TECHNICAL REPORT IC-PARC-03-2

Developing Applications with ECLiPSe

H. Simonis
Parc Technologies Limited
8th Floor, The Tower Building
11 York Road
London SE1 7NX
email: Helmut.Simonis@parc-technologies.com

IC-PARC
Centre for Planning and Resource Control
William Penney Laboratory
Imperial College London
London
SW7 2AZ

Abstract

In this tutorial we provide a methodology for developing large scale applications with the ECLiPSe system. This methodology follows a top-down approach from the original problem specification to detailed implementation aspects of the resulting code. We concentrate on general design and programming aspects, and largely ignore specific problem solving techniques, which can be found in other tutorials. This current report is intended for programmers developing applications with ECLiPSe as well as those tasked with the maintenance of such programs.
This tutorial is based on an earlier version restricted to internal Parc Technologies use, and contains numerous corrections and improvements suggested by the ECLiPSe team.

Contents

1 Introduction **5**
- 1.1 Intended audience . . . 5
- 1.2 Related work . . . 6
- 1.3 Structure of tutorial . . . 7

2 High Level Design **8**
- 2.1 Application structure . . . 8
- 2.2 Queries . . . 9
- 2.3 API . . . 10
- 2.4 High level structure . . . 12
  - 2.4.1 Classical LSCO structure . . . 12
  - 2.4.2 Data transformation structure . . . 13
  - 2.4.3 Data flow considerations . . . 14
- 2.5 Modules . . . 14
  - 2.5.1 Making predicates available . . . 15
  - 2.5.2 Using packaged functionality . . . 15
- 2.6 Naming schemes . . . 16
  - 2.6.1 Modules and files . . . 16
  - 2.6.2 Predicates . . . 16
  - 2.6.3 Variables . . . 16
- 2.7 Documentation . . . 16
  - 2.7.1 Comment directive . . . 17
  - 2.7.2 Documenting all interfaces . . . 18
  - 2.7.3 Mode declaration . . . 18
  - 2.7.4 Documentation query . . . 19

3 Data Structures **20**
- 3.1 External data representation . . . 20
- 3.2 Internal data representation . . . 22
  - 3.2.1 Named structures . . . 22
  - 3.2.2 Placeholder variables . . . 24

3.2.3 Nested structures vs. key lookup 24
3.2.4 Lists 24
3.2.5 Hash tables 25
3.2.6 Vectors and arrays 25
3.2.7 Multi-representation structures 25

4 Getting it to Work 27
4.1 Stubs 27
4.2 Argument checking 28
4.3 Early testing 28
4.4 Line Coverage 28
4.5 Heeding warnings 28
4.6 Keep it working 29

5 Programming Concepts 30
5.1 Overview 30
5.2 Alternatives 31
5.3 Iteration on lists 34
5.4 Iteration on terms 37
5.5 Iteration on array 39
5.6 Transformation 41
5.7 Filter 43
5.8 Combine 46
5.9 Minimum 48
5.10 Best and rest 49
5.11 Sum 51
5.12 Merge 53
5.13 Group 56
5.14 Lookup 58
5.15 Fill matrix 59
5.16 Cartesian 60
5.17 Ordered pairs 62

6 Input/Output 64
6.1 Reading input data into data structures 64
6.2 How to use DCGs 65
6.3 Creating output data files 69
6.3.1 Creating Prolog data 69
6.3.2 Simple tabular format 69
6.3.3 Using *printf* 70
6.4 Good and bad data formats 71

6.5 Report generation 71

7 If it doesn't Work 73
7.1 Understanding failure 73
7.1.1 Run-time errors 73
7.1.2 Environment limits 74
7.1.3 Failure 75
7.1.4 Wrong answer 75
7.1.5 Missing answer 76
7.2 Checking program points 76
7.3 Debugger 77
7.3.1 Tracing 77
7.3.2 Jumping to a program point 79

8 Correctness and Performance 80
8.1 Testing 80
8.2 Profiling 82
8.3 Reviewing 83
8.4 Issues to check for 84
8.4.1 Unwanted choicepoints 84
8.4.2 Open streams 84
8.4.3 Modified global state 84
8.4.4 Delayed goals 84

A Style Guide 85
A.1 Style rules 85
A.2 Module structure 87
A.3 Predicate definition 88

B Layout Rules 89
B.1 General guidelines 89
B.2 Predicates and clauses 90
B.3 If-then-elses 90
B.4 Disjunctions 90
B.5 Do loops 91
B.6 Comments 91

C Core Predicates 93
C.1 Modules 93
C.2 Predicate definition 93
C.3 Control 93

C.3.1 Do Loops 94
C.4 I/O 94
C.5 Arrays 95
C.6 Hash Tables 95
C.7 Arithmetic 95
C.8 Terms and structures 96
C.9 List Handling 96

Chapter 1

Introduction

This tutorial is an introduction to the design, development, test and maintenance of large scale applications with the ECLiPSe system. It follows a top-down methodology of converting an initial high-level specification into executable code of high quality. We assume that fundamental decisions on selecting the right tool and the right problem solver have already been taken so that we are commited to using ECLiPSe and one of its problem solving libraries. We are basically interested in the engineering aspects of the program development, not its research and development content.
At the same time we provide a number of programming concepts, standard templates that can be used for many ECLiPSe programming tasks. A knowledge of these concepts helps the programmer not only to understand existing code more rapidly, but also should lead to standard solutions whenever a new problem has to be solved.
The tutorial uses excerpts from one of applications developed by Parc Technologies to highlight some of the design choices and also to provide code examples taken from real-life components. This application is centered on IP network analysis, so devices like routers and interfaces will occur throughout the example code. A detailed understanding of the application domain should not be required to understand and re-use the examples.

1.1 Intended audience

The tutorial is aimed at programmers learning how to develop applications with ECLiPSe and tries to codify a 'best practice' for developers. It assumes a basic familarity with concepts of constraint and logic programming and is not a reference manual, so many concepts and language elements are only introduced very briefly. The material also includes some general topics from

software engineering. Knowledge of general OO-related methodologies may be helpful to put that material into context.
The tutorial should not only benefit the designers and developers of ECLiPSe applications, but also those who maintain existing systems. It explains a number of typical design patterns in ECLiPSe programs, and also discusses how to isolate problems and how to make small, incremental changes to a design.

1.2 Related work

Most of the existing textbooks on logic programming carefully describe all language features, but do not provide information how to use these features to obtain particular objectives or how to develop complete systems that can be maintained and extended easily.
The methodology developed in the CHIC-II project looks at a much wider methodology question (http://www.icparc.ic.ac.uk/chic2/methodology/). It not only covers problems like how to select a problem solving technology to fit a particular problem, but also discusses project related issues like team composition, rapid prototyping and development of problem specifications. On the other hand, it does not discuss the details of developing an ECLiPSe application once the main design choices have been taken.
The book "The Craft of Prolog" by Richard O'Keefe is an invaluable source for writing efficient Prolog programs. Many of the techniques and tricks presented are also applicable to writing efficient ECLiPSe programs. The underlying maxim "elegance is not optional" summarizes the spirit of declarative programming. As the book is based on a standard Edinburgh style Prolog, it does not handle issues of module structure and in-line documentation, nor other ECLiPSe specific features.
The ECLiPSe documentation contains most of the information provided in this tutorial, but presents it in a very different way. Invariably, it describes all features of the system, many of which are only required for quite specific (if important) tasks like developing new constraint engines inside the system. It can be difficult to find which parts of the documentation contains important hints to solve a particular problem. On the other hand, it will be useful to look up each feature in the user manual and/or the reference manual as they occur in the tutorial.
The software engineering techniques used in this tutorial are mainly adaptations of OO-development methodologies. This field is far too wide to give specific references here, but the excellent Object Technology Series (http://www.awl.com/cseng/otseries) is a good starting point.

1.3 Structure of tutorial

The tutorial follows a top-down methodology for the design of an application. Chapter 2 discusses general issues of modular design and self-documenting code for ECLiPSe programs. The next chapter on data structures compares different ways of representing data internally and externally, and presents a canonical multi-representation format which allows effective access to data in all parts of an application. Chapter 4 shows how to convert a high level specification into an executable program early in the development stage. The bulk of the tutorial is contained in chapter 5, where we present a number of different programming concepts which can be used as development templates to solve particular problems. This is followed by chapter 6 on input/output, a particularly important aspect of developing extensible programs. The last two chapters deal with debugging (chapter 7) and testing (chapter 8).
There are three appendices to the tutorial. The first one summarizes style rules that good ECLiPSe programs should satisfy. Most of the rules are explained inside the main tutorial text, they are presented here for easy reference. Appendix B gives some rules how ECLiPSe programs should be indented. The last section (appendix C) lists core ECLiPSe predicates which should cover most programming needs and which all ECLiPSe programmers should be familiar with.

Chapter 2

High Level Design

In this chapter we discuss some high-level design decisions which should be clarified before starting the actual development of any code.

2.1 Application structure

We first review the overall application structure found in systems developed at Parc Technologies (at least those using ECLiPSe for part of their development). We distinguish between two application types, one a full application with user-interface, database, reporting, etc, and the other a much smaller system, typically reading data from and to files and performing a single, batch type operation.

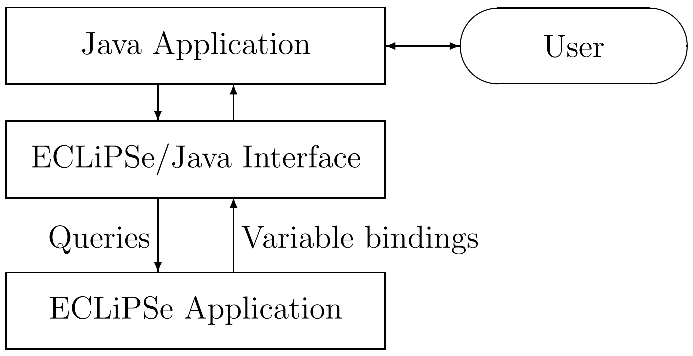


Figure 2.1: Full application

Examples of the first type (see figure 2.1) are Parc Technologies applications (http://www.parc-technologies.com) like AirPlanner and RiskWise[1], where

[1]In the following we use a number of examples from the RiskWise application. It is a

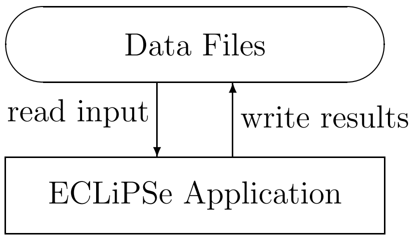


Figure 2.2: Batch type application

everything except the problem solver is developed in Java or related tools. The interface between the main application and the problem solver written in ECLiPSe is via a Java-ECLiPSe interface. In this interface, the main application poses queries for the ECLiPSe solver, passing data and arguments into ECLiPSe. The problem solver then runs the query and returns results as variable bindings in the given query. The Java side only knows about these queries, their data format and the expected results. The internals of the solver, how the queries are resolved, is completely hidden. This defines a nice interface between the application parts, as long as the queries are well defined and documented. Once that design is frozen, the developers for the different parts can continue development independently from each other, using stubs or dummy routines to simulate the other application parts.
The NDI-Mapper in RiskWise is an example of the second application type (see figure 2.2). The application reads some data files (defined in a clear specification), performs some operation on the data and produces results in another set of data files. The top-level query typically just states where the data should be found and where the results should be written to. This batch command then internally calls more detailed routines to load data, etc.

2.2 Queries

For each required function of the interface, we should define a specific query. The query consists of three parts. The first is the predicate name, which obviously should have a relation to the intended function. The second part consists of the input arguments, which are used to pass information from the outside to the problem solver. The structure of these arguments should be as

network analysis tool for IP networks, which uses a constraint solver to determine traffic pattern in the network. If this doesn't make any sense to you, relax. An understanding of the networking application domain is not required to follow this tutorial.

simple as possible, easy to generate and to analyze. The third part consists of the output arguments, which are used to pass information from the problem solver back to the calling interface. When calling the query these arguments will be free variables, which are instantiated inside the solver to some result data structure.
The critical issue in defining the queries lies in identifying which data are required and/or produced, without building an actual implementation of the system. Another issue is the format of the data, which should allow a simple and efficient implementation without extra overhead for data manipulation. It is most important to get this interface right from the start, as any change will create re-work and problems integrating different versions of the software.

2.3 API

The API (application programmer's interface) definition should be finished and signed-off by the designers of the main application and of the problem solver before any code is written[2]. The syntax of each query (predicate) should be defined completely, stating the argument structure and types for both input and outputs. A mode declaration should be used to indicate input and output arguments.
An important part of the API definition is the identification of constraints attached to the data. Constraints in this context are conditions that the data must satisfy in order to obtain meaningful results. These constraints ensure that the data set is complete, minimal and consistent.

complete All data required for the application should be present, with proper links between different records. An example from RiskWise: if an interface of a router is mentioned, then the router to which the interface belongs should be mentioned as well.

minimal The data set should not contain any unneeded information, like duplicated records or references to objects that are not used in the system.

consistent The data should be consistent overall, so that different records can be used without verifying their correctness every time. This can increase processing speed dramatically, and simplifies the code inside the application.

[2]This doesn't mean that the interface can't evolve over time, but at each time point there should be a current, valid definition of the API accepted by all concerned.

Any constraints identified should be added to the specification of a data record in a numbered list, so that they can be identified easily. The example below shows the constraints attached to the

`backbone_line(Node1, Interface1, Node2, Interface2)`

structure in RiskWise.

1. Each pair (Node, Interface) occurs at most once in the entries.
2. There can be no entry starting and ending in the same node.
3. Each node Node1 must have a node/2 fact.
4. Each node Node2 must have a node/2 fact.
5. The interface Interface1 cannot be 'local'.
6. The interface Interface2 cannot be 'local'.
7. Each pair (Node1, Interface1) must have a bandwidth/5 fact.
8. Each pair (Node2, Interface2) must have a bandwidth/5 fact.
9. There can be no backbone line between two net nodes.
10. There is at most one direct connection between two nodes, each line is uniquely described by the pair (Node1, Node2).
11. If there is a connection (Node1, Node2) between two nodes, then we cannot have the inverse connection (Node2, Node1) as well.

