y’]:[] ] ],
   body:[]:[
      atom:[module:user, predicate:’=’,
         arity:2]:[
         var:[name:’Goal’]:[],
         term:[functor:add]:[
            term:[functor:1]:[],
            var:[name:’X’]:[],
            var:[name:’Y’]:[] ] ],
      atom:[module:user, predicate:call,
         arity:1]:[
         var:[name:’Goal’]:[] ] ] ]
```

D.2 XML Configuration Special Predicates

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<predicate_groups>
  <predicates group="meta" class="skip" type="hide_node">
    <atom predicate="," arity="2"
          module="user" calls="[0, 0]"/>
  </predicates>
  <predicates group="meta" class="a" type="duplicate_node">
    <atom predicate="if_then" arity="2"
          module="user" calls="[0, 0]"/>
  </predicates>
  <predicates group="meta" class="a" type="duplicate_node">
    <atom predicate="or" arity="2"
          module="user" calls="[0, 0]"/>
  </predicates>
  <predicates group="meta" class="test" type="duplicate_node">
    <atom predicate="my_meta" arity="3"
          module="rar_test_1" calls="[n, 1, n]"/>
  </predicates>
  <predicates group="meta" class="b" type="duplicate_node">
    <atom predicate="jaml_apply_method" arity="4"
          module="user" calls="[2, n, n, n]"/>
    <atom predicate="stock_goal_to_stock_file" arity="2"
          module="user" calls="[0, n]"/>
    <atom predicate="dsa_output_list" arity="2"
          module="user" calls="[1, n]"/>
    <atom predicate="dislog_dialog_variable" arity="2"
          module="user" calls="[3, n]"/>
    <atom predicate="dspy" arity="1"
          module="user" calls="[0]"/>
    <atom predicate="viewer" arity="3"
          module="visur" calls="[1, n, n]"/>
    <atom predicate="determine_runtime" arity="1"
          module="user" calls="[0]"/>
    <atom predicate="determine_runtime" arity="2"
          module="user" calls="[n, 0]"/>
    <atom predicate="transitive_closure" arity="3"
          module="rar" calls="[2, n, n]"/>
    <atom predicate="measure_runtime" arity="1"
          module="user" calls="[0]"/>
    <atom predicate="measure" arity="1"
          module="user" calls="[0]"/>
    <atom predicate="measure" arity="2"
          module="user" calls="[0, n]"/>
    <atom predicate="measure" arity="3"
          module="user" calls="[n, 0, n]"/>
    <atom predicate="measure_runtime" arity="2"
          module="user" calls="[n, 0]"/>
    <atom predicate="select" arity="3"
          module="rar_dml" calls="[4, n, n]"/>
    <atom predicate="select" arity="4"
          module="rar_dml" calls="[6, n, n, n]"/>
    <atom predicate="do" arity="2"
          module="user" calls="[n, 0]"/>
    <atom predicate="hidden" arity="1"
          module="user" calls="[0]"/>
    <atom predicate="hidden" arity="2"
          module="user" calls="[0, 0]"/>
    <atom predicate="call" arity="1"
          module="user" calls="[0]"/>
    <atom predicate="catch" arity="3"
          module="user" calls="[0, n, n]"/>
    <atom predicate="predicate_to_file" arity="2"
          module="user" calls="[n, 0]"/>
    <atom predicate="checklist" arity="2"
          module="user" calls="[1, n]"/>
    <atom predicate="checklist_with_status_bar" arity="3"
          module="user" calls="[n, 1, n]"/>
    <atom predicate="findall" arity="3"
          module="user" calls="[n, 0, n]"/>
    <atom predicate="forall" arity="2"
          module="user" calls="[0, 0]"/>
    <atom predicate="iterate_list" arity="4"
          module="user" calls="[3, n, n, n]"/>
    <atom predicate="maplist" arity="3"
          module="user" calls="[2, n, n]"/>
    <atom predicate="maplist" arity="2"
          module="user" calls="[2, n]"/>
    <atom predicate="maplist_with_status_bar" arity="4"
          module="user" calls="[n, 2, n, n]"/>
    <atom predicate="not" arity="1"
          module="user" calls="[0]"/>
  </predicates>
  <predicates group="meta" class="c" type="duplicate_node">
    <atom predicate="select" arity="3"
          module="rar" calls="[4, n, n]"/>
```

```
    <atom predicate="select" arity="4"
          module="rar" calls="[6, n, n, n]"/>
  </predicates>
  <predicates group="built_in" class="rar_2" type="duplicate_node">
    <atom predicate="called_predicates" arity="2"
          module="slice"/>
    <atom predicate="rar_transitive_closure" arity="2"
          module="slice"/>
  </predicates>
  <predicates group="library" class="ordsets" type="duplicate_node">
    <atom predicate="list_to_ord_set" arity="2" module="ordsets"/>
  </predicates>
  <predicates group="built_in" class="rar" type="duplicate_node">
    <atom predicate="writelq" arity="1" module="user"/>
    <atom predicate="special_predicate_attributes" arity="5"
          module="rar"/>
    <atom predicate="rar_variable_get" arity="2"
          module="user"/>
    <atom predicate="maplist_with_status_bar" arity="4"
          module="user"/>
    <atom predicate="c" arity="1"
          module="module_b"/>
    <atom predicate="rar_variable" arity="2"
          module="user"/>
    <atom predicate="rar_variable_set" arity="2"
          module="user"/>
  </predicates>
  <predicates group="built_in" class="xpce" type="duplicate_node">
    <atom predicate="get" arity="3" module="user"/>
    <atom predicate="get" arity="4" module="user"/>
  </predicates>
  <predicates group="built_in" class="dislog" type="duplicate_node">
    <atom predicate=":=" arity="2" module="user"/>
    <atom predicate="append" arity="2" module="user"/>
    <atom predicate="concat" arity="2" module="user"/>
    <atom predicate="concat" arity="3" module="user"/>
    <atom predicate="dislog_variable_get" arity="2" module="user"/>
    <atom predicate="dislog_variable_set" arity="2" module="user"/>
    <atom predicate="dislog_variable" arity="2" module="user"/>
    <atom predicate="dislog_variable" arity="3" module="user"/>
    <atom predicate="dread" arity="2" module="user"/>
    <atom predicate="dwrite" arity="2" module="user"/>
    <atom predicate="fn_to_xml" arity="1" module="user"/>
    <atom predicate="list_to_ord_set" arity="2" module="user"/>
    <atom predicate="name_append" arity="2" module="user"/>
    <atom predicate="name_append" arity="3" module="user"/>
    <atom predicate="ord_union" arity="3" module="user"/>
    <atom predicate="resolve" arity="3" module="user"/>
    <atom predicate="send" arity="4" module="user"/>
    <atom predicate="send_list" arity="3" module="user"/>
    <atom predicate="start_timer" arity="1" module="user"/>
    <atom predicate="substitute" arity="3" module="user"/>
    <atom predicate="union" arity="3" module="user"/>
    <atom predicate="write_list" arity="1" module="user"/>
  </predicates>
  <predicates group="irrelevant" class="nv" type="hide_node_hide_subtree">

    <atom predicate="e" arity="1" module="rar_test_1"/>
  </predicates>
  <predicates group="hide_nodes" class="nv" type="hide_node">
    <atom predicate="no_head" arity="0" module="user"/>
    <atom predicate="no_head" arity="0" module="module_b"/>
  </predicates>
  <predicates group="virtual_nodes" class="nv" type="hide_sub_tree">
    <atom predicate="virtual_node_for_outgoing_calls"
          arity="0" module="user"/>
  </predicates>
  <predicates group="built_in" class="prolog" type="duplicate_node"/>
  <predicates group="default" class="default" type="single"/>
  <predicates group="predicate" class="predicate" type="single"/>
</predicate_groups>
```