For major application interfaces, in particular those linking the Java and the ECLiPSe side, it is recommended to write specific code checking the data for compliance with the constraints. This simplifies the integration task by either pointing out which data do not satisfy the constraints and are therefore invalid, or by guaranteeing that the data passed are correct, and any problem is on the receiving side of the interface.
At this point it is also a good idea to write some dummy batch tests which create the proper interface structures to test the queries. These dummy tests must satisfy the constraints on the data, but do not need to contain useful data. They will be used to exercise the interface and the data constraint checks for coverage with consistent data while integration testing is not yet possible.

2.4 High level structure

Once the external API is clearly defined, we can start looking at the next level of internal structure. This will depend on the intended purpose of the system, but we can identify some typical structures that can be found in many applications. Here, we present two typical designs, one for solving combinatorial problems, the other for transforming data.

2.4.1 Classical LSCO structure

This structure is typical for large scale combinatorial optimization (LSCO) problems. The flow analysis part of RiskWise follows this structure. It consists of five parts, where each performs one particular task.

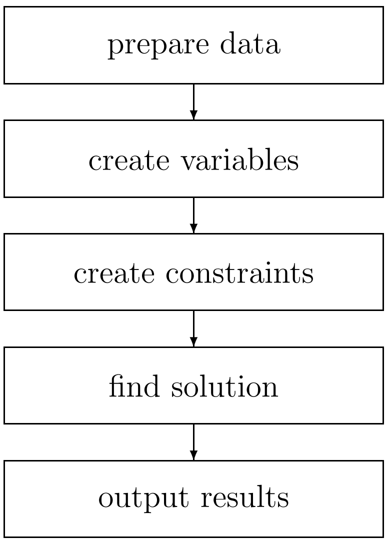


Figure 2.3: LCSO Application structure

prepare data In this module we read the data and prepare the data structures that are required for the problem solver. These data structures should group information that belongs together in one record, and must allow fast access to the data, so that the following application parts can access it easily.

create variables This is followed by the variable generation, where we create all variables that will be used in the solver. These variables will simply be placed in slots already provided in the data structures.

create constraints Once the variables are in place, we can create the constraints between them. This generation should be done in such a way that we can disable or exchange constraints easily.

find solution Having stated all constraints, we can then find a solution (or the optimal solution) to the problem. This will instantiate the problem variables that we have created in step 2.

output results We then have to take the solution and produce the output results, either as data structures, or by creating some result files.

It may not be immediately obvious, but it is very important to clearly separate the different phases. For example, it sometimes may seem faster to generate some variables together with the constraints that are set up between them, as this will save at least one scan of the data structure. But at the same time it makes it much more difficult to disable this constraint or to rearrange the constraints in a different order.
Exceptions to this rule are auxiliary variables which are only introduced to express some constraint and which are not part of the problem variables proper, or constraints that are introduced inside the search process in order to find a solution.

2.4.2 Data transformation structure

The second high-level design is a data transformation structure, used for example in the NDI-Mapper application. It consists of three parts.

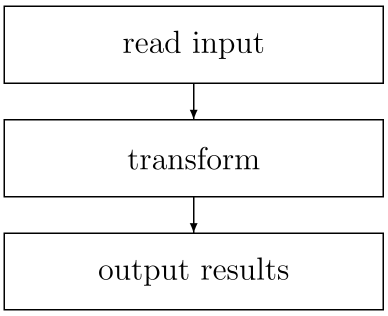


Figure 2.4: Data transformation structure

read input In the first module, we read all data into data structures.

transform We then use these data structures to create new, transformed data in the tranformation part of the application.

output results The last module consists in an output routine, which creates the result data files.

The main limitation of this approach is that all data must be loaded into the application (in main memory). This limits the size of the data sets that can be handled to several megabytes. On the other hand, all information can be made available in an efficient way using lookup hash tables or arrays, so that the implementation of the transformation is quite simple.

2.4.3 Data flow considerations

Once we have decided on a top-level structure, we must consider the input/output arguments of each part. We have to decide which data must be fed into which modules, where new data structures will be created and which other modules require this data. For each piece of information we must identify its source and its possible sinks. Designing these data flows is an iterative process, assigning functionality to modules and making sure that the required information is available. The design aim should be to minimize the amount of information that must be passed across module boundaries and to arrange functions in the correct data flow order, so that all information is produced before it is required in another module.
It can be helpful to draw a data flow graph of the system as a directed graph, showing the main components as nodes and links between sources and sinks of information. A graph of this form should not contain any cycles, and if many links exist between two nodes, we may have to reconsider the split of functionality between.

2.5 Modules

We now have an idea of the overall structure of our application and can now turn this into the top-level code structure. We use the module concept of ECLiPSe to clearly separate the components and to define fixed interfaces between them. Each component of our top-level structure should define a module in one file. We place a *module* directive at the beginning of the file to indicate the module name.
The following examples are taken from the file *flow_prepare_data.ecl*.

```
:-module(flow_prepare_data).
```

2.5.1 Making predicates available

The *export* directive makes some predicate definition available to users outside the module. All other predicates can be only used inside the module, but cannot be used from outside. A module effectively hides all non exported predicates in a separate name space, so that we do not have to worry about using the same name in two different modules.
In our example, the module exports a single predicate *prepare_data/12.*

```
:-export(prepare_data/12).
```

2.5.2 Using packaged functionality

If we want to use some function which has been exported from some module in another module, we have to use the *use_module* directive. It says that all exported predicates of a module should be available in the current module, and ensures that the module is loaded into the system. We can use the same module in more than one place, but the directive makes sure it is loaded only once. The *use_module* directive assumes that the module is defined in a file in the current directory, which has the same name as the module.
The *flow_prepare_data* module uses predicates from a variety of sources.

```
:-use_module(data_topology).
:-use_module(data_peering_policy).
:-use_module(data_traffic).
:-use_module(data_routing).
:-use_module(data_group).
:-use_module(data_general).

:-use_module(flow_statistics).
:-use_module(flow_structures).
:-use_module(flow_utility).
:-use_module(report).
:-use_module(report_utility).
:-use_module(utility).
```

If we want to use predicates defined in some library, we have to use the *lib* directive. It loads the library from the library directory in the ECLiPSe installation, unless we have changed the library path setting to include other directories in the search path. Libraries in ECLiPSe are modules like all others, they are just stored in a different place in the file system.
In our example, a single library called “hash” is loaded.

```
:-lib(hash).
```

2.6 Naming schemes

We now introduce a naming scheme that should be used to define identifiers for the various entities in an ECLiPSe program. A consistent naming scheme makes it easier to maintain and modify the program, and also makes sure that contributions from different programmers can be merged without extra work. The scheme presented here is used for RiskWise.

2.6.1 Modules and files

Each module must have a unique name of the form [a-z][a-z_]+ . There is a one-to-one correspondence of modules and program files.
All program files in an application should be placed in a single directory, with each file named after the module it contains. The extension for ECLiPSe programs is *.ecl*.

2.6.2 Predicates

Predicate names are of form [a-z][a-z_]*[0-9]* . Underscores are used to separate words. Digits should only be used at the end of the name. Words should be English. The top-level query of an application should be named *top*, and should execute a default query with some valid data set. Although predicates in different modules can use the same name, this can be confusing and should be avoided. You should also avoid using the same predicate name with different arity. Although legal, this can create problems when modifying code.

2.6.3 Variables

Variable names are of form [A-Z_][a-zA-Z]*[0-9]* . Separate words start with capital letters. Digits should only be used at the end. Words should be English. If a variable occurs only once in a clause, then its name should start with an underscore. This marks it as a *singleton variable*, and stops compiler warnings about the single occurrence of a variable. If a variable occurs more than once in a clause, then its name should not start with an underscore.

2.7 Documentation

A program that is not documented is not usable. It is very hard to deduce the intention of a program piece just from the implementation, but even a few sentences of explanation can simplify this task dramatically. On the other hand, it is imperative that the documentation and the implementation are consistent. In ECLiPSe we use in-line *comment* directives to integrate the documentation with the program code. This makes it much easier to keep the documentation up to date than with a separate description file.

2.7.1 Comment directive

The *comment* directives are placed in the source code together with other parts of the program. The ECLiPSe system contains some library predicates which extract the comment directives and build a set of interrelated HTML pages for the documentation. The same system is used to generate the reference manual documentation of the library predicates. The *comment* directives come in two flavours. One is a comment for a module, which gives an overview what is module is for. As an example we again use the file *flow_prepare_data.ecl.*

```
:-comment(summary,"This file contains the data preparation for
the flow analysis.").
:-comment(desc,html("This file contains the data preparation for
the flow analysis.
")).
:-comment(author,"R. Rodosek, H. Simonis").
:-comment(copyright,"Parc Technologies Ltd").
:-comment(date,"$Date: 2003/04/29 17:49:48 $").
```

The other form of the *comment* directive is the predicate comment, which describes a particular predicate.

```
:-comment(prepare_data/12,[
summary:"creates the data structures for the flow analysis
",
amode:prepare_data(+,+,+,-,-,-,-,-,-,+,-,-),
args:[
"Dir":"directory for report output",
"Type":"the type of report to be generated",
"Summary":"a summary term",
"Nodes":"a nodes data structure",
"Interfaces":"a interfaces data structure",
"Lines":"a lines data structure",
"Groups":"a groups data structure",
"CMatrix":"a matrix (indexed by nodes) of contributions to
traffic",
"FMatrix":"a matrix (indexed by nodes) of flow variables",
"ScaleFactor":"the scale factor applied to the traffic data",
"Sizes":"the sizes of min, small, large, max packets",
"PMatrix":"The Pi Matrix containing the next hop information,
indexed by node keys"
],
desc:html("
This route creates the data structures for the flow analysis.
...
"),
see_also:[hop/3]
]).
```

The exact form of the different fields is described in the the documentation of the *directives* section of the ECLiPSe built-in predicate documentation in the ECLiPSe reference manual.

2.7.2 Documenting all interfaces

Many predicates in a module only perform very simple tasks which are immediately obvious from the implementation. It would be overkill to document all these predicates. We are working with a rule that all module interfaces must be documented, as well as all predicates which either have a complex implementation or which are expected to be customized by the user.
For predicates without a comment directive, we should use a one line description by a normal ECLiPSe comment in the source code.

2.7.3 Mode declaration

We also use mode declarations to document the calling pattern of predicates. This uses four symbols

+ for arguments which are *instantiated*, i.e. are not free variables, but which may contain variables

++ for arguments that are *ground*, i.e. do not contain variables

- for arguments which are free variables

? for an unknown mode, where we either don't know the mode or we do not care if the argument is instantiated or not

While the compiler uses the mode declarations for code optimization, we basically only use it to document the calling pattern. It allows to isolate

a predicate definition and to understand its purpose without checking all callers[3].
To continue with our example module *flow_prepare_data*, the one exported predicate has the following mode declaration[4].

```
:-mode prepare_data(+,+,+,-,-,-,-,-,-,+,-,-).
```

2.7.4 Documentation query

If a system contains many modules, it can be helpful to provide a query which automatically generates the documentation for all files. In RiskWise, there is a module *document* with an entry point *document/0* which creates the complete documentation tree. It uses the built-in predicates *icompile/1* and *ecis_to_htlms/4* to extract the documentation information from the source files and to build the HTML files required. Whenever we add a new module to the source of the application, we have to add its name into the *components* list.

```
document:-
        components(L),
        (foreach(Module,L) do
           icompile(Module)
        ),
        getcwd(Dir),
        ecis_to_htmls([Dir],'HTML Doc',[],'ApplicationName').


components([
        module1,
        module2,
        ...
        modulen]).
```

[3]Note that the compiler currently does not perform a mode check, i.e. it does not generate compile/runtime errors if a predicate is called with a wrong mode.

[4]We can see that we have not followed the rule to place all input arguments before the output arguments.

Chapter 3

Data Structures

In this chapter we discuss the choice of data structures for the different application parts. Next to the top-level design, this is the most important aspect that must be specified correctly right from the beginning of a project. The wrong choice of a data structure may mean significant re-work in order to change some deficiency later on, while on the other hand a good data structure design can simplify the coding process and can lead to a very efficient implementation.

3.1 External data representation

The first question is how the data will be fed to the application. We can distinguish five alternatives.

arguments In the first alternative, all data are passed in arguments to the query. Multiple items of the same type will usually be represented as lists, with structures to hold different attributes of the different objects. This form has the advantage that each query can be run with a completely new data set without changing the database or creating a new set of files. But debugging data in this form can be more difficult, as there is not direct way to look up some data item. This method also requires work on the Java side to build all the data structures before a call to the ECLiPSe solver. A similar effort is required to develop testing code written in ECLiPSe which exercises the interface.

data files The second alternative is to use data files in a fixed format. The ECLiPSe program then has to read these files and build the internal data structures at the same time. Depending on the format, this may require parsing the input format with definite clause grammars (DCG) (see section 6.2), adding to the development effort[1]. But as the files can be read and written easily, it is quite simple to create test data sets and to analyze problems by hand. The design for the fixed format may require some extra effort if we want to use the full character set for atoms and strings. A proper quoting mechanism may be required in order to distinguish say a comma separator from a comma contained inside a data field.

prolog terms The third alternative is to use data files as before, but to format them as valid Prolog terms that can directly read with the ECLiPSe term I/O predicates. This avoids the overhead of writing parsers in ECLiPSe, but may be difficult for the calling side of the application, unless that is also written in ECLiPSe. Note that we again may face quoting problems, in particular for single and double quotes.

EXDR ECLiPSe also provides a binary data format called EXDR that can be used to exchange information. This can be generated and parsed quite easily in ECLiPSe and in Java, and often allows significant space savings. In addition, problems with quoting are avoided. A disadvantage is that EXDR files are not directly readable by humans, and so may require extra effort during debugging.

facts The last alternative is to store the data as facts in the application. They can then be accessed from any part of the ECLiPSe code quite easily. Testing the code is simple by compiling some data files into the system. The Java interface can also store facts into the database quite easily. But changing the data for a new query can be rather complex, and may require recompiling some data modules.

We should note that instead of using files we can also build queues between the ECLiPSe and the Java parts of the application, avoiding the need for file system space.
Which of these methods should be used? This depends on the application. Passing data as arguments clearly is the cleanest way, but requires significant work on the interface and on code for testing. Using data files in fixed formats is simple if the format is defined correctly, but its use of the file system can cause problems when multiple queries should be run concurrently on the same machine. Using Prolog terms in data files has the same disadvantage, but is

[1]ECLiPSEe 5.4 contains a freeware XML (http://www.xml.org) parser which handles most of the detail of parsing XML files. This makes the use of XML as a data exchange format for ECLiPSe are very exiting new possibility. The parser is described in the "Third Party Library" section of the reference manual.

very simple to use if different ECLiPSe systems exchange data. EXDR files are the safest form to store data, but also the least intuitive. Using queues instead of files avoids problems with multiple instances running at the same time, but require some form of logging to allow debugging. Using facts is a valid alternative if most of the data do not change from one query to the next, but requires extra work to reclaim memory after each change. The following table tries to summarize the advantages and disadvantages of each method.

Property	Argument	Data file	Terms	Facts	EXDR
Multiple runs	++	+	+	-	+
Debugging	-	+	+	++	-
Test generation effort	-	+	+	+	-
Java I/O effort	-	+	-	+	+
ECLiPSe I/O effort	++	+	++	++	++
Memory	++	-	-	- -	-
Development effort	+	-	+	+	-

Table 3.1: Data representation

3.2 Internal data representation

The next question is how the data should be represented inside the application. For this purpose we will have to introduce data structures which allow rapid access to information, which deal with multiple data sets in different parts of the application and where we can add information in a structured way. It should be clear that the built-in fact data base cannot be used for this purpose. Instead, we have to pass the information via arguments of the predicates. In the following sections, we will discuss how the data should be structured to simplify access and coding.
Note that all our data structures use *single assignment*, i.e. there is no destructive assignment in the language[2]. Instead of removing or changing elements of a structuce, we will always make a near-copy with some information being removed or changed.

3.2.1 Named structures

The principal data representation feature of ECLiPSe are named structures. These are terms were each argument is linked to an argument name. We

[2]Destructive assignment in the hash library is hidden from the user.

can access one or more of the arguments with the *with* operator. Named structures are very similar to structures in other languages, the arguments of the structure correspond to attributes of the entity represented. Different attributes can have different types, so that we can store diverse information in a named structure.
In order to use a structure, it must be defined with a *struct* definition. We can define a structure either *local* to a module or *export* the definition so that the same structure can be used in other modules which import the definition. As part of a systematic design we normally create a module which contains nothing but exported structure definitions. This module is then imported with a *use_module* directive in all other modules of the application which use the structures. If a structure is used in one module only, we should define it as a local structure in that module.
We also use comment directives to document the named structures, just like we do for exported predicates. For each attribute name, we define the data type of the attribute. Normally, these will be atomic data types (integer, real, atom, string), but that is not required. The attribute can hold any data type that we can pass as an argument to a predicate.
As an example of a named structure we use a small part of the RiskWise module *flow_structures*.

```
:-comment(struct(group),[
summary:"
this data structure describes the group object
",
fields:[
"name":"atom, name of the group",
"type":"atom, one of pop, interconntion, vpn or tariff_class",
"index":"integer, unique index of the group",
"list":"list of interface indices belonging to the group",
"nodes":"list of nodes which contain interfaces of that group"
]
]).
:-export struct(group(name,
                      type,
                      index,
                      list,
                      nodes)).
```

3.2.2 Placeholder variables

If we do not specify a fixed attribute value when the named structure is created, then its value will be a free variable which can be bound later on. This is useful for two main purposes. On one side we can define attributes of a structure which will hold the constraint variables of a problem, on the other side we can leave some attributes initially unassigned so that we can fill these slots with results of a computation later on.

3.2.3 Nested structures vs. key lookup

A very common data representation problem is how to access information about some structure from another structure, for example in RiskWise how to access the information about a router from an interface of the router. There are two main alternatives. The first is to insert the data of the first entity (router) directly in the representation of the second entity (interface) as an additional attribute, the second is to store a key which can be used to look up the entity. Although the first method has the advantage of avoiding the extra lookup, we do not recommend this approach. If we have recursive references to objects (in our example above if the router also contains a link to all its interfaces) then this direct representation becomes an infinite data structure, which causes problems for printing and debugging. If we use the second approach, we obviously need a way to find the entity belonging to a particular key without too much overhead. The choice of the key depends on the representation of our overall data structure, which we will discuss in the next sections.

3.2.4 Lists

A natural way to represent a collection of items of the same type is to use lists. They are very convenient to handle an arbitrary number of items by iterating on successive heads of the list, until the empty list is reached. Unfortunately, finding a particular item in a list is a very expensive operation, as we have to scan the list sequentially.
We should never use a list when we can use a structure instead. If we know that a collection will always have the same number of items (say 3), it is much better to use a structure with that number of arguments than to use a list.

3.2.5 Hash tables

Hash tables are a very useful alternative to lists, if we sometimes want to look up items rather than iterate over all of them. They are defined in the library *hash*. We can add items one by one, without an a priori limit on the number of items. As key we can use numbers, atoms or arbitrary terms, but atoms would be the most common key in a hash table. This is very useful when converting input data, since the external data representation often will use names (atoms) to identify objects.
While it is possible to iterate over all items of a hash table, this is not as simple as iteration over a list or an array.

3.2.6 Vectors and arrays

Vectors are another way to represent a collection of items. Each item is associated with an integer key in the range from 1 to N, where N is the size of the vector. Unfortunately, the value N must be known a priori, when we first create the vector. Accessing individual entries by index is very fast, and iterating over all entries is nearly as simple as for lists. The main drawbacks of a vector representation are that we have to know the total number of items beforehand and that the keys must be consecutive integers in the range from 1 to N.
Multi-dimensional arrays are simple nested vectors, they are created with the *dim* predicate for a given dimension and size. Access to an element is with the *subscript* predicate (see section 5.5 for an example).

3.2.7 Multi-representation structures

Each of the alternative representations given above has some advantages and disadvantages. To obtain a very flexible representation, we can choose a multi-representation structure. In this structure, a collection of items is represented as a list and as a hash table and as an array. The list representation can be used for a very simple iteration over all items, the hash table is used in the initial data input phase to find items with a given name and the array of items is used in the core routines of the solver for the fastest access by an integer index.
The memory requirements of this multi-representation scheme are quite low. The storage to hold the items themselves is shared for all representations, we only need the additional space for the list, hash table and array structures.
In RiskWise, we use the multi-representation scheme for most data structures, with special access predicates like *find_interface/3* to access items with

either numeric indices or atom keys. References from one entity to another are by integer key, e.g. each interface structure contains as an attribute the integer key value of the node (router) to which it belongs.

Chapter 4

Getting it to Work

Once the initial design is finished, and the top-level structure of the program has been defined, we should convert this specification into workable code as quickly as possible. Problems with the parameter passing, missing information or a wrong sequence of the data flow can be detected much more easily this way. We propose to use stubs and dummy code to get an (incomplete) implementation directly from the specification.

4.1 Stubs

Each query has been defined in form of a predicate, with input and output parameters. Regardless of the actual function of the query, it is easy to generate a predicate which syntactically behaves like the finished program. It reads the input parameters and creates output terms which satisfy the specification. Initially, it does not matter if the output parameters are not consistent with the input values, as long as their form is correct.
If a top-level query has already been revised into smaller components, we can immediately write the body of the top-level predicate calling the individual components in the right order and with the correct parameters. Adding stub definitions of these components again leads to an executable program.
Whenever we finish development of some of the components, we can immediately replace the stub code with the working implementation. Provided that the inputs are sufficiently simple, we get a simulated version of our application that we can convert piece by piece into the real application.

4.2 Argument checking

It is a good idea, at least for the top-level queries, to verify all parameters systematically. In the specification, we have defined various constraints that the input data must satisfy. Most of these constraints can be translated without too much work into checks that verify the constraints. A separate module for error checking can handle this work and leave the application core to rely on the correctness of the data.
In RiskWise, the module *error_checking* performs these checks, using a simple language to define data constraints into executable rules.

4.3 Early testing

Experience has shown that the testing and tuning of an application are by far the most time consuming activities in the development of a LSCO system. It is very important that we prepare test data sets as early as possible, together with some test routines which exercise our API queries with these tests.
If we use different test sets right from the start on our stub implementation, then we can detect problems early on during the development of individual components.

4.4 Line Coverage

Another tool that is very useful at this stage of development is the line coverage profiler in the *coverage* library. Running this tool on the stub implementation we can check that each piece of code is exercised by our test sets.

4.5 Heeding warnings

When we load our first implementation into the ECLiPSe system, it is quite possible that we find a number of error and warning messages. Errors will usually be caused by simple syntax problems, by forgetting to define some predicate or by not importing a module where it is required. These errors are typically easy to fix once we understand which part of the program is responsible.
It is tempting to ignore warnings in order to get the code running as quickly as possible. That would be a big mistake. We should eliminate all warnings about singleton variables and missing predicate definitions before continuing.

Not only will this lead to the detection of problems in the code at this point, we will also immediately see if new warnings are introduced when we change some part of the program.

4.6 Keep it working

As a general rule, once we have created a working stub system, we should always keep the program working by making changes in small increments and testing after each change. This way we know which part of the program was modified last and which is therefore most likely to cause the problem.

Chapter 5

Programming Concepts

5.1 Overview

In this chapter we will present typical programming concepts in ECLiPSe with example uses in the RiskWise application. These programming concepts each perform one particular operation in an efficient way, and show how these tasks should be programmed in ECLiPSe. They can be adapted to specific tasks by adding additional parameters, changing calls inside the concepts or passing different data structures.
The presentation of the concepts follows the same pattern: We first describe the concept in general terms and then present the parameters required. This is followed by one or several implementations of the concept in ECLiPSe, and some larger examples from the RiskWise code.

5.2 Alternatives

Description This concept is used to choose between alternative actions based on some data structure. For each alternative, a guard *q_i* is specified. The guard is a test which succeeds if the condition for selecting one alternative is met. The actions *r_i* are executed when the guard succeeds. In order to choose only the right alternative, and not to leave any unwanted choicepoints in the execution, we must eliminate the remaining alternatives after the guard succeeds. For this we use a cut (!) after each guard but the last. We can leave out the cut after the last guard, as there are no choices left at this point.

Parameters

X a data structure

Schema

```
:-mode alternatives(+).
alternatives(X):-
        q1(X),
        !,
        r1(X).
alternatives(X):-
        q2(X),
        !,
        r2(X).
alternatives(X):-
        qn(X),
        rn(X).
```

Comments Very often, other parameters must be passed either to the guards, or to the actions.
The errors which are introduced if a cut to commit to a choice is left out are very hard to debug, and may only show after long execution. Much better to always cut after each guard.
When adding new parameters it is important to ensure that they are added to all clauses of the predicate. If a parameter is not used in some clause, then it should be added as a singleton variable. If we miss an argument on one of the clauses in the middle, the compiler will create an error message about *non consecutive clauses*. But if we miss an argument for either the

first or the last clause, the compiler will just treat this as another predicate definition with the same name, but a different arity. Errors of this form are very hard to spot.

Example

```
:-mode interface_type(+,+,-).
interface_type(_Node,local,local):-
        !.
interface_type(Node,_Interface,backbone_net):-
        node(Node,net),
        !.
interface_type(Node,Interface,backbone):-
        backbone_line(Node,Interface,_,_),
        !.
interface_type(Node,Interface,backbone):-
        backbone_line(_,_,Node,Interface),
        !.
interface_type(Node,Interface,interconnection):-
        group(interconnection,_,Node,Interface),
        !.
interface_type(_Node,_Interface,customer).
```

Here we branch on information passed in the first two arguments, and return a result in the last argument. The last clause is a default rule, saying that the interface type is *customer*, if none of the other rules applied.
Some programmers perfer to make the output unification explicit, like so:

```
:-mode interface_type(+,+,-).
interface_type(_Node,local,Result):-
        !,
        Result = local.
interface_type(Node,_Interface,Result):-
        node(Node,net),
        !,
        Result = backbone_net.
interface_type(Node,Interface,Result):-
        backbone_line(Node,Interface,_,_),
        !,
        Result = backbone.
interface_type(Node,Interface,Result):-
        backbone_line(_,_,Node,Interface),
        !,
        Result = backbone.
interface_type(Node,Interface,Result):-
        group(interconnection,_,Node,Interface),
        !,
        Result = interconnection.
interface_type(_Node,_Interface,Result):-
        Result = customer.
```

This has advantages if the predicate may be called with the last argument instantiated.

5.3 Iteration on lists

Description This concept is used to perform some action on each element of a list. There are two implementations given here. The first uses the *do* loop of ECLiPSe, the second uses recursion to achieve the same purpose. In the *do* loop, the *foreach* keyword describes an action for each element of a list. The first argument (here X) is a formal parameter of the loop. At each iteration, it will be bound to one element of the list. The second argument is the list over which we iterate.
It is a matter of style whether to use the first or second variant. For simple iterations, the *do* loop is usually more elegant. We can also often use it *inline*, and avoid introducing a new predicate name just to perform some iteration.

Parameters

L a list

Schema

```
/* version 1 */

:-mode iteration(+).
iteration(L):-
        (foreach(X,L) do
            q(X)
        ).

/* version 2 */

:-mode iteration(+).
iteration([]).
iteration([H|T]):-
        q(H),
        iteration(T).
```

Comments If we want to scan several lists in parallel, we can use multiple *foreach* statements in the same *do* loop. The following code fragment calls predicate q for the first elements of list L and K, then for the second elements, etc.

```
:-mode iteration(+,+).
iteration(L,K):-
        (foreach(X,L),
         foreach(Y,K) do
            q(X,Y)
        ).
```

This requires that the lists are of the same length, otherwise this *do* loop will fail.
Note that we can put as many parallel operations into a *do* loop as we want, they are all executed inside one big loop. We can of course also nest *do* loops so that one loop is executed inside another loop.
The *foreach* operator can also be used to create a list in a *do* loop. This is shown in the transformation concept.
Very often, we have to pass additional parameters into the *do* loop. We do this with the *param* parameter, which lists all variables from outside the loop that we want to use inside the loop. A variable which is not mentioned as a *param* argument, is unbound inside the loop. Normally, this will create a warning about a singleton variable inside a *do* loop. The following code fragment shows the use of *param* to pass variables A, B and C to the call of predicate q.

```
:-mode iteration(+,+,+,+).
iteration(L,A,B,C):-
        (foreach(X,L),
         param(A,B,C) do
            q(X,A,B,C)
        ).
```

Example

```
% set the group fields inside the interfaces for each interface
:-mode set_group_of_interfaces(+,+).
set_group_of_interfaces(L,Interfaces):-
        (foreach(group with [type:Type,
                             name:Name,
                             interface:I],L),
         param(Interfaces) do
            find_interface(I,Interfaces,Interface),
            set_group_of_interface(Type,Name,Interface)
        ).
```

Here we use the information that each member of the list L is a term *group/4* to replace the formal parameter with a term structure where we access individual fields directly. Also note that the body of the loop may contain more than one predicate call.