D.3 Default GXL Settings

The current default GXL document containing the presettings looks as follows:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
  <prmtr label="Default GXL"
        width="450"
        height="400"
        background="white"
        mouse_click="picture"/>
```

```
  <graph id="default_graph"
         edgeids="true"
         edgemode="directed"
         hypergraph="false">
    <node id="default_id_1">
      <prmtr mouse_click="node"
             color="grey"
             size="small"
             symbol="circle"
             x_pos="205"
             y_pos="169"
             handles="default">
        <string font_style="times, roman, 12"
                bubble="Default Node">
          default
          node
        </string>
      </prmtr>
    </node>
    <edge from="default_id_1"
          id="default_id_2"
          to="default_id_1">
      <prmtr arrows="both"
             color="black"
             pen="1"
             mouse_click="edge"
             first_arrow="first_arrow"
             second_arrow="second_arrow"
             weight="1">
        <string font_style="times, roman, 12"
                bubble="Default Edge">
        </string>
      </prmtr>
    </edge>
  </graph>
</gxl>
```

D.4 Default Visualization Configuration

The current default configuration for visualizing graph looks as follows:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<config>
  <gxl_layout
             x_start="90" y_start="90"
             x_step="140" y_step="70" y_variance="10"/>
  <symbols scc_size="medium">
    <symbol name="circle"/>
    <symbol name="box"/>
    <symbol name="rhombus"/>
    <symbol name="honeycomb"/>
    <symbol name="triangle"/>
  </symbols>
  <node_colors>
    <color name="grey" scc="true"/>
    <color name="red" scc="true"/>
    <color name="blue" scc="true"/>
    <color name="yellow" scc="true"/>
    <color name="orange" scc="true"/>
    <color name="green" scc="true"/>
    <color name="black" scc="true"/>
    <color name="white" scc="false"/>
  </node_colors>
  <sizes>
    <size alias="small" points="6"/>
    <size alias="medium" points="18"/>
    <size alias="large" points="36"/>
  </sizes>
  <edge_colors>
    <edge type="tree_edge" color="blue"/>
    <edge type="cross_edge" color="green"/>
    <edge type="back_edge" color="red"/>
    <edge type="forward_edge" color="green"/>
    <edge type="default" color="black"/>
  </edge_colors>
  <arrows>
    <arrow alias="first_arrow" pen="line" width="0"
      length="0" fill_pattern="white_image" color="line" style="open"/>
    <arrow alias="second_arrow" pen="line" width="5"
      length="6" fill_pattern="black_image" color="line" style="open"/>
  </arrows>
  <handles>
    <handle_group alias="default">
```

```
    <handle pos="top_left" reference="symbol" factor="100" direction="in"/>
    <handle pos="top_left" reference="symbol" factor="100" direction="out"/>
    <handle pos="top_right" reference="symbol" factor="100" direction="in"/>
    <handle pos="top_right" reference="symbol" factor="100" direction="out"/>
    <handle pos="bottom_left" reference="symbol" factor="100" direction="in"/>
    <handle pos="bottom_left" reference="symbol" factor="100" direction="out"/>
    <handle pos="bottom_right" reference="symbol" factor="100" direction="in"/>
    <handle pos="bottom_right" reference="symbol" factor="100" direction="out"/>
    <handle pos="left" reference="symbol" factor="100" direction="in"/>
    <handle pos="right" reference="symbol" factor="100" direction="in"/>
    <handle pos="bottom" reference="text" factor="100" direction="in"/>
    <handle pos="top" reference="symbol" factor="100" direction="in"/>
    <handle pos="left" reference="symbol" factor="100" direction="out"/>
    <handle pos="right" reference="symbol" factor="100" direction="out"/>
    <handle pos="bottom" reference="symbol" factor="100" direction="out"/>
    <handle pos="top" reference="symbol" factor="100" direction="out"/>
  </handle_group>
  <handle_group alias="text">
    <handle pos="top_left" reference="symbol" factor="100" direction="in"/>
    <handle pos="top_left" reference="symbol" factor="100" direction="out"/>
    <handle pos="top_right" reference="symbol" factor="100" direction="in"/>
    <handle pos="top_right" reference="symbol" factor="100" direction="out"/>
    <handle pos="bottom_left" reference="symbol" factor="100" direction="in"/>
    <handle pos="bottom_left" reference="symbol" factor="100" direction="out"/>
    <handle pos="bottom_right" reference="symbol" factor="100" direction="in"/>
    <handle pos="bottom_right" reference="symbol" factor="100" direction="out"/>
    <handle pos="left" reference="symbol" factor="100" direction="in"/>
    <handle pos="right" reference="symbol" factor="100" direction="in"/>
    <handle pos="bottom" reference="symbol" factor="100" direction="in"/>
    <handle pos="top" reference="symbol" factor="100" direction="in"/>
    <handle pos="left" reference="text" factor="100" direction="out"/>
    <handle pos="right" reference="text" factor="100" direction="out"/>
    <handle pos="bottom" reference="text" factor="100" direction="out"/>
    <handle pos="top" reference="text" factor="100" direction="out"/>
  </handle_group>
  <handle_group alias="node">
    <handle pos="left" reference="text" direction="out" factor="100"/>
  </handle_group>
</handles>

<mouse_clicks>
  <mouse_click alias="unit-unit">
    <popup>
      <menu name="Menu Unit:" predicate="writeln" prmtrs="['Menu Unit']"
        end_group="on"/>
      <menu name="Cross References" end_group="off">
        <menu name="Incoming and Outgoing Unit References" module="visur"
          predicate="visualize_calls" prmtrs="[@2, unit, in_out]" end_group="off"/>
        <menu name="Incoming Unit References" module="visur"
          predicate="visualize_calls" prmtrs="[@2, unit, incoming]"
          message="prolog" end_group="off"/>
        <menu name="Outgoing Unit References" module="visur"
          predicate="visualize_calls" prmtrs="[@2, unit, outgoing]"
          message="prolog" end_group="off"/>
      </menu>
      <menu name="View Node" module="visur" predicate="view_xml"
        prmtrs="[@2]" message="prolog" end_group="off"/>
    </popup>
  </mouse_click>
  <mouse_click alias="incoming-unit">
    <popup>
      <menu name="Menu Unit:" predicate="writeln" prmtrs="['Menu Unit']"
        end_group="on"/>
      <menu name="Cross References" end_group="off">
        <menu name="Incoming and Outgoing Unit References" module="visur"
          predicate="visualize_calls" prmtrs="[@2, unit, in_out]"
          message="prolog" end_group="off"/>
        <menu name="Incoming Unit References" module="visur"
          predicate="visualize_calls" prmtrs="[@2, unit, incoming]"
          message="prolog" end_group="off"/>
        <menu name="Outgoing Unit References" module="visur"
          predicate="visualize_calls" prmtrs="[@2, unit, outgoing]"
          message="prolog" end_group="off"/>
      </menu>
      <menu name="View Node" module="visur" predicate="view_xml"