5.4 Iteration on terms

Description We can iterate not only over all elements of a list, as in the previous concept, but also over all arguments of a term. Obviously, this only makes sense if all arguments of the term are of a similar type i.e. the term is used as a *vector*. The *foreacharg* keyword of the *do* loop iterates over each argument of a term.

Parameters

T a term

Schema

```
:-mode iteration(+).
iteration(T):-
        (foreacharg(X,T) do
            q(X)
        ).
```

Comments We can use multiple *foreacharg* keywords to scan multiple vectors at the same time, but we cannot use *foreacharg* to create terms (we do not know the functor of the term). If we want to create a new term, we have to generate it with the right functor and arity before the *do* loop. The following code segment performs vector addition $\vec{C} = \vec{A} + \vec{B}$.

```
:-mode vector_add(+,+,-).
vector_add(A,B,C):-
        functor(A,F,N),
        functor(C,F,N),
        (foreacharg(X,A),
         foreacharg(Y,B),
         foreacharg(Z,C) do
           Z is X + Y
        ).
```

If the terms A and B do not have the same number of arguments, the predicate will fail.

Example

```
:-mode interface_mib_add(+,+,-).
interface_mib_add(A,B,C):-
        C = interface_mib with [],
        (foreacharg(A1,A),
         foreacharg(B1,B),
         foreacharg(C1,C) do
            C1 is A1 + B1
        ).
```

This predicate adds vectors with the functor *interface_mib* and returns such a vector.

5.5 Iteration on array

Description The next concept is iteration on an array structure. We often want to perform some action on each element of a two-dimensional array. Again, we present two implementations. The first uses nested *foreacharg do* loops to perform some operation q on each element of an array. The second uses nested *for* loops to iterate over all index combinations I and J. This second variant is more complex, and should be used only if we require the index values I and J as well as the matrix element X.

Parameters

Matrix a matrix

Schema

```
/* version 1 */

:-mode iteration(+).
iteration(Matrix):-
        (foreacharg(Line,Matrix) do
            (foreacharg(X,Line) do
                q(X)
            )
        ).

/* version 2 */

:-mode iteration(+).
iteration(Matrix):-
        dim(Matrix,[N,M]),
        (for(I,1,N),
         param(M,Matrix) do
            (for(J,1,M),
             param(I,Matrix) do
                subscript(Matrix,[I,J],X),
                q(X,I,J)
            )
        ).
```

Comments The *dim* predicate can not only be used to create arrays, but also to find the size of an existing array.
Note the strange way in which parameters M, I and $Matrix$ are passed through the nested *for* loops with *param* arguments. But if we do not do this, then the variable $Matrix$ outside and inside the *do* loop are unrelated.

Example The example calls the predicate *fill_empty/3* for each index combination of entries in a matrix $PMatrix$.

```
:-mode fill_rest_with_empty(+,+).
fill_rest_with_empty(N,PMatrix):-
        (for(I,1,N),
         param(PMatrix,N) do
            (for(J,1,N),
             param(PMatrix,I) do
                fill_empty(PMatrix,I,J)
            )
        ).
```

5.6 Transformation

Description This next concept is used to perform some transformation on each element of a list and to create a list of the transformed elements. At the end, both lists will have the same length, and the elements match, i.e. the first element of the second list is the transformed first element of the first list.
This concept uses the *foreach* keyword in two different ways. The first is used to scan an existing list L, the second is used to construct a list K as the result of the operation.

Parameters

L a list

K a free variable, will be bound to a list

Schema

```
:-mode transformation(+,-).
transformation(L,K):-
        (foreach(X,L),
         foreach(Y,K) do
            q(X,Y)
        ).
```

Comments In the code above we cannot see that list L is an input and list K is an output. This can only be deduced from the calling pattern or from the mode declaration.

Example The example takes a list of *router_mib_data* terms and builds a list of temporary *t/2* terms where the second argument consists of *router_mib* terms.

```
:-mode convert_to_router_mib(+,-,-).
convert_to_router_mib(L,K,Router):-
        (foreach(router_mib_data with
                 [router:Router,
                  time:Time,
                  tcp_segm_in:A,
                  tcp_segm_out:B,
                  udp_datagram_in:C,
                  udp_datagram_out:D],L),
         foreach(t(Time,router_mib with
                   [tcp_segm_in:A,
                   tcp_segm_out:B,
                   udp_datagram_in:C,
                   udp_datagram_out:D]),K),
         param(Router) do
            true
         ).
```

In this example the transformation is completely handled by matching arguments in the *foreach* statements. We use the predicate *true* for an empty loop body.
Figuring out what is happening with the variable *Router* is left as an exercise for the advanced reader.

5.7 Filter

Description The filter concept extracts from a list of elements those that satisfy some condition q and returns a list of these elements.
We present three implementations, one using recursion, the others using a *do* loop with the *fromto* keyword.

Parameters

L a list

K a free variable, will be bound to a list

Schema

```
/* version 1 */

:-mode filter(+,-).
filter([],[]).
filter([A|A1],[A|B1]):-
        q(A),
        !,
        filter(A1,B1).
filter([_A|A1],B1):-
        filter(A1,B1).

/* version 2 */

:-mode filter(+,-).
filter(L,K):-
        (foreach(X,L),
         fromto([],In,Out,K) do
            q(X,In,Out)
        ).

q(X,L,[X|L]):-
        q(X),
        !.
q(_,L,L).
```

```
/* version 3 */

:-mode filter(+,-).
filter(L,K):-
        (foreach(X,L),
         fromto(K,In,Out,[]) do
            q(X,In,Out)
        ).

q(X,[X|L],L):-
        q(X),
        !.
q(_,L,L).
```

Comments The difference between versions 2 and 3 lies in the order of the elements in the result list. Version 2 produces the elements in the inverse order of version 1, whereas version 3 produces them in the same order as version 1. This shows that the *fromto* statement can be used to build lists forwards or backwards. Please note that the predicate *q/3* is also different in variants 2 and 3.
The cuts (!) in the program clauses are very important, as they remove the possibility that a selected element is not included in the filtered list. If we remove the cuts, then the *filter* predicate has an exponential number of "solutions". Only the first solution will be correct, on backtracking we will decide to reject elements which satisfy the test criterion and we will explore all combinations until we reach the empty list as the last "solution".

Example The following program is used to extract interfaces related to customers (types customer, selected and remaining) as a list of *customer/3* terms, group them by node and perform some action on each group.

```
:-mode selected_min_max(+,+).
selected_min_max(Type,Interfaces):-
        Interfaces = interfaces with list:List,
        (foreach(Interface,List),
         fromto([],In,Out,Customers) do
            selected_customer(Interface,In,Out)
        ),
        sort(0,=<,Customers,Sorted),
        customers_by_node(Sorted,Grouped),
        selected_together(Type,Grouped,Interfaces).

selected_customer(interface with [type:Type,
                                  index:I,
                                  node_index:Node],
                  In,
                  [customer with [node:Node,
                                  index:I,
                                  type:Type]|In]):-
        memberchk(Type,[customer,selected,remaining]),
        !.
% all other types: backbone,backbone_net,interconnection,local
selected_customer(_,In,In).
```

5.8 Combine

Description This concept takes a list, combines consecutive elements according to some criterion and returns a list of the combined elements.
The typical use of this concept will first sort the input list so that elements that can be combined are consecutive in the list.

Parameters

L a list

Res a free variable, will be bound to a list

Schema

```
:-mode combine(+,-).
combine([],[]).
combine([A,B|R],Res):-
        can_combine(A,B,C),
        !,
        combine([C|R],Res).
combine([A|A1],[A|Res]):-
        combine(A1,Res).
```

Comments It is important to note that the recursive call in the second clause continues with the combined element C, since it may be combined with more elements of the rest of the list R.
The cut in the second clause ensures that elements that can be combined are always combined, and that we do not leave a choice point in the execution.
The most simple use of the concept is the removal of duplicate entries in a sorted list.

Example

```
:-mode combine_traffic(+,-).
combine_traffic([],[]).
combine_traffic([A,B|R],L):-
        try_to_combine(A,B,C),
        !,
        combine_traffic([C|R],L).
combine_traffic([A|R],[A|S]):-
        combine_traffic(R,S).

try_to_combine(interface_traffic_sample(Time,Router,Interface,
                                        X1,X2,X3,X4,X5,
                                        X6,X7,X8,X9,X10),
        interface_traffic_sample(Time,Router,Interface,
                                 Y1,Y2,Y3,Y4,Y5,
                                 Y6,Y7,Y8,Y9,Y10),
        interface_traffic_sample(Time,Router,Interface,
                                 Z1,Z2,Z3,Z4,Z5,
                                 Z6,Z7,Z8,Z9,Z10)):-
        Z1 is X1+Y1,
        Z2 is X2+Y2,
        Z3 is X3+Y3,
        Z4 is X4+Y4,
        Z5 is X5+Y5,
        Z6 is X6+Y6,
        Z7 is X7+Y7,
        Z8 is X8+Y8,
        Z9 is X9+Y9,
        Z10 is X10+Y10.
```

Here we combine traffic samples for the same interface and time point by adding the sample values $X\_1...X\_10$ and $Y\_1...Y\_10$. The predicate *try_to_combine* will only succeed if the two input arguments have the same time stamp, router and interface, but it will fail if the arguments differ on these fields.
Also note that we do not use named structures in this example. This is justified as any extension of the structure would probably entail a change of the program anyway.

5.9 Minimum

Description This concept selects the smallest element of a list according to some comparison operator *better*.

Parameters

L a list

V a free variable, will be bound to an entry of *L*

Schema

```
:-mode minimum(+,-).
minimum([H|T],V):-
        (foreach(X,T),
         fromto(H,In,Out,V) do
            minimum_step(X,In,Out)
        ).

minimum_step(X,Old,X):-
        better(X,Old),
        !.
minimum_step(X,Old,Old).
```

Comments This implementation of minimum fails if the input list has no elements. This means that somewhere else in the program we have to handle the case where the input list is empty. This seems to be the most clear definition of minimum, an empty list does not have a smallest element.
If several elements of the list have the same minimal value, only the first one is returned.

5.10 Best and rest

Description This concept is an extension of the minimum concept. It not only returns the best element in the input list, but also the rest of the original list without the best element. This rest can then be used for example to select another element, and so on.

Parameters

L a list

Best a free variable, will be bound to an entry of *L*

Rest a free variable, will be bound to a list of the entries of *L* without *Best*

Schema

```
:-mode best_and_rest(+,-,-).
best_and_rest([H|T],Best,Rest):-
        (foreach(X,T),
         fromto(H,In1,Out1,Best),
         fromto([],In2,Out2,Rest) do
            best(X,In1,Out1,In2,Out2)
        ).

best(X,Y,X,L,[Y|L]):-
        better(X,Y),
        !.
best(X,Y,Y,L,[X|L]).
```

Comments The predicate fails if the input list is empty. We must handle that case somewhere else in the program.
If several elements of the list have the same best value, only the first one is selected.
The order of elements in *Rest* may be quite different from the order in the input list. If that is not acceptable, we must use a different implementation. A rather clever one is given below:

```
best_and_rest([First|Xs],Best,Rest):-
        (foreach(X,Xs),
         fromto(First,BestSoFar,NextBest,Best),
         fromto(Start,Rest1,Rest2,[]),
         fromto(Start,Head1,Head2,Gap),
         fromto(Rest,Tail1,Tail2,Gap) do
             (better(X,BestSoFar) ->
                 NextBest = X,
                 Tail1 = [BestSoFar|Head1],
                 Tail2 = Rest1,
                 Head2 = Rest2
             ;
                 NextBest = BestSoFar,
                 Tail2 = Tail1,
                 Head2 = Head1,
                 Rest1 = [X|Rest2]
             )
        ).
```

5.11 Sum

Description The sum concept returns the sum of values which have been extracted from a list of data structures. It uses a *foreach* to scan each elements of the list and a *fromto* to accumulate the total sum.

Parameters

L a list

Sum a free variable, will be bound to a value

Schema

```
:-mode sum(+,-).
sum(L,Sum):-
        (foreach(X,L),
         fromto(0,In,Out,Sum) do
            q(X,V),
            Out is In+V
        ).
```

Comments The initial value for the sum accumulator here is 0. We could use another initial value, this can be useful if we want to obtain the total over several summations.

Example The program counts how many entries in the *interface_mib_data* list refer to active interfaces (octet count non-zero).

```
:-mode non_zero_measurements(+,-).
non_zero_measurements(L,N):-
        (foreach(X,L),
         fromto(0,In,Out,N) do
            non_zero_measurement(X,In,Out)
        ).

non_zero_measurement(interface_mib_data with [octet_in:A,
                                              octet_out:B],
                     In,Out):-
        A+B > 0,
        !,
        Out is In+1.
non_zero_measurement(_X,In,In).
```

5.12 Merge

Description The merge concept is used when we want to match corresponding entries in two lists. We sort both lists on the same key, and then scan them in parallel, always discarding the entry with the smallest key first. We can use this concept to combine information from the two lists, to find differences between lists quickly, or to lookup information from the second list for all elements of the first list.

Parameters

L a list

K a list

Schema

```
:-mode merge(+,+).
merge(L,K):-
        sort_on_key(L,L1),
        sort_on_key(K,K1),
        merge_lp(L1,K1).

merge_lp([],_):-
        !.
merge_lp([_|_],[]):-
        !.
merge_lp([A|A1],[B|B1]):-
        merge_compare(A,B,Op),
        merge_cont(Op,A,A1,B,B1).

merge_cont(<,A,A1,B,B1):-
        merge_lp(A1,[B|B1]).
merge_cont(=,A,A1,B,B1):-
        merge_op(A,B),
        merge_lp(A1,[B|B1]).
merge_cont(>,A,A1,B,B1):-
        merge_lp([A|A1],B1).
```

Comments The cuts in *merge_lp* are used to remove choicepoints left by the compiler[1].
The schema looks quite complex, but its performance is nearly always significantly better than a simple lookup in the second list.

[1]As ECLiPSe only uses indexing on a single argument, the compiler cannot recognize that the clause patterns are exclusive.

Example The example takes data from two different sources and merges it. The first argument is a list of *interface_topology* terms, the second a list of *ndi_interface* structures. For matching $Node - Interface$ pairs, we copy information from the first structure into the second. If the $Node-Interface$ pairs do not match, then we don't do anything.
Also note the use of *compare/3* to obtain a lexicographical ordering of $Node-Interface$ pairs.

```
:-mode insert_topology(+,+).
insert_topology([],_):-
        !.
insert_topology([_|_],[]):-
        !.
insert_topology([A|A1],[B|B1]):-
        compare_index_interface(A,B,Op),
        insert_topology_op(Op,A,B,A1,B1).

compare_index_interface(interface_topology(_,Router1,
                                           Index1,_,_),
                  ndi_interface with [router:Router2,
                                      index:Index2],Op):-
        compare(Op,Router1-Index1,Router2-Index2).

insert_topology_op(<,A,B,A1,B1):-
        insert_topology(A1,[B|B1]).
insert_topology_op(=,A,B,A1,B1):-
        insert_one_topology(A,B),
        insert_topology(A1,B1).
insert_topology_op(>,A,_B,A1,B1):-
        insert_topology([A|A1],B1).

insert_one_topology(interface_topology(_,_,_,Ip,Mask),
                    ndi_interface with [ip_address:Ip,
                                        subnet_mask:Mask,
                                        subnet:Subnet]):-
        subnet(Ip,Mask,Subnet).
```

5.13 Group

Description This concept takes a sorted list of items and creates a list of lists, where items with the same key are put in the same sub-list. This works even for the empty input list.
The second argument of *group_lp* serves as an accumulator to collect items with the same key. As long as the next item uses the same key, it is put into this accumulator (2nd clause). If the remaining list is empty (1st clause) or it starts with an element of a different key (3rd clause), the accumulated list is put into the output list.

Parameters

L a list

K a free variable, will be bound to a list

Schema

```
:-mode group(+,-).
group([],[]).
group([H|T],K):-
        group_lp(T,[H],K).

group_lp([],L,[L]).
group_lp([H|T],[A|A1],K):-
        same_group(H,A),
        !,
        group_lp(T,[H,A|A1],K).
group_lp([H|T],L,[L|K]):-
        group_lp(T,[H],K).
```

Comments The order of items in the resulting sub lists is the reverse of their order in the initial list.
The order of the sub lists in the result is the same as the order of the keys in the original list.
If the initial list is not sorted by the same key that is used in *same_group*, then this program does not work at all.

Example The following program takes a list of terms and groups them according to some argument number N. It returns a list of *group/2* terms, where the first argument is the common key in the group, and the second argument is a list of all terms which share that key.

```
:-mode group(+,+,-).
group([],_,[]):-
        !.
group(Terms,N,Grouped):-
        sort(N,=<,Terms,[H|T]),
        arg(N,H,Group),
        group1(T,Group,[H],N,Grouped).

group1([],Group,L,_,[group(Group,L)]).
group1([H|T],Group,L,N,Grouped):-
        arg(N,H,Group),
        !,
        group1(T,Group,[H|L],N,Grouped).
group1([H|T],Old,L,N,[group(Old,L)|Grouped]):-
        arg(N,H,Group),
        group1(T,Group,[H],N,Grouped).
```

5.14 Lookup

Description The lookup concept is used to convert data stored in the local database into a list of terms that can be manipulated in the program. The most natural template (first argument of *findall*) is to use the same term as for the facts in the database.

Parameters

Res a free variable, will be bound to a list

Schema

```
:-mode lookup(-).
lookup(Res):-
        findall(q(X),q(X),Res).
```

Comments *findall* examplifies a typical *meta-predicate*, the second argument is actually a goal that will be executed. There are a number of other predicates of this type, and this feature can be extremely useful in writing interpreters, emulators etc, which treat data as program parts.
The *findall* predicate is significantly faster than *bagof* or *setof*. Their use is not recommended.

Example

```
% find all hops routing information
:-mode gather_hops(-).
gather_hops(Hops):-
        findall(hop(A,B,C),hop(A,B,C),Hops).
```

5.15 Fill matrix

Description This concept takes a list of entries with indices I and J and a value V, and put the value in a matrix M at position M_i, j.

Parameters

L a list of entries

Matrix a matrix

Schema

```
:-mode fill_matrix(+,+).
fill_matrix(L,Matrix):-
        (foreach(entry with [i:I,j:J,value:V],L),
         param(Matrix) do
            subscript(Matrix,[I,J],V)
        ).
```

Comments The program may fail if two entries in the list refer to the same index pair I and J, as the program would then try to insert two values at the same position.
It is not required that the input list contains all index combinations, we can use the *iteration on array* concept to fill any un-set elements with a default value.

Example The example fills an array $PMatrix$ with values from a list of *hop/3* terms. We also convert the node names in the *hop* term into node indices for lookup in the $PMatrix$ matrix.

```
:-mode fill_with_hops(+,+,+).
fill_with_hops([],_,_).
fill_with_hops([hop(Source,Dest,List)|R],Nodes,PMatrix):-
        find_node_index(Source,Nodes,S),
        find_node_index(Dest,Nodes,D),
        find_node_indices(List,Nodes,L),
        length(L,N),
        subscript(PMatrix,[S,D],pi_entry with [list:L,
                                               length:N]),
        fill_with_hops(R,Nodes,PMatrix).
```

5.16 Cartesian

Description This concept takes two input lists L and K and creates a list of pairs *Res*, in which each combination of elements of the first and the second list occurs exactly once.
The result is a list of terms *pair(X, Y)*, where X is an element of list L and Y is an element of list K.
The implementation uses nested *foreach do* loops to create each combination of elements once. The *fromto* accumulators are used to collect the result list.

Parameters

L a list

K a list

Res a free variable, will be bound to a list

Schema

```
:-mode cartesian(+,+,-).
cartesian(L,K,Res):-
        (foreach(X,L),
         fromto([],In,Out,Res),
         param(K) do
            (foreach(Y,K),
             fromto(In,In1,[pair(X,Y)|In1],Out),
             param(X) do
                true
            )
        ).
```

Comments Note the use of an empty body (*true*) in the innermost loop. All calculations are done by parameter passing alone.
If we want to create the elements in the same order as the elements in the input list, we have to exchange input and output arguments of the *fromto* statements, like so:

```
:-mode cartesian(+,+,-).
cartesian(L,K,Res):-
        (foreach(X,L),
         fromto(Res,In,Out,[]),
         param(K) do
            (foreach(Y,K),
             fromto(In,[pair(X,Y)|In1],In1,Out),
             param(X) do
                true
            )
        ).
```

Example The example builds all pairs of sources and sink nodes for flows and creates contribution structures from them. An additional accumulator *NoPath* is used to collect cases where there is no route between the nodes.

```
:-mode create_contribution(+,+,+,-,-).
create_contribution(FromList,ToList,
                    PMatrix,Contribution,NoPath):-
        (foreach(From,FromList),
         fromto([],In1,Out1,Contribution),
         fromto([],NP1,NP2,NoPath),
         param(ToList,PMatrix) do
            (foreach(To,ToList),
             fromto(In1,In,Out,Out1),
             fromto(NP1,NoPath1,NoPath2,NP2),
             param(From,PMatrix) do
                contribution(From,To,From,To,1,PMatrix,
                             In,Out,NoPath1,NoPath2)
            )
        ).
```

5.17 Ordered pairs

Description This concept creates ordered pairs of entries from a list. Each combination where the first element occurs in the input list before the second element is created exactly once.
The result is a list of terms *pair(X, Y)* where X and Y are elements of the input list L.

Parameters

L a list

K a free variable, will be bound to a list

Schema

```
:-mode ordered_pairs(+,-).
ordered_pairs(L,K):-
        ordered_pairs_lp(L,[],K).

ordered_pairs_lp([],L,L).
ordered_pairs_lp([H|T],In,Out):-
        ordered_pairs_lp2(H,T,In,In1),
        ordered_pairs_lp(T,In1,Out).

ordered_pairs_lp2(H,[],L,L).
ordered_pairs_lp2(H,[A|A1],In,Out):-
        ordered_pairs_lp2(H,A1,[pair(H,A)|In],Out).
```

Comments The second and third argument of *ordered_pairs_lp* and the third and fourth argument of *ordered_pairs_lp2* serve as an accumulator to collect the results.
This concept can also be implemented with nested *do* loops. The recursive version seems more natural.

```
ordered_pairs(L,K):-
        (fromto(L,[El|Rest],Rest,[_]),
         fromto(K,TPairs,RPairs,[]) do
            (foreach(R,Rest),
             param(El),
             fromto(TPairs,[pair(El,R)|Pairs],Pairs,RPairs) do
                true
            )
        ).
```

Chapter 6

Input/Output

In this chapter we will discuss input and output with the ECLiPSe system. We will first discuss how to read data into an ECLiPSe program, then discuss different output methods. From this we extract some rules about good and bad data formats that may be useful when defining a data exchange format between different applications. At the end we show how to use a simple report generator library in RiskWise, which converts data lists into HTML reports.

6.1 Reading input data into data structures

The easiest way to read data into ECLiPSe programs is to use the Prolog term format for the data. Each term is terminated by a fullstop, which is a dot (.) followed by some white space. The following code reads terms from a file until the end of file is encountered and returns the terms in a list.

```
:-mode read_data(++,-).
read_data(File,Result):-
        open(File,read,S),
        read(S,X),
        read_data_lp(S,X,Result),
        close(S).

read_data_lp(_S,end_of_file,[]):-
        !.
read_data_lp(S,X,[X|R]):-
        read(S,Y),
        read_data_lp(S,Y,R).
```

This method is very easy to use if both source and sink of the data are ECLiPSe programs. Unfortunately, data provided by other applications will normally not be in the Prolog term format. For them we will have to use some other techniques[1].

6.2 How to use DCGs

In this section we describe the use of tokenizers and DCG (definite clause grammar) to produce a very flexible input system. The input routine of the NDI mapper of RiskWise is completely implemented with this method, and we use some of the code in the examples below.

In this approach the data input is split into two parts, a tokenizer and a parser. The tokenizer read the input and splits it into tokens. Each token corresponds to one field in a data item. The parser is used to recognize the structure of the data and to group all data belonging to one item together.

Using these techniques to read data files is a bit of overkill, they are much more powerful and are used for example to read ECLiPSe terms themselves. But, given the right grammar, this process is very fast and extremely easy to modify and extend.

The top level routine *read_file* opens the file, calls the tokenizer, closes the file and starts the parser. We assume here that at the end of the parsing the complete input stream has been read (third argument in predicate *phrase*). Normally, we would check the unread part and produce an error message.

```
:-mode read_file(++,-).
read_file(File,Term):-
        open(File,'read',S),
        tokenizer(S,1,L),
        close(S),
        phrase(file(Term),L,[]).
```

The tokenizer is a bit complicated, since our NDI data format explicitly mentions end-of-line markers, and does not distinguish between lower and upper case spelling. Otherwise, we might be able to use the built-in tokenizer of ECLiPSe (predicate *read_token*).

The tokenizer reads one line of the input at a time and returns it as a string. After each line, we insert a *end_of_line(N)* token into the output with N the current line number. This can be used for meaningful error messages, if the

[1]We should at this point again mention the possibilities of the EXDR format which can be easily read into ECLiPSe, and which is usually simpler to generate in other languages than the canonical Prolog format.

parsing fails (not shown here). We then split the input line into white space separated strings, eliminate any empty strings and return the rest as our tokens.
The output of the tokenizer will be a list of strings intermixed with end-of-line markers.

```
:-mode tokenizer(++,++,-).
tokenizer(S,N,L):-
        read_string(S,'end_of_line',_,Line),
        !,
        open(string(Line),read,StringStream),
        tokenizer_string(S,N,StringStream,L).
tokenizer(_S,_N,[]).

tokenizer_string(S,N,StringStream,[H|T]):-
        non_empty_string(StringStream,H),
        !,
        tokenizer_string(S,N,StringStream,T).
tokenizer_string(S,N,StringStream,[end_of_line(N)|L]):-
        close(StringStream),
        N1 is N+1,
        tokenizer(S,N1,L).

non_empty_string(Stream,Token):-
        read_string(Stream, " \t\r\n", _, Token1),
        (Token1 = "" ->
            non_empty_string(Stream,Token)
        ;
            Token = Token1
        ).
```

We now show an example of grammar rules which define one data file of the NDI mapper, the RouterTrafficSample file. The grammar for the file format is shown below:

```
file                := <file_type_line>
                       <header_break>
                       [<data_line>]*
file_type_line      := RouterTrafficSample <newline>
header_break        := # <newline>
data_line           := <timestamp>
                       <router_name>
```

```
                     <tcp_segments_in>
                     <tcp_segments_out>
                     <udp_datagrams_in>
                     <udp_datagrams_in>
                     <newline>
timestamp         := <timestamp_ms>
router_name       := <name_string>
tcp_segments_in   := integer
tcp_segments_out  := integer
udp_datagrams_in  := integer
udp_datagrams_out := integer
```

This grammar translates directly into a DCG representation. The start symbol of the grammar is *file(X)*, the argument X will be bound to the parse tree for the grammar. Each rule uses the symbol `-->` to define a rule head on the left side and its body on the right. All rules for this particular data file replace one non-terminal symbol with one or more non-terminal symbols. The argument in the rules is used to put the parse tree together. For this data file, the parse tree will be a term *router_traffic_sample(L)* with L a list of terms *router_traffic_sample(A,B,C,D,E,F)* where the arguments are simple types (atoms and integers).

```
file(X) --> router_traffic_sample(X).

router_traffic_sample(router_traffic_sample(L)) -->
        file_type_line("RouterTrafficSample"),
        header_break,
        router_traffic_sample_data_lines(L).

router_traffic_sample_data_lines([H|T]) -->
        router_traffic_sample_data_line(H),
        !,
        router_traffic_sample_data_lines(T).
router_traffic_sample_data_lines([]) --> [].

router_traffic_sample_data_line(
            router_traffic_sample(A,B,C,D,E,F)) -->
        time_stamp(A),
        router_name(B),
        tcp_segments_in(C),
        tcp_segments_out(D),
        udp_datagrams_in(E),
        udp_datagrams_out(F),
        new_line.

tcp_segments_in(X) --> integer(X).

tcp_segments_out(X) --> integer(X).

udp_datagrams_in(X) --> integer(X).

udp_datagrams_out(X) --> integer(X).
```

Note the cut in the definition of *router_traffic_sample_data_lines*, which is used to commit to the rule when a complete data line as been read. If a format error occurs in the file, then we will stop reading at this point, and the remaining part of the input will be returned in *phrase*.
The following rules define the basic symbols of the grammar. Terminal symbols are placed in square brackets, while additional Prolog code is added with braces. The *time_stamp* rule for example reads one token X. It first checks that X is a string, then converts it to a number N, and then uses a library predicate *eclipse_date* to convert N into a date representation $Date$: 2003/04/2917 : 49 : 48, which is returned as the parse result.

```
file_type_line(X) --> ndi_string(X), new_line.

header_break -->
        ["#"],
        new_line.

router_name(X) --> name_string(X).

time_stamp(Date) -->
        [X],
        {string(X),
         number_string(N,X),
         eclipse_date(N,Date)
        }.

integer(N) --> [X],{string(X),number_string(N,X),integer(N)}.

name_string(A) --> ndi_string(X),{atom_string(A,X)}.
```

```
ndi_string(X) --> [X],{string(X)}.

new_line --> [end_of_line(_)].
```

These primitives are reused for all files, so that adding the code to read a new file format basically just requires some rules defining the format.