        prmtrs="[@2]" message="prolog" end_group="off"/>
    </popup>
  </mouse_click>
  <mouse_click alias="outgoing-unit">
    <popup>
      <menu name="Menu Unit:" predicate="writeln" prmtrs="['Menu Unit']" end_group="on"/>
      <menu name="Cross References" end_group="off">
        <menu name="Incoming and Outgoing Unit References" module="visur"
          predicate="visualize_calls" prmtrs="[@2, unit, in_out]"
          message="prolog" end_group="off"/>
        <menu name="Incoming Unit References" module="visur"
          predicate="visualize_calls" prmtrs="[@2, unit, incoming]"
          message="prolog" end_group="off"/>
```

```
          <menu name="Outgoing Unit References" module="visur"
            predicate="visualize_calls" prmtrs="[@2, unit, outgoing]"
            message="prolog" end_group="off"/>
        </menu>
        <menu name="View Node" module="visur" predicate="view_xml"
          prmtrs="[@2]" message="prolog" end_group="off"/>
      </popup>
    </mouse_click>
    <mouse_click alias="in_out-unit">
      <popup>
        <menu name="Menu Unit:" predicate="writeln" prmtrs="['Menu Unit']" end_group="on"/>
        <menu name="Cross References" end_group="off">
          <menu name="Incoming and Outgoing Unit References" module="visur"
            predicate="visualize_calls" prmtrs="[@2, unit, in_out]" message="prolog" end_group="off"/>
          <menu name="Incoming Unit References" module="visur"
            predicate="visualize_calls" prmtrs="[@2, unit, incoming]"
            message="prolog" end_group="off"/>
          <menu name="Outgoing Unit References" module="visur"
            predicate="visualize_calls" prmtrs="[@2, unit, outgoing]"
            message="prolog" end_group="off"/>
        </menu>
        <menu name="View Node" module="visur" predicate="view_xml"
          prmtrs="[@2]" message="prolog" end_group="off"/>
      </popup>
    </mouse_click>
    <mouse_click alias="module-module">
      <popup>
        <menu name="Menu Module:" predicate="writeln" prmtrs="['Menu Module']"
          message="prolog" end_group="on"/>
        <menu name="Cross References" end_group="off">
          <menu name="Incoming and Outgoing Module References"
            module="visur" predicate="visualize_calls"
            prmtrs="[@2, module, in_out]" message="prolog" end_group="off"/>
          <menu name="Incoming Module References" module="visur"
            predicate="visualize_calls" prmtrs="[@2, module, incoming]"
            message="prolog" end_group="off"/>
          <menu name="Outgoing Module References" module="visur"
            predicate="visualize_calls" prmtrs="[@2, module, outgoing]"
            message="prolog" end_group="off"/>
        </menu>
        <menu name="View Node" module="visur" predicate="view_xml"
          prmtrs="[@2]" message="prolog" end_group="off"/>
      </popup>
    </mouse_click>
  <mouse_click alias="incoming-module">
    <popup>
      <menu name="Menu Module:" predicate="writeln" prmtrs="['Menu Module']"
        message="prolog"  end_group="on"/>
      <menu name="Cross References" end_group="off">
        <menu name="Incoming and Outgoing Module References"
          module="visur" predicate="visualize_calls"
          prmtrs="[@2, module, in_out]" message="prolog" end_group="off"/>
        <menu name="Incoming Module References" module="visur"
          predicate="visualize_calls" prmtrs="[@2, module, incoming]"
          message="prolog" end_group="off"/>
        <menu name="Outgoing Module References" module="visur"
          predicate="visualize_calls" prmtrs="[@2, module, outgoing]"
          message="prolog" end_group="off"/>
      </menu>
      <menu name="View Node" module="visur" predicate="view_xml"
        prmtrs="[@2]" message="prolog" end_group="off"/>
    </popup>
  </mouse_click>
  <mouse_click alias="outgoing-module">
    <popup>
      <menu name="Menu Module:" predicate="writeln"
        prmtrs="['Menu Module']" message="prolog"  end_group="on"/>
      <menu name="Cross References" end_group="off">
        <menu name="Incoming and Outgoing Module References"
          module="visur" predicate="visualize_calls"
          prmtrs="[@2, module, in_out]" message="prolog" end_group="off"/>
        <menu name="Incoming Module References" module="visur"
          predicate="visualize_calls" prmtrs="[@2, module, incoming]"
          message="prolog" end_group="off"/>
        <menu name="Outgoing Module References" module="visur"
          predicate="visualize_calls" prmtrs="[@2, module, outgoing]"
          message="prolog" end_group="off"/>
      </menu>
      <menu name="View Node" module="visur" predicate="view_xml"
        prmtrs="[@2]" message="prolog" end_group="off"/>
    </popup>
  </mouse_click>
  <mouse_click alias="in_out-module">
    <popup>
      <menu name="Menu Module:" predicate="writeln"
        prmtrs="['Menu Module']" message="prolog"  end_group="on"/>
      <menu name="Cross References" end_group="off">
        <menu name="Incoming and Outgoing Module References"
```

```
        module="visur" predicate="visualize_calls" prmtrs="[@2, module, in_out]"
        message="prolog" end_group="off"/>
      <menu name="Incoming Module References" module="visur"
        predicate="visualize_calls" prmtrs="[@2, module, incoming]"
        message="prolog" end_group="off"/>
      <menu name="Outgoing Module References" module="visur"
        predicate="visualize_calls" prmtrs="[@2, module, outgoing]"
        message="prolog" end_group="off"/>
    </menu>
    <menu name="View Node" module="visur" predicate="view_xml"
      prmtrs="[@2]" message="prolog" end_group="off"/>
  </popup>
</mouse_click>
<mouse_click alias="file-file">
  <on_click button="left" type="double" module="visur"
    predicate="view_with_emacs_double_click" prmtrs="[@2]"
    message="prolog"/>
  <popup>
    <menu name="Menu File:" predicate="writeln" prmtrs="['Menu File']" end_group="on"/>
    <menu name="Cross References" end_group="off">
      <menu name="Incoming and Outgoing File References"
        module="visur" predicate="visualize_calls"
        prmtrs="[@2, file, in_out]" message="prolog" end_group="off"/>
      <menu name="Incoming File References" module="visur"
        predicate="visualize_calls" prmtrs="[@2, file, incoming]"
        message="prolog" end_group="off"/>
      <menu name="Outgoing File References" module="visur"
        predicate="visualize_calls" prmtrs="[@2, file, outgoing]"
        message="prolog" end_group="off"/>
    </menu>
    <menu name="Visualize Rule/Goal Graph in new Picture" module="visur"
      predicate="visualize_calls" prmtrs="[@2]"
      message="prolog" end_group="off"/>
    <menu name="Edit File with XPCE emacs" module="visur"
      predicate="view_with_emacs_popup" prmtrs="[@2]"
      message="prolog" end_group="off"/>
    <menu name="View Node" module="visur"
      predicate="view_xml" prmtrs="[@2]"
      message="prolog" end_group="off"/>
  </popup>
</mouse_click>
<mouse_click alias="outgoing-file">