6.3 Creating output data files

In this section we discuss how to generate output data files. We present three methods which implement different output formats.

6.3.1 Creating Prolog data

We first look at a special case where we want to create a file which can be read back with the input routine shown in section 6.1. The predicate *output_data* writes each item in a list of terms on one line of the output file, each line terminated by a dot (.). The predicate *writeq* ensures that atoms are quoted, operator definitions are handled correctly, etc.

```
:-mode output_data(++,+).
output_data(File,L):-
        open(File,'write',S),
        (foreach(X,L),
         param(S) do
           writeq(S,X),writeln('.')
        ),
        close(S).
```

It is not possible to write unbound constrained variables to a file and to load them later, they will not be re-created with their previous state and constraints. In general, it is a good idea to restrict the data format to ground terms, i.e. terms that do not contain any variables.

6.3.2 Simple tabular format

Generating data in Prolog format is easy if the receiver of the data is another ECLiPSe program, but may be inconvienient if the receiver is written in another language. In that case a tabular format that can be read with routines like *scanf* is easier to handle. The following program segment shows how this is done. For each item in a list we extract the relevant arguments, and write them to the output file separated by white space.

```
:-mode output_data(++,+).
output_data(File,L):-
        open(File,'write',S),
        (foreach(X,L),
         param(S) do
            output_item(S,X)
        ),
        close(S).

output_item(S,data_item with [attr1:A1,attr2:A2]):-
        write(S,A1),
        write(S,' '),
        write(S,A2),
        nl(S).
```

We use the predicate *write* to print out the individual fields. As this predicate handles arbitrary argument types, this is quite simple, but it does not give us much control over the format of the fields.

6.3.3 Using *printf*

Instead, we can use the predicate *printf* which behaves much like the C-library routine. For each argument we must specify the argument type and an optional format. If we make a mistake in the format specification, a run-time error will result.

```
:-mode output_data(++,+).
output_data(File,L):-
        open(File,'write',S),
        (foreach(X,L),
         param(S) do
            output_item(S,X)
        ),
        close(S).

output_item(S,data_item with [attr1:A1,attr2:A2]):-
        printf(S,"%s %d\n",[A1,A2]).
```

We can use the same schema for creating tab or comma separated files, which provides a simple interface to spreadsheets and data base readers.

6.4 Good and bad data formats

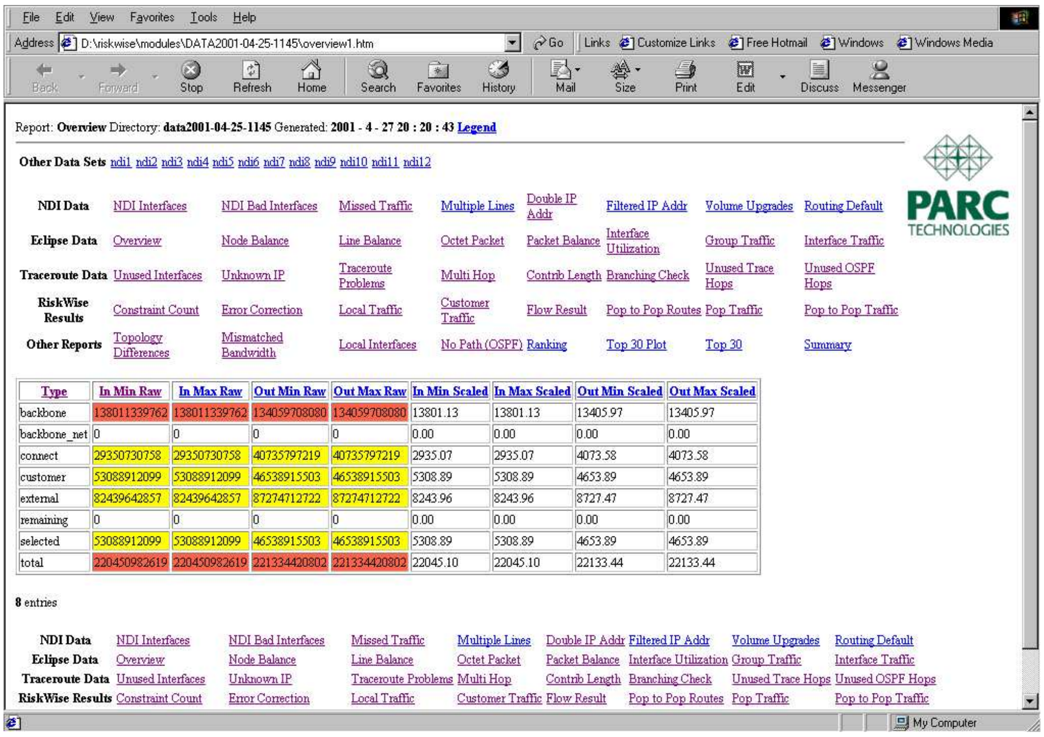
When defining the data format for an input or output file, we should choose a representation which suits the ECLiPSe application. Table 6.1 shows good and bad formats. Prolog terms are very easy to read and to write, simple tabular forms are easy to write, but more complex to read. Comma separated files need a special tokenizer which separates fields by comma characters. The most complex input format is given by a fixed column format, for example generated by FORTRAN applications. We should avoid such data formats as input if possible, since they require significant development effort.

Format	Input	Output
Prolog terms	++	++
EXDR	++	++
White space separated	+	++
Comma separated	-	++
Fixed columns	- -	+

Table 6.1: Good and bad input formats

6.5 Report generation

There is another output format that can be generated quite easily. RiskWise uses a *report* library, which presents lists of items as HTML tables in hyper linked files. This format is very useful to print some data in a human readable form, as it allows some navigation across files and sorting of tables by different columns. Figure 6.1 shows an example from the RiskWise application.



Address D:\riskwise\modules\DATA2001-04-25-1145\overview1.htm

Report: **Overview** Directory: **data2001-04-25-1145** Generated: **2001 - 4 - 27 20 : 20 : 43** **Legend**

Other Data Sets ndi1 ndi2 ndi3 ndi4 ndi5 ndi6 ndi7 ndi8 ndi9 ndi10 ndi11 ndi12

NDI Data	NDI Interfaces	NDI Bad Interfaces	Missed Traffic	Multiple Lines	Double IP Addr	Filtered IP Addr	Volume Upgrades	Routing Default
Eclipse Data	Overview	Node Balance	Line Balance	Octet Packet	Packet Balance	Interface Utilization	Group Traffic	Interface Traffic
Traceroute Data	Unused Interfaces	Unknown IP	Traceroute Problems	Multi Hop	Contrib Length	Branching Check	Unused Trace Hops	Unused OSPF Hops
RiskWise Results	Constraint Count	Error Correction	Local Traffic	Customer Traffic	Flow Result	Pop to Pop Routes	Pop Traffic	Pop to Pop Traffic
Other Reports	Topology Differences	Mismatched Bandwidth	Local Interfaces	No Path (OSPF)	Ranking	Top 30 Plot	Top 30	Summary

Type	**In Min Raw**	**In Max Raw**	**Out Min Raw**	**Out Max Raw**	**In Min Scaled**	**In Max Scaled**	**Out Min Scaled**	**Out Max Scaled**
backbone	138011339762	138011339762	134059708080	134059708080	13801.13	13801.13	13405.97	13405.97
backbone_net	0	0	0	0	0.00	0.00	0.00	0.00
connect	29350730758	29350730758	40735797219	40735797219	2935.07	2935.07	4073.58	4073.58
customer	53088912099	53088912099	46538915503	46538915503	5308.89	5308.89	4653.89	4653.89
external	82439642857	82439642857	87274712722	87274712722	8243.96	8243.96	8727.47	8727.47
remaining	0	0	0	0	0.00	0.00	0.00	0.00
selected	53088912099	53088912099	46538915503	46538915503	5308.89	5308.89	4653.89	4653.89
total	220450982619	220450982619	221334420802	221334420802	22045.10	22045.10	22133.44	22133.44

8 entries

NDI Data	NDI Interfaces	NDI Bad Interfaces	Missed Traffic	Multiple Lines	Double IP Addr	Filtered IP Addr	Volume Upgrades	Routing Default
Eclipse Data	Overview	Node Balance	Line Balance	Octet Packet	Packet Balance	Interface Utilization	Group Traffic	Interface Traffic
Traceroute Data	Unused Interfaces	Unknown IP	Traceroute Problems	Multi Hop	Contrib Length	Branching Check	Unused Trace Hops	Unused OSPF Hops
RiskWise Results	Constraint Count	Error Correction	Local Traffic	Customer Traffic	Flow Result	Pop to Pop Routes	Pop Traffic	Pop to Pop Traffic

Figure 6.1: HTML Report

Chapter 7

If it doesn't Work

Suppose we have followed our methodology, and just made some modification of our previously running program. Unfortunately, after the change the program stops working. What do we do next to find the problem and to correct it? We first have to understand what types of errors can occur in an ECLiPSe program.

7.1 Understanding failure

We can distinguish five types of failure, each with its own set of causes and possible remedies.

7.1.1 Run-time errors

Run-time errors occur if we call a built-in predicate with a wrong argument pattern. This will usually either be a type mismatch, i.e. using a number where an atom is expected, or an instantiation problem, i.e. passing a variable where a ground value was expected or vice versa. In this case the ECLiPSe system prints out the offending call together with an error message indicating the problem. If we are lucky, we can identify the problem immediately, otherwise we may have to look up the documentation of the built-in to understand the problem.
In the following example *bug0*, we have called the predicate *open/3* with an incorrect first argument.

```
:-export(bug0/0).
bug0:-
        open(1,read,S), % wrong
        read(S,X),
        writeln(X),
        close(S).
```

When we run the query *bug0*, the ECLiPSe system prints out the message:

```
type error in open(1, read, S)
```

In general run-time errors are quite simple to locate and to fix. But the system does not indicate which particular call of a built-in is to blame. We may need to use the tracer to find out which of dozens of calls to the predicate *is/2* for example may have caused a particular problem. There are several things that may help to avoid the tedious tracing.

- If the call to the predicate contains some variable name, we may be able to locate the problem by searching for that name in the source file(s).
- Well placed logging messages may also be helpful, they indicate which program part is responsible.
- If we have only applied a small change to a previously working program, then it is likely that the error is located close to the change we have made. Testing the program after each change will speed up development.

7.1.2 Environment limits

These errors occur when we hit some limit of the run-time system, typically exceeding available memory. An example is the run-away recursion in the program *bug1* below:

```
:-export(bug1/0).
bug1:-
        lp(X),  % wrong
        writeln(X).

lp([_H|T]):-
        lp(T).
lp([]).
```

After some seconds, the system returns an error message of the form:

```
*** Overflow of the local/control stack!
You can use the "-l kBytes" (LOCALSIZE) option to have a larger stack.
Peak sizes were: local stack 13184 kbytes, control stack 117888 kbytes
```

Typical error sources are passing free variables to a recursion, where the terminating clause is not executed first or the use of an infinite data structure.

7.1.3 Failure

Probably the most common error symptom is a failure of the application. Instead of producing an answer, the system returns 'no'. This is caused by:

- Calling a user-defined predicate with the wrong calling pattern. If none of the rules of a predicate matches the calling pattern, then the predicate will fail. Of course, quite often this is intentional (tests for some condition for example). It becomes a problem if the calling predicate does not handle the failure properly. We should know for each predicate that we define if it can fail and make sure that any calling predicate handles that situation.
- Calling a built-in predicate with arguments so that it fails unexpectedly. This is much less likely, but some built-in predicates can fail if the wrong arguments are passed.
- Wrong control structure. A common problem is to miss the alternative branch in an *if-then-else construct*. If the condition part fails, then the whole call fails. We must always add an *else* part, perhaps with an empty statement *true*.

The best way of finding failures is by code inspection, helped by logging messages which indicate the general area of the failure. If this turns out to be too complex, we may have to use the tracer.

7.1.4 Wrong answer

More rare is the situation where a "wrong" answer is returned. This means that the program computes a result, which we do not accept as an answer. The cause of the problem typically lies in a mismatch of the intended behaviour of the program (what we think it is doing) and the actual behaviour. In a constraint problem we then have to identify which constraint(s) should eliminate this answer and why that didn't happen. There are two typical scenarios.

- The more simple one is caused by missing a constraint alltogether, or misinterpreting the meaning of a constraint that we did include.
- The more complex scenario is a situation where the constraint did not trigger and therefore was not activated somewhere in the program execution.

We can often distinguish these problems by re-stating the constraint a second time after the wrong answer has been found.
If the constraint still accepts the solution, then we can assume that it simply does not exclude this solution. We will have to reconsider the declarative definition of the constraint to reject this wrong answer.
If the program fails when the constraint is re-stated, then we have a problem with the dynamic execution of the constraints in the constraint solver. That normally is a much more difficult problem to solve, and may involve the constraint engine itself.

7.1.5 Missing answer

Probably the most complex problem is a missing answer, i.e. the program produces correct answers, but not all of them. This assumes that we know this missing answer, or we know how many answers there should be to a particular problem and we get a different count when we try to generate them all. In the first case we can try to instantiate our problem variables with the missing answer before stating the constraints and then check where this solution is rejected.
This type of problem often occurs when we develop our own constraints and have made a mistake in one of the propagation rules. It should not occur if we use a pre-defined constraint solver.