  <on_click button="left" type="double" module="visur"
    predicate="view_with_emacs_double_click"
    prmtrs="[@2]" message="prolog"/>
  <popup>
    <menu name="Menu File:" predicate="writeln"
      prmtrs="['Menu File']" end_group="on"/>
    <menu name="Cross References" end_group="off">
      <menu name="Incoming and Outgoing File References"
        module="visur" predicate="visualize_calls"
        prmtrs="[@2, file, in_out]" message="prolog"
        end_group="off"/>
      <menu name="Incoming File References" module="visur"
        predicate="visualize_calls" prmtrs="[@2, file, incoming]"
        message="prolog" end_group="off"/>
      <menu name="Outgoing File References" module="visur"
        predicate="visualize_calls" prmtrs="[@2, file, outgoing]"
        message="prolog" end_group="off"/>
    </menu>
    <menu name="Visualize Rule/Goal Graph in new Picture"
      module="visur" predicate="visualize_calls" prmtrs="[@2]"
      message="prolog" end_group="off"/>
    <menu name="Edit File with XPCE emacs" module="visur"
      predicate="view_with_emacs_popup" prmtrs="[@2]"
      message="prolog" end_group="off"/>
    <menu name="View Node" module="visur"
      predicate="view_xml" prmtrs="[@2]"
      message="prolog" end_group="off"/>
  </popup>
</mouse_click>
<mouse_click alias="incoming-file">
  <on_click button="left" type="double" module="visur"
    predicate="view_with_emacs_double_click" prmtrs="[@2]"
    message="prolog"/>
  <popup>
    <menu name="Menu File:" predicate="writeln" prmtrs="['Menu File']" end_group="on"/>
    <menu name="Cross References" end_group="off">
      <menu name="Incoming and Outgoing File References" module="visur"
      predicate="visualize_calls" prmtrs="[@2, file, in_out]"
      message="prolog" end_group="off"/>
      <menu name="Incoming File References" module="visur"
      predicate="visualize_calls" prmtrs="[@2, file, incoming]"
      message="prolog" end_group="off"/>
      <menu name="Outgoing File References" module="visur"
      predicate="visualize_calls" prmtrs="[@2, file, outgoing]"
      message="prolog" end_group="off"/>
    </menu>
```

```
      <menu name="Visualize Rule/Goal Graph in new Picture"
      module="visur" predicate="visualize_calls" prmtrs="[@2]"
      message="prolog" end_group="off"/>
      <menu name="Edit File with XPCE emacs" module="visur"
      predicate="view_with_emacs_popup" prmtrs="[@2]"
      message="prolog" end_group="off"/>
      <menu name="View Node" module="visur" predicate="view_xml"
      prmtrs="[@2]" message="prolog" end_group="off"/>
    </popup>
  </mouse_click>
  <mouse_click alias="in_out-file">
    <on_click button="left" type="double" module="visur"
    predicate="view_with_emacs_double_click" prmtrs="[@2]"
    message="prolog"/>
    <popup>
      <menu name="Menu File:" predicate="writeln"
      prmtrs="['Menu File']" end_group="on"/>
      <menu name="Cross References" end_group="off">
        <menu name="Incoming and Outgoing File References"
        module="visur" predicate="visualize_calls"
        prmtrs="[@2, file, in_out]" message="prolog" end_group="off"/>
        <menu name="Incoming File References" module="visur"
        predicate="visualize_calls" prmtrs="[@2, file, incoming]"
        message="prolog" end_group="off"/>
        <menu name="Outgoing File References" module="visur"
        predicate="visualize_calls" prmtrs="[@2, file, outgoing]"
        message="prolog" end_group="off"/>
      </menu>
      <menu name="Visualize Rule/Goal Graph in new Picture"
      module="visur" predicate="visualize_calls" prmtrs="[@2]"
      message="prolog" end_group="off"/>
      <menu name="Edit File with XPCE emacs" module="visur"
      predicate="view_with_emacs_popup" prmtrs="[@2]" message="prolog" end_group="off"/>
      <menu name="View Node" module="visur" predicate="view_xml"
      prmtrs="[@2]" message="prolog" end_group="off"/>
    </popup>
  </mouse_click>
  <mouse_click alias="rule_node-rule_node">
    <popup>
      <menu name="Menu Rule Node:" predicate="writeln"
      prmtrs="['Menu Rule Node']" end_group="on"/>
      <menu name="View XML Source Code" module="visur"
      predicate="view_rule" prmtrs="[@2]" message="prolog" end_group="off"/>
      <menu name="View Node" module="visur" predicate="view_xml"
      prmtrs="[@2]" message="prolog" end_group="off"/>
    </popup>
  </mouse_click>
  <mouse_click alias="xpce-built_in">
    <popup>
      <menu name="Menu XPCE Built_In Node:" predicate="writeln"
      prmtrs="['Menu XPCE Built_In Node']" end_group="on"/>
      <menu name="View Listing" module="visur" predicate="view_listing"
      prmtrs="[@2]" message="prolog" end_group="off"/>
      <menu name="View Node" module="visur" predicate="view_xml"
      prmtrs="[@2]" message="prolog" end_group="off"/>
    </popup>
  </mouse_click>
  <mouse_click alias="predicate-predicate">
    <on_click button="left" type="double" module="visur"
    predicate="visualize_calls" prmtrs="[@5, rule_goal]" message="prolog"/>
    <popup>
      <menu name="Menu Predicate Node:" predicate="writeln"
      prmtrs="['Menu Predicate Built_In Node']" end_group="on"/>
      <menu name="Visualizations" end_group="off">
        <menu name="Visualize Rule/Goal Graph in new Picture"
        module="visur" predicate="visualize_calls" prmtrs="[@2, rule_goal]"
        message="prolog" end_group="on"/>
        <menu name="Visualize Predicates calling this Predicate (detailed)"
        module="visur" predicate="visualize_calls" prmtrs="[@2, inv_calls_detailed]"
        message="prolog" end_group="off"/>
        <menu name="Visualize Predicates calling this Predicate (normal)"
        module="visur" predicate="visualize_calls" prmtrs="[@2, inv_calls_normal]"
         message="prolog" end_group="off"/>
      </menu>
      <menu name="Views" end_group="on">
        <menu name="View XML Source Code" module="visur"
        predicate="view_xml_source_code" prmtrs="[@2]" message="prolog" end_group="off"/>
        <menu name="View Listing" module="visur" predicate="view_listing"
        prmtrs="[@2]" message="prolog" end_group="on"/>
        <menu name="View Predicates calling this Predicate" module="visur"
        predicate="view_inverse_calls" prmtrs="[@2]" message="prolog" end_group="on"/>
        <menu name="View Node" module="visur" predicate="view_xml" prmtrs="[@2]"
        message="prolog" end_group="off"/>
      </menu>
    </popup>
  </mouse_click>
  <mouse_click alias="rar-built_in">
    <on_click button="left" type="double" module="visur"
```

```
  predicate="visualize_calls" prmtrs="[@5, rule_goal]" message="prolog"/>
  <popup>
    <menu name="Menu RAR Built_In Node:" predicate="writeln"
    prmtrs="['Menu RAR Built_In Node']" end_group="on"/>
    <menu name="Visualizations" end_group="off">
      <menu name="Visualize Rule/Goal Graph in new Picture"
      module="visur" predicate="visualize_calls" prmtrs="[@2, rule_goal]"
      message="prolog" end_group="on"/>
      <menu name="Visualize Predicates calling this Predicate (detailed)"
      module="visur" predicate="visualize_calls" prmtrs="[@2, inv_calls_detailed]"
      message="prolog" end_group="off"/>
      <menu name="Visualize Predicates calling this Predicate (normal)"
      module="visur" predicate="visualize_calls" prmtrs="[@2, inv_calls_normal]"
      message="prolog" end_group="off"/>
    </menu>
    <menu name="Views" end_group="on">
      <menu name="View XML Source Code" module="visur"
      predicate="view_xml_source_code" prmtrs="[@2]"
      message="prolog" end_group="off"/>
      <menu name="View Listing" module="visur"
      predicate="view_listing" prmtrs="[@2]"
      message="prolog" end_group="on"/>
      <menu name="View Predicates calling this Predicate"
      module="visur" predicate="view_inverse_calls"
       prmtrs="[@2]" message="prolog" end_group="on"/>
      <menu name="View Node" module="visur"
      predicate="view_xml" prmtrs="[@2]"
      message="prolog" end_group="off"/>
    </menu>
  </popup>
</mouse_click>
<mouse_click alias="ordsets-library">
  <on_click button="left" type="double" module="visur"
  predicate="visualize_calls" prmtrs="[@5, rule_goal]"
  message="prolog"/>
  <popup>
    <menu name="Menu library ordsets Node:" predicate="writeln"
    prmtrs="['Menu library ordsets Node']" end_group="on"/>
    <menu name="Visualizations" end_group="off">
      <menu name="Visualize Rule/Goal Graph in new Picture"
      module="visur" predicate="visualize_calls" prmtrs="[@2, rule_goal]"
      message="prolog" end_group="on"/>
      <menu name="Visualize Predicates calling this Predicate (detailed)"
      module="visur" predicate="visualize_calls" prmtrs="[@2, inv_calls_detailed]"
      message="prolog" end_group="off"/>
      <menu name="Visualize Predicates calling this Predicate (normal)"
      module="visur" predicate="visualize_calls" prmtrs="[@2, inv_calls_normal]"
      message="prolog" end_group="off"/>
    </menu>
    <menu name="Views" end_group="on">
      <menu name="View XML Source Code" module="visur"
      predicate="view_xml_source_code" prmtrs="[@2]"
      message="prolog" end_group="off"/>
      <menu name="View Listing" module="visur"
      predicate="view_listing" prmtrs="[@2]"
      message="prolog" end_group="on"/>
      <menu name="View Predicates calling this Predicate"
      module="visur" predicate="view_inverse_calls" prmtrs="[@2]"
      message="prolog" end_group="on"/>
      <menu name="View Node" module="visur" predicate="view_xml"
      prmtrs="[@2]" message="prolog" end_group="off"/>
    </menu>
  </popup>
</mouse_click>
<mouse_click alias="test-meta">
  <popup>
    <menu name="Menu Meta Node:" predicate="writeln"
    prmtrs="['Menu Meta Node']" end_group="on"/>
    <menu name="View XML Source Code" module="visur"
    predicate="view_rule" prmtrs="[@2]"
    message="prolog" end_group="off"/>
    <menu name="View Listing / XML Definition" module="visur"
    predicate="view_listing" prmtrs="[@2]" message="prolog"
    end_group="off"/>
    <menu name="View Node" module="visur" predicate="view_xml"
    prmtrs="[@2]" message="prolog" end_group="off"/>
  </popup>
</mouse_click>
<mouse_click alias="a-meta">
  <popup>
    <menu name="Menu Meta Node:" predicate="writeln"
    prmtrs="['Menu Meta Node']" end_group="on"/>
    <menu name="View XML Source Code" module="visur"
    predicate="view_rule" prmtrs="[@2]" message="prolog" end_group="off"/>
    <menu name="View Listing / XML Definition" module="visur"
    predicate="view_listing" prmtrs="[@2]" message="prolog" end_group="off"/>
    <menu name="View Node" module="visur" predicate="view_xml"
    prmtrs="[@2]" message="prolog" end_group="off"/>
```

```
    </popup>
  </mouse_click>
  <mouse_click alias="b-meta">
    <popup>
      <menu name="Menu Meta Node:" predicate="writeln"
      prmtrs="['Menu Meta Node']" end_group="on"/>
      <menu name="View XML Source Code" module="visur"
      predicate="view_rule" prmtrs="[@2]" message="prolog" end_group="off"/>
      <menu name="View Listing / XML Definition" module="visur"
       predicate="view_listing" prmtrs="[@2]" message="prolog"
       end_group="off"/>
      <menu name="View Node" module="visur" predicate="view_xml"
      prmtrs="[@2]" message="prolog" end_group="off"/>
    </popup>
  </mouse_click>
  <mouse_click alias="node">
    <on_click button="left" type="double" predicate="node_click"
    prmtrs="[@1, @2, @3, @4]" message="prolog"/>
    <on_click button="middle" type="single" predicate="node_click"
    prmtrs="[@1, @2, @3, @4]" message="prolog"/>
    <on_click button="right" type="double" predicate="node_click"
    prmtrs="[@1, @2, @3, @4]" message="prolog"/>
    <popup>
      <menu name="Node 1" predicate="writelq" prmtrs="[@2]"
      message="prolog" end_group="on"/>
      <menu name="Node 2" predicate="writelq" prmtrs="[@1]"
      message="prolog" end_group="off"/>
      <menu name="Node 3" predicate="writelq" prmtrs="[test]"
      end_group="on"/>
    </popup>
  </mouse_click>
    <mouse_click alias="file_file_edge">
      <on_click button="left" type="double" predicate="edge_click"
      prmtrs="[@1, @2, @3, @4]" message="prolog"/>
      <popup>
        <menu name="Menu Edge:" predicate="writeln" prmtrs="['Menu Edge']"
        end_group="on"/>
        <menu name="View Calls" module="visur" predicate="view_calls"
        prmtrs="[@2]" message="prolog" end_group="off"/>
        <menu name="View Edge" module="visur" predicate="view_xml"
        prmtrs="[@2]" message="prolog" end_group="off"/>
      </popup>
    </mouse_click>
    <mouse_click alias="module_module_edge">
      <on_click button="left" type="double" predicate="edge_click"
      prmtrs="[@1, @2, @3, @4]" message="prolog"/>
      <popup>
        <menu name="Menu Edge:" predicate="writeln" prmtrs="['Menu Edge']"
        end_group="on"/>
        <menu name="Visualize" end_group="on">
          <menu name="Visualize Cross Called Files in new Picture"
          module="visur" predicate="visualize_edge_calls" prmtrs="[@2]"
          message="prolog" end_group="on"/>
        </menu>
        <menu name="View">
          <menu name="View Calls" module="visur" predicate="view_calls"
          prmtrs="[@2]" message="prolog" end_group="off"/>
          <menu name="View Edge" module="visur" predicate="view_xml"
          prmtrs="[@2]" message="prolog" end_group="off"/>
        </menu>
      </popup>
    </mouse_click>
    <mouse_click alias="unit_unit_edge">
      <on_click button="left" type="double" predicate="edge_click"
      prmtrs="[@1, @2, @3, @4]" message="prolog"/>
      <popup>
        <menu name="Menu Edge:" predicate="writeln" prmtrs="['Menu Edge']"
        end_group="on"/>
        <menu name="View Calls" module="visur" predicate="view_calls"
        prmtrs="[@2]" message="prolog" end_group="off"/>
        <menu name="View Edge" module="visur" predicate="view_xml"
        prmtrs="[@2]" message="prolog" end_group="off"/>
      </popup>
    </mouse_click>
    <mouse_click ref="../common/menu_picture.xml"/>
    <mouse_click ref="../common/menu_embedded_node.xml"/>
  </mouse_clicks>
</config>
```