7.2 Checking program points

A very simple debugging technique is the logging of certain predicate calls as they occur in the program.

```
:-export(bug2/0).

bug2:-
        ...
        generate_term(Term),
        analyze_term(Term,Result),
        ...
```

Suppose we assume that the predicate *analyze_term* is responsible for a problem. We can then simply add a *writeln* call before and/or after the predicate call to print out the suspected predicate. The output before that call should show all input arguments, the output after the call should also show the output results.

```
:-export(bug2/0).

bug2:-
        ...
        generate_term(Term),
        writeln(analyze_term(Term,Result)),
        analyze_term(Term,Result),
        writeln(analyze_term(Term,Result)),
        ...
```

Used sparingly for suspected problems, this technique can avoid the use of the debugger and at the same time give a clear indication of the problem. The ECLiPSe system will normally restrict the output of a term to a certain complexity, so that long lists are not printed completely. This can be influenced by the *print_depth* flag of *set_flag*.
Such messages can be written to the *log_output* stream (or a user-defined stream) which can be later be discarded by

```
set_stream(log_output,null)
```

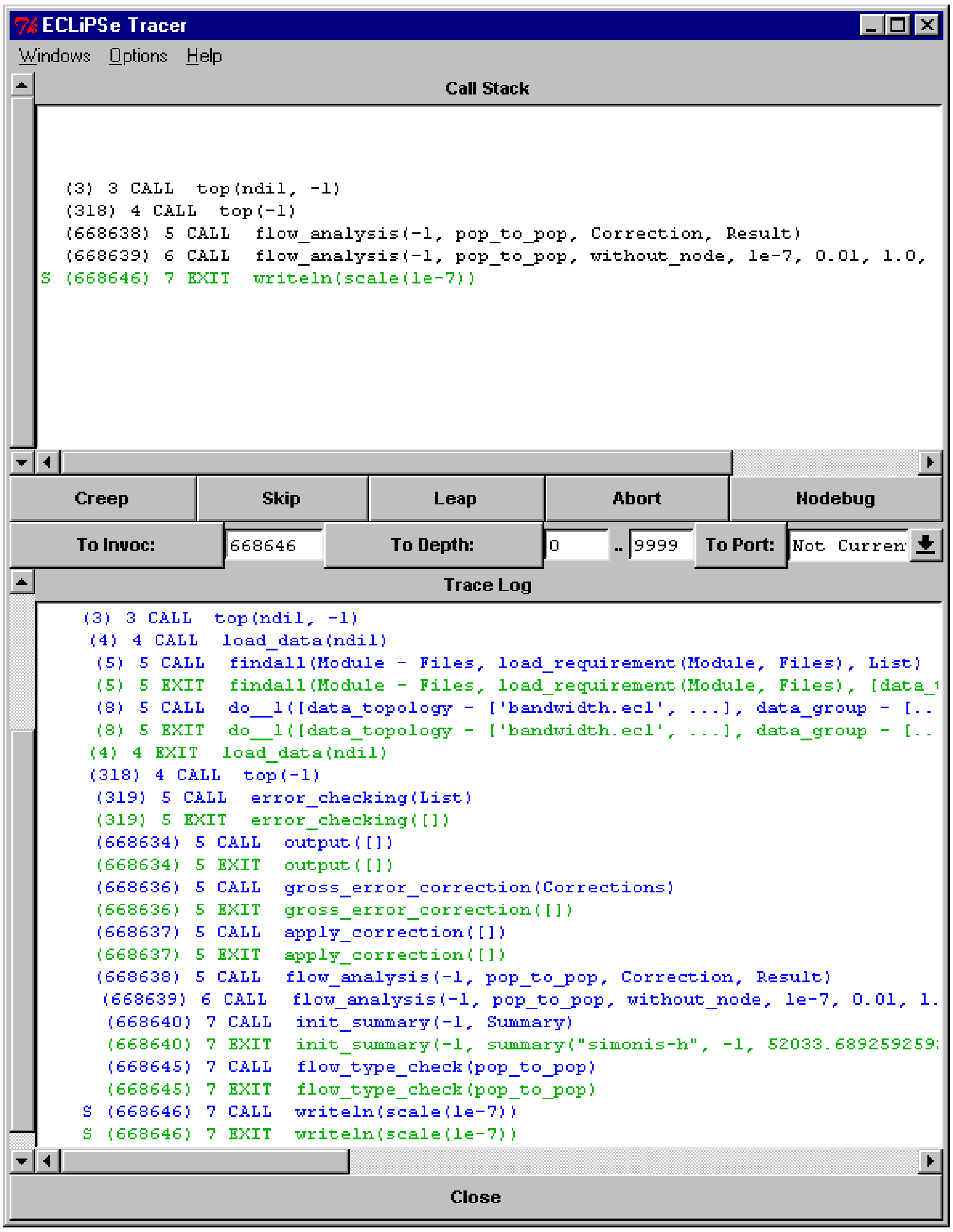
or which can be directed to a log file. This can be useful in "live" code, in order to capture information about problems in failure cases.

7.3 Debugger

Ideally, we can figure out all problems without running the application in the debugger. But at some point we may need to understand what the system is doing in more detail. We can then start the ECLiPSe tracer tool from the *Tools* menu. Figure 7.1 shows a typical output. In the top window we see the current stack of predicate calls, in the bottom we see a history of the program execution.

7.3.1 Tracing

The three buttons *Creep*, *Skip*, *Leap* show the main commands of the tracer. The creep operation single-steps to the next statement in the program. The



```
  (3) 3 CALL  top(ndil, -1)
  (318) 4 CALL  top(-1)
  (668638) 5 CALL  flow_analysis(-1, pop_to_pop, Correction, Result)
  (668639) 6 CALL  flow_analysis(-1, pop_to_pop, without_node, 1e-7, 0.01, 1.0,
S (668646) 7 EXIT  writeln(scale(1e-7))
```

```
    (3) 3 CALL  top(ndil, -1)
     (4) 4 CALL  load_data(ndil)
      (5) 5 CALL  findall(Module - Files, load_requirement(Module, Files), List)
      (5) 5 EXIT  findall(Module - Files, load_requirement(Module, Files), [data_
      (8) 5 CALL  do__1([data_topology - ['bandwidth.ecl', ...], data_group - [..
      (8) 5 EXIT  do__1([data_topology - ['bandwidth.ecl', ...], data_group - [..
     (4) 4 EXIT  load_data(ndil)
     (318) 4 CALL  top(-1)
      (319) 5 CALL  error_checking(List)
      (319) 5 EXIT  error_checking([])
      (668634) 5 CALL  output([])
      (668634) 5 EXIT  output([])
      (668636) 5 CALL  gross_error_correction(Corrections)
      (668636) 5 EXIT  gross_error_correction([])
      (668637) 5 CALL  apply_correction([])
      (668637) 5 EXIT  apply_correction([])
      (668638) 5 CALL  flow_analysis(-1, pop_to_pop, Correction, Result)
       (668639) 6 CALL  flow_analysis(-1, pop_to_pop, without_node, 1e-7, 0.01, 1.
        (668640) 7 CALL  init_summary(-1, Summary)
        (668640) 7 EXIT  init_summary(-1, summary("simonis-h", -1, 52033.689259259
        (668645) 7 CALL  flow_type_check(pop_to_pop)
        (668645) 7 EXIT  flow_type_check(pop_to_pop)
    S (668646) 7 CALL  writeln(scale(1e-7))
    S (668646) 7 EXIT  writeln(scale(1e-7))
```

Figure 7.1: ECLiPSe Tracer

skip operation executes the current query and stops again when that query either succeeds or fails. The leap operation continues execution until it reaches a spy point, a predicate call for which we have indicated some interest before. A normal debugging session consists of a sequence of leap, skip and creep operations to reach an interesting part of the program and then a more detailed step-by step analysis of what is happening. It is pointless and very time consuming to single-step through a part of the program that we have already checked, but we have to be careful not to skip over the interesting part of the program.

7.3.2 Jumping to a program point

In a large program, it may be difficult to leap directly to the interesting part of the program. But we may have to repeat this operation several times, if we repeatedly leap/skip over an interesting statement. We can use the invocation number of a statement to jump to this exact place in the execution again. The invocation number is printed in parentheses at the beginning of each trace line. By re-starting the debugger, copying this number into the text field to the right of the button *To Invoc:* and then pressing this button we can directly jump to this location.
Unfortunately, jumping to an invocation number can be quite slow in a large program. The debugger has to scan each statement to check its invocation number. It can be faster (but more tedious) to use skip and leap commands to reach a program point[1].

[1]Experiment! Your mileage may vary!

Chapter 8

Correctness and Performance

The effort in design and implementation is aimed at producing a working and maintainable program with minimal effort. But how do we check that a program is actually working?
We typically define what a correct program should do only in an informal way, stating constraints that should be satisfied, data formats that should be accepted, etc. Proving correctness would require a much more formal approach, where we first agree on a specification in a formal language and then prove that the implementation satisfies that specification. But even that does not protect us from a misunderstanding in defining the spec itself, which may not reflect the wishes of the user.
Lacking this formal approach, the best hope of checking correctness of a program lies in exercising it with different types of tests.
Related to the question of correctness is the issue of performance. The program should not only produce correct results, but must produce them within a limited time. The correct result which is available after five years of computation is (most of the time) as useless as the wrong result obtained in five seconds.

8.1 Testing

Testing is often one of the activities that gets dropped when delivery time points get closer. This is a sad mistake, as any problem that is not recognized immediately will take much more effort to find and correct later on. In an ideal world, tests for each component are defined before implementation and cover each level of the design. In the real world we must find the right compromise between spending time on defining tests in the beginning and time spent on developing the application core.

A test may have different objectives:

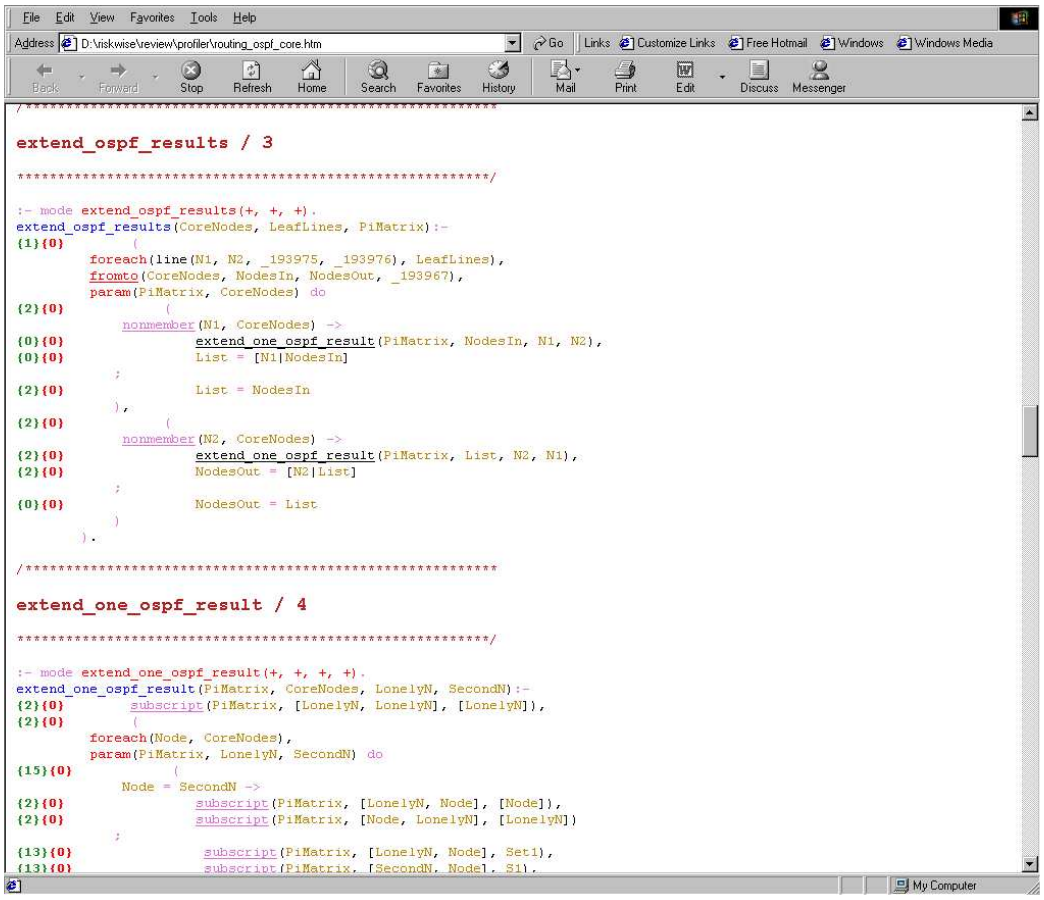
- The weakest form is a test that is run to check if a program works with the test data, i.e. produces some answer without generating an error. We have suggested this form of test above to check an (incomplete) early implementation.
- A more complex test can be used to exercise all (or nearly all) of the program code. The test data must contain enough variation to force all alternative parts in the program. This can be achieved either by accumulating large number of tests or by designing particular tests to handle special cases.
- Another test form is *regression* testing. This form of testing is used to evaluate results of a program run after each modification of the code. Results of previous runs are used to check the current results. This does not check for correctness, but rather for the same answer before and after a change.
- *Correctness* of a program can only be checked if we have obtained the expected results of a test in an independent way, either by hand or by a trusted program, or simply by re-using benchmark sets from a third party.
- We may also be using some tests to do *performance testing*, i.e. to check that the program finds the solution within given limits on execution time and/or system resources. Clearly, this only makes sense if we also check the result for correctness.

It is important to realize the limitations of the tests that we perform. If we have never checked that the solutions produced by a regression test are correct, then they will be most likely *not* correct. We only know that they are still the same solutions that the program has found before.

Unfortunately, testing of combinatorial problem solvers is even more complex than testing of “normal” programs. Often, a given problem has more than one solution, and it is not clear in which order the answers will be produced. Providing one solution may not be enough, but there may be millions of solutions in total.

8.2 Profiling

The line profiling[1] tool that we already mentioned above can be used to check the coverage of the program with some query. We can easily find lines that are not executed at all, indicating the need for another test case. If we cannot construct a test which executes a program segment, then this indicates some problem.
We can use the same profile to find program parts which are executed very often and this can provide hints for optimization. Normally it is better not just to concentrate on those parts that are called very often, but on those which are calling these predicates.