List of Figures

1.1 Increasing Number of Lines of Source Code of the Linux Kernel 6
1.2 V-Model 9
1.3 Iterative Development Model 10
1.4 Call Dependencies between Predicates and Files 16

2.1 XPCE Manual 20
2.2 Prolog Navigator 21
2.3 Cross referencer: gxref 21
2.4 Visual Prolog Studio Shell 22
2.5 CIDER 23
2.6 PROLOG Plug-in for Eclipse 24
2.7 PROLOG Refactoring Browser 25

3.1 Software Repository PROSORE 36
3.2 The Structure of PROLOGML 38
3.3 The DTD of the Element <rule> 40
3.4 Usage of the Hierarchy Tree 44
3.5 The Structure of the DDK Hierarchy 76

4.1 Memberships and Dependencies 85
4.2 The Extended Rule/Goal Graph of write_list/1 86
4.3 Rule/Goal Graph 87
4.4 The rule/goal graph of write_list/1 87
4.5 The Rule/Goal Graph of write_list/1 87
4.6 Embedded Call from a Meta-Call Predicate 88
4.7 Comparison of not duplicated and duplicated Meta-Call Predicate 88
4.8 File Dependency Graph of the Module *Slice* 90
4.9 Strongly Connected Components 93
4.10 System Statistics of the DDK 98
4.11 Unit Statistics of the DDK 99
4.12 Statistics about “Undefined Predicates” 101
4.13 Undefined Predicates 102
4.14 Dead Predicates 108
4.15 Excluded Dead Code Predicates 109

4.16 Predicate Statistics 110
4.17 Multifile and Distributed Predicates 111
4.18 File Statistics 113
4.19 Table, showing the Modules consisting of several Files 113
4.20 Rules with Directly Recursive Predicates (in %) 115
4.21 Call Graph of writeln_list/1 118
4.22 Spectrum of a Predicate 119
4.23 Number of Calls of Basic Predicates 121
4.24 Package Association of Basic Predicates 122
4.25 Polar Diagram showing the Package Association of a Predicate 123
4.26 Sorting of Predicates 124
4.27 Distribution of the Predicates in the Called Levels 125
4.28 Distribution of the Predicates of the Called Levels of a Package 126
4.29 Calls of the Units *basic_algebra* and *nm_reasoning* 127
4.30 Calls of the Units *xml* and *databases* 127
4.31 Calls of the Units *source_code_analysis* and *development* 128
4.32 Calls of the Units *stock_tool* and *projects* 128
4.33 Unit Dependency Graph of the DDK in SCAV 128

5.1 Percentage of Use of the Predicates in each Unit and Module of the Sliced Predicate slice/3 136
5.2 Extended Rule/Goal Graph in VISUR 144

6.1 A Graph with Integrated Bar Charts as Node Symbol 151
6.2 File Dependency Graph of the Module Slice: BFS Layout 158
6.3 File Dependency Graph of the Module Slice: DFS Layout 159
6.4 File Dependency Graph of the Module Slice: XPCE Layout 160
6.5 Tree-, Forward-, Back-, and Cross Edges 187
6.6 Merged Nodes 189
6.7 Sending Nodes and Edges Separately to a Picture 198
6.8 A Graph with Three Nodes in an XPCE-Picture 200

7.1 The GUI of VISUR 204
7.2 The Menu *Edit* 205
7.3 The Shortcuts in the Symbol Bar 207
7.4 Menu File 209
7.5 Menu Browser 209
7.6 Menu Database 209
7.7 Menu Visualization 210
7.8 Pop-up Menu of the Hierarchy Browser 210
7.9 Extended Rule/Goal Graph in VISUR 212
7.10 Unit Dependency Graph of DISLOG in VISUR 213
7.11 Incoming Calls of the Module Gxl 216
7.12 Outgoing Calls from the Module 217

7.13 Incoming and Outgoing Calls of the Module 217
7.14 File Dependency Graph of the Module GXL 218
7.15 Calls across Modules 220
7.16 Transitive Derivation Tree of the Diagnosis *P181*. 221
7.17 Visualization of an ER Diagram by VISUR 223
7.18 A DTD visualized in the Hierarchy Browser 224
7.19 A DTD visualized in an XPCE Picture 224

A.1 A Menu Bar Build with xml_menu_to_xpce/3 240
A.2 Hierarchy Browser and File Dependency Graph of the Module basics . . . 241
A.3 The Hierarchy Browser 247
A.4 The Hierarchy Browser Visualizing a GXL Document 249
A.5 The Alias Browser 254
A.6 The Root, Aliases and Alias Icons 254
A.7 The File Icons of the Alias Browser 255

B.1 Different Arrow Types and their Parameters 261
B.2 Different Handles 264

C.1 The Continent, Region, City and Ancestor Relations 269
C.2 The Hierarchy Browser and the Pop-up Menu 272
C.3 The Hierarchy Tree of the Ancestor Relations 275

Index

GXL
- gxl_id_to_xpce_address/3, 196
- gxl_presetting/[2, 3], 155
- gxl_to_edge_types/6, 169
- gxl_to_gxl/[3, 4], 157, 160, 161, 175, 177–182, 185, 187, 192, 193
- gxl_to_missing_nodes/2, 167
- gxl_to_node_types/6, 167
- gxl_to_picture/3, 195
- gxl_to_single_nodes/2, 167
- gxl_to_tree/2, 248
- gxl_to_ugraph/2, 174
- gxl_to_vertices_edges/4, 172
- picture_to_gxl/3, 200
- picture_to_ugraph/2, 201
- picture_to_vertices_edges/3, 201
- ugraph_to_gxl/2, 173
- ugraph_to_picture/3, 199
- vertices_edges_to_gxl/3, 170
- vertices_edges_to_picture/4, 199
- xpce_address_to_gxl_id/2, 196
- xpce_graph_layout/2, 159
- xpce_picture_clear/1, 195

PROSORE
- calls/2, 72, 74, 75
- contains/2, 76, 77
- create_repository/3, 63
- delete/[3, 4], 72
- delete_rules/1, 69
- dislog_sources_to_tree/1, 48
- file_system_to_xml/2, 48
- file_to_directives/2, 59
- insert/[3, 4], 70
- insert_rules/2, 69
- load_rar_db/2, 67
- prologml_to_prolog/1, 43
- replace/[3, 4], 71
- replace_or_insert/[3, 4], 72
- repository_reset/0, 66
- save_rar_db/2, 67
- select/[3, 4], 67
- sources_to_tree/3, 49
- update/[3, 4], 71
- update_repository/0, 69

Source Code Analysis
- add_file_to_alias/2, 251
- alias_browser, 253
- alias_to_file/2, 250
- alias_to_fn_triple/2, 250
- delete_alias/2, 252
- delete_file_from_alias/2, 252
- distributed_predicates/2, 110
- file_called_from_external_ps/2, 90
- file_calls_external_ps/2, 91
- hierarchy_to_browser_ddk/2, 242
- list_dead_code/1, 106
- list_undefined_code/1, 100
- multilevel_predicate/3, 111
- predicate_to_spectrum/2, 118
- predicates_of_type/2, 111
- spectrum_to_picture/1, 119
- table_dead_code/0, 107
- table_file_statistics/0, 112
- table_multifile_predicates/0, 111
- table_predicate_statistics/0, 109
- table_prolog_module_to_files/1, 113
- table_strong_components/0, 92
- table_system_statistics/0, 99
- table_undefined_code/0, 101
- table_units_ps_rules_loc/0, 99

Slicing
  slice/3, 131
  table_slice_statistics/3, 135
  tree_to_statistics_tree/4, 134
VISUR
  dtd_to_browser/1, 224
  dtd_to_gxl/2, 224
  dtd_to_tree/2, 224
  er_diagram_to_gxl/3, 223
  package_to_gxl/3, 213
  visur_gui/0, 203

Bibliography

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

[2] *B. Boehm:* http://sunset.usc.edu/Research_Group/barry.html, 2008.

[3] *I. Bratko:* PROLOG– Programming for Artificial Intelligence, Addison Wesley, 1990.

[4] *J. Baumeister:* Agile Development of Diagnostic Knowledge Systems DISKI 284, IOS Press, 272 pp, ISBN: 978-1-58603-463-4, 2004.

[5] *J. Baumeister, F. Puppe, D. Seipel:* An Agile Process Model for Developing Diagnostic Knowledge Systems, Künstliche Intelligenz 3/2004: Special Issue on "AI and Software Engineering", 12-16, 2004.

[6] *J. Baumeister; D. Seipel; F. Puppe:*, Using Automated Tests and Restructuring Methods for an Agile Development of Diagnostic Knowledge Systems, Proc. of the 17th International Florida Artificial Intelligence Research Society Conference (FLAIRS-2004), AAAI Press, 2004.

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

[8] *W.F. Clocksin, C.S. Mellish:* Programming in PROLOG, Springer, 1987.

[9] *T. DeMarco:* Controlling Software Projects: Management, Measurement & Estimation, Yourdon Press, New York, USA, 1982.

[10] *D. Dochev:* Modellgetriebene Refaktorisierung von PROLOG-Programmen auf verschiedenen Abstraktionsebenen, Diploma Thesis, Universität Würzburg, 2007.

[11] *J. Ellson, S. North:* Graphviz – Graph Visualization Software http://www.graphviz.org/, 2006.

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

[13] *M. Fowler:* Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999, ISBN 0-201-48567-2, http://www.refactoring.com/, 2006.

[14] *Ch. Graiger:* PHPML – Eine XML-Repräsentation zur Analyse von PHP-Quellcode, Diploma Thesis, Universität Würzburg, 2006.

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

[16] *M. Hanus, J. Koj:*
http://www.informatik.uni-kiel.de/ pakcs/cider/, 2008.

[17] *IBM:* The Integrated Development Environment ECLIPSE, http://www.eclipse.org/, 2007.

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

[19] *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/, 2000.

[20] *M. Hopfner:* Eine graphische Oberfläche zur Verwaltung und zum Retrieval von Regeln in deduktiven Datenbanken, Diploma Thesis, Universität Würzburg, 2002.

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

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

[23] *M. Hopfner, D. Seipel, J. Wolff von Gudenberg, G. Fischer:* Reasoning about Source Code in XML-Representation, Workshop on Software-Reengineering, (WSR 2003).

[24] *M. Hopfner, D. Seipel, J. Baumeister:* A PROLOG Tool for Slicing Source Code, Proc. 18th Workshop on Logic Programming, (WLP 2005).

[25] IMAGIX CORPORATION
http://www.imagix.com/

[26] *S. Konst:* Approximate Dynamic Slicing, Institut für Programmiersprachen und Informationssysteme, Abteilung Softwaretechnologie, Studienarbeit, 1999.

[27] *J. Krinke:* Advanced Slicing of Sequential and Concurrent Programs, Universität Passau, Fakultät für Mathematik und Informatik, Dissertation, 2003.

[28] *J. Krinke:* Advanced Slicing of Sequential and Concurrent Programs, GI-Edition – Lecture Notes in Informatics (LNI), pp. 101-110, 2004.

[29] *J. Krinke:* Visualization of Program Dependence and Slices, Proc. International Conference on Software Maintenance, Chicago, pp. 168-177, (ICSM 2004).

[30] *J. Krinke:* Effects of Context on Program Slicing, Journal of Systems and Software 79(9), pp. 1249-1260, 2006.