File Edit View Favorites Tools Help
Address D:\riskwise\review\profiler\routing_ospf_core.htm
Go
Links Customize Links Free Hotmail Windows Windows Media
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss Messenger
My Computer

```
/***********************************************************

extend_ospf_results / 3

***********************************************************/

:- mode extend_ospf_results(+, +, +).
extend_ospf_results(CoreNodes, LeafLines, PiMatrix):-
{1}{0}          (
         foreach(line(N1, N2, _193975, _193976), LeafLines),
         fromto(CoreNodes, NodesIn, NodesOut, _193967),
         param(PiMatrix, CoreNodes) do
{2}{0}              (
             nonmember(N1, CoreNodes) ->
{0}{0}                      extend_one_ospf_result(PiMatrix, NodesIn, N1, N2),
{0}{0}                      List = [N1|NodesIn]
             ;
{2}{0}                      List = NodesIn
             ),
{2}{0}              (
             nonmember(N2, CoreNodes) ->
{2}{0}                      extend_one_ospf_result(PiMatrix, List, N2, N1),
{2}{0}                      NodesOut = [N2|List]
             ;
{0}{0}                      NodesOut = List
             )
         ).

/***********************************************************

extend_one_ospf_result / 4

***********************************************************/

:- mode extend_one_ospf_result(+, +, +, +).
extend_one_ospf_result(PiMatrix, CoreNodes, LonelyN, SecondN):-
{2}{0}          subscript(PiMatrix, [LonelyN, LonelyN], [LonelyN]),
{2}{0}          (
         foreach(Node, CoreNodes),
         param(PiMatrix, LonelyN, SecondN) do
{15}{0}              (
             Node = SecondN ->
{2}{0}                      subscript(PiMatrix, [LonelyN, Node], [Node]),
{2}{0}                      subscript(PiMatrix, [Node, LonelyN], [LonelyN])
             ;
{13}{0}                      subscript(PiMatrix, [LonelyN, Node], Set1),
{13}{0}                      subscript(PiMatrix, [SecondN, Node], S1),
```

Figure 8.1: Profiling Example

Figure 8.1 shows the output of the profiling tool. Each line is marked with the

[1]The profiling facility is now available as one of the ECLiPSe libraries in the library coverage. The actual output looks slightly different.

number of times it is executed (the first number in green) and the number of times we backtrack over this line (the second number in red). In this example shown there are two parts of if-then-else predicates which have not been selected by the test example.

8.3 Reviewing

A standard technique to check for consistency in the development is a reviewing process. Each module of an application goes through a review process, where persons not connected with the development check the deliverables (source code and documentation) for completeness and consistency. This review process serves multiple purposes:

- It forces the developer to finish a version of the program with a certain polish.
- It helps to find inconsistencies or missing explanations in the documentation.
- It encourages "best practice" in the ECLiPSe application development, bringing together experts from different application teams.
- It helps spread knowledge about applications and their sub-systems, so that re-use opportunities are recognized earlier.

On the other hand, a number of problems are normally not recognized by a review:

- The review checks one version of an application at a given time point. It does not guarantee that changes and modifications after the review are performed to the same standard.
- A successful code review does not imply that the application code is correct. Reviewers might sometimes note suspect code, but a review cannot replace proper testing.
- If nobody actually checks the code, then the whole process becomes useless overhead. This means that resources must be properly allocated to the review, it is not a task that reviewers can undertake in their spare time.
- Comments and change requests in the review must be recorded and acted on. A formal review comment form may be used, alternatively we might work with detailed and complete minutes.

8.4 Issues to check for

8.4.1 Unwanted choicepoints

It is important to remove all unwanted choicepoints in an application, since they are a common source of errors. In addition, a choicepoint requires a significant amount of memory, so that leaving unwanted choicepoints is also a performance problem.
For most predicates, in particular those following one of the programming concepts in chapter 5, it is quite simple to avoid unwanted choicepoints. Other predicates may need more effort. We can use the ECLiPSe debugger to detect if there are any unwanted choicepoints. In the trace output for the *EXIT* port ECLiPSe will print a * if the predicate leaves a choicepoint. We can easily check a complete query by skipping over its execution and checking the exit port. If a choicepoint is indicated, we can re-run the query to locate the missed choicepoint.

8.4.2 Open streams

A common error is to open some file without closing it before leaving. This typically happens if it is opened in one part of the program and closed in another part. Normally everything works fine, but under some exceptional circumstances the second part is not executed, leaving the file open. This can consume system resources quite quickly, leading to follow-on problems. It is a good idea to verify that every call to *open/3* is followed by a *close/1*, even if some other part of the program unexpectedly fails.

8.4.3 Modified global state

We have already stated that changing global state should be used as a last resort, not as a common practice. But if the program modifies dynamic predicates, creates global variables or uses *record*, then we must be very careful to restore the state properly, i.e. remove dynamic predicates after use, reset global variables etc.

8.4.4 Delayed goals

Normally, a solver should not leave delayed goals after it is finished. For some solvers, we can enforce this by instantiating all solver variables in a solution, others require more complex actions. If a query returns with delayed goals, this should be seen as an error message that needs to be investigated.

Appendix A

Style Guide

A.1 Style rules

1. There is one directory containing all code and its documentation (using sub-directories).
2. Filenames are of form [a-z][a-z_]+ with extension .ecl .
3. One file per module, one module per file.
4. Each module is documented with comment directives.
5. All required interfaces are defined in separate spec files which are included in the source with a *comment include* directive. This helps to separate specification and implementation code.
6. The actual data of the problem is loaded dynamically from the Java interface; for stand-alone testing data files from the data directory are included in the correct modules.
7. The file name is equal to the module name.
8. Predicate names are of form [a-z][a-z_]*[0-9]* . Underscores are used to separate words. Digits should only be used at the end of the name. Words should be English.
9. Variable names are of form [A-Z_][a-zA-Z]*[0-9]* . Separate words star with capital letters. Words should be English. The plural should be used for lists and other collections. Digits should only be used at the end to distinguish different versions of the same conceptual thing.

10. The code should not contain singleton variables, unless their names start with _. The final program may not generate singleton warnings.
11. Each exported predicate is documented with a comment directive.
12. Clauses for a predicate must be consecutive.
13. Base clauses should be stated before recursive cases.
14. Input arguments should be placed before output arguments.
15. Predicates which are not exported should be documented with a single line comment. It is possible to use comment directives instead.
16. The sequence of predicates in a file is top-down with a (possibly empty) utility section at the end.
17. All structures are defined in one file (e.g. flow_structures.ecl) and are documented with comment directives.
18. Terms should not be used; instead use named structures.
19. When possible, use do loops instead of recursion.
20. When possible, use separate predicates instead of disjunction or if-then-else.
21. There should be no nested if-then-else construct in the code.
22. All input data should be converted into structures at the beginning of the program; there should be no direct access to the data afterwards.
23. All numeric constants should be parametrized via facts. There should be no numeric values (other than 0 and 1) in rules.
24. The final code should not use failure-loops; they are acceptable for debugging or testing purposes.
25. Cuts (!) should be inserted only to eliminate clearly defined choice points.
26. The final code may not contain open choice points, except for alternative solutions that still can be explored. This is verified with the tracer tool in the debugger.
27. Customizable data facts should always be at the end of a file; their use is deprecated.

28. The predicate member/2 should only be used where backtracking is required; otherwise use memberchk/2 to avoid hidden choice points.
29. The final code may not contain dead code except in the file/module unsupported.ecl. This file should contain all program pieces which are kept for information/debugging, but which are not part of the deliverable.
30. The test set(s) should exercise 100 percent of the final code. Conformity is checked with the line coverage profiler.
31. Explicit unification (=/2) should be replaced with unification inside terms where possible.
32. There is a top-level file (document.ecl) which can be used to generated all on-line documentation automatically.
33. Don't use ','/2 to make tuples.
34. Don't use lists to make tuples.
35. Avoid append/3 where possible, use accumulators instead.

A.2 Module structure

The general form of a module is:

1. module definition
2. module comment or inclusion of a spec file
3. exported/reexported predicates
4. used modules
5. used libraries
6. local variable definitions
7. other global operations and settings
8. predicate definitions

A.3 Predicate definition

The general form of a predicate definition is:

1. predicate comment directive
2. mode declaration
3. predicate body

Appendix B

Layout Rules

In formulating these layout rules, we have taken certain arbitrary choices between different sensible formatting variants. While these choices work well for us, you may want to adapt a slightly different style. As long as it is done consistently (and agreed within a team), you should follow your own preferences.

B.1 General guidelines

Code should be indented consistently. Tabs should be every 8 spaces. Indentation should be one tab per level of indentation, except for highly indented code which may be indented at 4 spaces per level of indentation after the first.
Each line should not extend beyond 79 characters, unless this is necessary due to the use of long string literals. If a statement or predicate call is too long, continue it on the next line indented two levels deeper. If the statement or call extends over more than two lines, then make sure the subsequent lines are indented to the same depth as the second line. For example:

```
Here is A_really_long_statement_that_does_not_fit +
           On_one_line + In_fact_it_doesnt_even_fit +
           On_two_lines.
```

Don't put more than one statement or call on a line.
Put spaces after commas in comma-separated lists. Put spaces either side of infix operators. In both cases, this makes them easier to read, particularly in expressions containing long names or names with underscores. (There are some exceptions, such as module qualification `foo:bar` and functor/arity pairings `foo/3`. But not unification `X = Y` or list consing `[Head | Tail]`.)

Make judicious use of single blank lines in clause bodies to separate logical components.

B.2 Predicates and clauses

Keep all clauses of a predicate in the one place. Leave at least one blank line between the clauses of one predicate and the next (particularly if they have similar names), since otherwise they may at first glance appear to be all from the same predicate (of course, this never happens because there's a comment before every predicate, right?). Similarly, it is best not to leave blank lines between the clauses of a predicate (particularly if done inconsistently), since otherwise at first glance they may appear to be from different predicates.
Clause heads should be flush against the left margin. As well as making them easier to pick out visually, this makes it easier to grep for the definitions of predicates (as opposed to their invocations). The head/body separator ':-' should follow the head on the same line. The body of the clause should then commence on the next line, indented one tab stop.

```
non_overlap(Start1, Dur1, Start2, _Dur2):-
        Start1 + Dur1 #=< Start2.
non_overlap(Start1, _Dur1, Start2, Dur2):-
        Start1 #>= Start2 + Dur2.
```

B.3 If-then-elses

If-then-elses should always be parenthesised. Always include the else part, even if you don't think it's required. Always put semicolons at the start of a new line, aligned with the opening and closing parentheses. Example:

```
( test1 ->
        goal1
;
        goal2
),
```

B.4 Disjunctions

Disjunctions should be formatted in a similar way. Example:

```
(
        goal1
;
        goal2
).
```

B.5 Do loops

Do loops should also always be parenthesised. Loop conditions should be listed one per line, starting after the opening parenthesis and indented one character. The ‘do’ keyword should be at the end of the last condition. The body of the loop should then follow on the next line, again indented. Example:

```
(for(J, I + 1, N),
 param(A, I) do
        Diff is J - I,
        A[I] #\= A[J],
        A[I] #\= A[J] + Diff,
        A[I] #\= A[J] - Diff
).
```

B.6 Comments

Every predicate should have a one line comment documenting what it does, all module interfaces should be properly documented with a comment directive.
If an individual clause is long, it should be broken into sections, and each section should have a *block comment* describing what it does; blank lines should be used to show the separation into sections. Comments should precede the code to which they apply, rather than following it.

```
%
% This is a block comment; it applies to the code in the next
% section (up to the next blank line).
%
        blah,
        blah,
        blahblah,
        blah,
```

If a particular line or two needs explanation, a *line* comment

```
        % This is a "line" comment;
        % it applies to the next line or two
        % of code
        blahblah
```

or an *inline* comment

```
        blahblah        % This is an "inline" comment
```

should be used.

Appendix C

Core Predicates

This section lists essential ECLiPSe built-in and library predicates. These predicates are used in most applications, and should be well understood. There are more than 1000 other built-in and library predicates which are required for specific tasks only.

C.1 Modules

module directive define a module

export directive make predicates available outside a module

use_module directive make predicates from other module available

lib directive make predicates from library available

: call a predicate from a particular module

C.2 Predicate definition

mode directive define input/output modes for a predicate

comment directive define comments about a module or a predicate

C.3 Control

, conjunction, and

; disjunction, or

-> implication, if-then-else together with disjunction

! cut, remove choices

call/1 call argument as a predicate call

once/1 call argument as a predicate call, find one solution only

not/1 negation, fail if call to argument succeeds

true/0 always succeed, empty predicate definition

block/3 define a block in order to catch exceptions

exit_block/1 jump out of a block

C.3.1 Do Loops

do general iteration operator

foreach iterate over each element in a list

foreacharg iterate over each argument of a term

fromto accumulator argument

for iterate over all integers between two bounds

param declare parameter variables which are passed into the do-loop

C.4 I/O

exists/1 succeeds if a file exists

open/3 open a file or a string as a stream

close/1 close a stream

write/2 write some term to a stream

nl/1 write a new-line to a stream

writeln/2 write a term to a stream, followed by a new-line

writeq/2 write a term in canoncial form, so that if can be read back

read/2 read a term from a stream

read_string/4 read a string from a stream

concat_string/2 build a string from a list of components

phrase/3 call DCG analyzer

read_exdr/2 read a term from a file in EXDR format

write_exdr/2 write a term to a file in EXDR format

flush/1 flush an output stream

C.5 Arrays

dim/2 define an array; can also be used to find size of an array

subscript/3 get an element of an array

C.6 Hash Tables

hash_create/1 create a hash table

hash_add/3 add an item to a hash table

hash_find/3 find if an item is in a hash table

C.7 Arithmetic

is/2 evaluate term

=:= **/2** evaluate two terms and compare for equality

=\= **/2** evaluate two terms and compare for disequality

>**/2** evaluate two terms and compare for inequality

>= **/2** evaluate two terms and compare for inequality

<**/2** evaluate two terms and compare for inequality

=<**/2** evaluate two terms and compare for inequality

C.8 Terms and structures

struct/1 define named structure

with access element(s) in named structure

of find argument position in named structure

functor/3 define/analyze term for functor and arity

arg/3 access argument of term

compare/3 compare two terms for lexicographical order

= /2 two terms can be unified

== /2 two terms are identical

\= /2 two terms cannot be unified

number/1 argument is a number

integer/1 argument is an integer

atom/1 argument is an atom

var/1 argument is a variable

string/1 argument is a string

real/1 argument is a float

compound/1 argument is a term

C.9 List Handling

sort/2/4 sort a list

memberchk/2 check if an item occurs in a list

length/2 find the length of a list or create a list with given length

findall/3 find all solutions to a query and return a list of all answers