[31] *J. Krinke:* Empirical Study of Optimization Techniques for Massive Slicing, ACM Transactions on Programming Languages and Systems, 2007.

[32] *S.O. Krumke, H. Noltemeier:* Graphentheoretische Konzepte und Algorithmen, Teubner B.G. GmbH, ISBN: 3-519-00526-3, 2005.

[33] *E. Larsson:*, GXLGraphpad, http://gxl.sourceforge.net/, 2006.

[34] *J. Martin:* Rapid application development, Macmillan Publishing Co., Inc., ISBN 0-02-376775-8, 1991.

[35] *M. Müller:* Refactoring von Source Code mittels logischer Programmierung, Diploma Thesis, University of Würzburg, 2005.

[36] *OWL:* http://www.w3.org/2004/OWL/, 2006.

[37] *OWL:* http://en.wikipedia.org/wiki/Web_Ontology_Language, 2006.

[38] *D. Reitter:*
http://www.david-reitter.com/compling/prolog/compare.html, 2008.

[39] *P.B. Reintjes:* PROLOG for Software Engineering,
http://www.cs.auckland.ac.nz/ j-hamer/07.363/prolog-for-se.html, 2007.

[40] *RuleML:* http://www.ruleml.org, 2007.

[41] *RuleML:* http://en.wikipedia.org/wiki/RuleML, 2006.

[42] *SCAM 2003:* Source Code Analysis and Manipulation,
http://people.brunel.ac.uk/ csstmmh2/scam2003/, 2008.

[43] *SCAM 2006:* Source Code Analysis and Manipulation,
http://www.dcs.kcl.ac.uk/staff/mark/scam2006/, 2008.

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

[45] *T. Schrijvers, A. Serebrenik:* Improving PROLOG Programs: Refactoring for PROLOG, 20th International Conference on Logic Programming,
http://www.cs.kuleuven.be/ toms/vipress/, (ICLP 2004).

[46] *T. Schrijvers, A. Serebrenik, B. Demoen:* Refactoring PROLOG Programs, Technical Report CW373, Department of Computerscience, K.U.Leuven, 2003.

[47] *T. Schrijvers, A. Serebrenik, B. Demoen:* Refactoring PROLOG Code, Proceedings of the 18th Workshop on (Constraint) Logic Programming, (WLP 2004).

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

[49] *D. Seipel:* PL4XML– An SWI-PROLOG Library for XML Data Management, (manual), http://www1.informatik.uni-wuerzburg.de/database/DisLog/fnq_manual.pdf, 2006.

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

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

[52] *D. Seipel:* The DISLOG DEVELOPERS' KIT, http://www1.informatik.uni-wuerzburg.de/database/DisLog/, 2007.

[53] *D. Seipel, M. Hopfner, B. Heumesser:* Analyzing and Visualizing PROLOG Programs based on XML Representations. Proc. International Workshop on Logic Programming Environments, (WLPE 2003).

[54] *D. Seipel, J. Baumeister, M. Hopfner:* Declaratively Querying and Visualizing Knowledge Bases, Workshop on Constraint Logic Programming, (WCLP 2004).

[55] *D. Seipel, H. Thöne:* DISLOG – A System for in Disjunctive Deductive Databases, DAISD 1994: 325-343, 1994.

[56] *J. Siegel:* Introduction to OMG's Unified Modeling Language http://www.uml.org/gettingstarted/what_is_uml.htm, 2008.

[57] *SWRL:* http://www.w3.org/Submission/SWRL, 2006.

[58] *SWRL:* http://en.wikipedia.org/wiki/SWRL, 2006.

[59] *F. Tip:* A survey of program slicing techniques, Journal of Programming Languages 3(3), 121-189, 1995.

[60] *Visual* PROLOG*:* PROLOG Development Center A/S, H.J. Holst Vej 3-5 C, DK-2605 Broendby, Copenhagen, Denmark, http://www.visual-prolog.com/, 2007.

[61] *J. Wielemaker:* SWI-PROLOG 5.0 Reference Manual, http://www.swi-prolog.org/, 2004.

[62] *J. Wielemaker:* XPCE/SWI-PROLOG, XPCE Why?, http://www.swi-prolog.org/packages/xpce/why.html, 2006.

[63] *J. Wielemaker, A. Anjewierden:* Programming in XPCE/PROLOG http://www.swi-prolog.org/, 2004.

[64] *A. Winter, B. Kullbach, V. Riediger:* An Overview of the GXL Graph Exchange Language, LNCS 2269, pp.324-336, http://www.gupro.de/GXL/, 2002

[65] WORLD WIDE WEB consortium (W3C):
http://www.w3.org/XML, 2005.

[66] *V. Wahler, D. Seipel, J. Wolff von Gudenberg, G. Fischer:* Clone Detection in Source Code by Frequent Itemset Techniques, Proceedings of the Source Code Analysis and Manipulation, pp.128-135, (SCAM 2004).

[67] *V. Wahler:* Erkennung von Klonen in Java-Programmen mit Data-Mining-Techniken, Diploma Thesis, University of Würzburg, 2004.

[68] *M. Wetzka:* Ein Tool zur Visualsierung, Analyse und Aufbereitung von SQL-Datenbankanwendungen, Diploma Thesis, University of Würzburg, 2005.

[69] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Call_graph, 2008.

[70] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Prolog, 2008.

[71] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Xml, 2008.

[72] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Software_engineering, 2008.

[73] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/History_of_software_engineering, 2008.

[74] WIKIPEDIA, diverse authors:
http://de.wikipedia.org/wiki/Linux-Kernel, 2008.

[75] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Source_lines_of_code, 2008.

[76] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Repository
http://de.wikipedia.org/wiki/Repository, 2008.

[77] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Software_repository, 2008.

[78] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Design_pattern_(computer_science), 2008.

[79] WIKIPEDIA, diverse authors:
http://de.wikipedia.org/wiki/Slicing, 2008.

[80] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Software_engineering, 2008.

[81] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Software_development_process, 2008.

[82] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Model_driven_development, 2008.

[83] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Software_metric
http://en.wikipedia.org/wiki/Software_package_metrics
http://en.wikipedia.org/wiki/Coupling_%28computer_science%29
http://en.wikipedia.org/wiki/Source_lines_of_code
http://en.wikipedia.org/wiki/Cohesion_%28computer_science%29, 2008.

[84] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/V-Model_%28software_development%29, 2008.

[85] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Iterative_and_incremental_development
http://en.wikipedia.org/wiki/Iterative_development, 2008.

[86] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Agile_software_development, 2008.

[87] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Test_Driven_Development, 2008.

[88] WIKIPEDIA, diverse authors:
http://en.wikipedia.org/wiki/Rapid_application_development, 2008.

[89] *W. Royce:* Managing the Development of Large Software Systems, Proceedings of IEEE WESCON 26, 1970.

[90] *Workshop on Refactoring Tools*:
https://netfiles.uiuc.edu/dig/RefactoringWorkshop/
Berlin, (WRT 2007).