Writing A Compiler for the Finite Domain CSP Modeling Language ESRA

Mats Norberg

14th March 2006

Abstract

Current generation constraint programming languages are considered by many, especially in industry, to be too low-level, difficult, and large. A high level relational constraint modeling language, called ESRA has been constructed to solve this problem. This thesis is about the construction of a compiler for a subset of this language. Due to time limitations I have made certain assumptions on the language to make the task manageable. The compiler translates ESRA source models into SICStus Prolog, a language which have a built in constraint solver, the library CLPFD (Constraint Logic Programming, Finite Domains). ESRA uses an abstract data type for mathematical relations. In order to translate ESRA models into Prolog, a concrete representation of this data type must be chosen. In this work I represent relations with iterated list structures containing decision variables in appropriate domains (integer domains for total functions and boolean ones for non-functional relations). I also have to translate quantified expressions into recursive procedure calls.

Mats Norberg
Examensarbete 20p 2006
Datavetenskapligt program 160p
Department of Information Technology
Uppsala University

Supervisor:
Pierre Flener

Examiner:
Justin Pearson

Sammanfattning

I detta examensarbete i villkorsteknik presenteras en kompilator för ett nytt modelleringsspråk för villkorslösare.

Bakgrund

Traditionella villkorsprogrammeringsspråk anses vara för svåranvända inom många tillämpningsområden; särskilt inom industrin. Nivån är för låg och ingående kunskaper i programmering krävs för att använda sådana verktyg. Ett alternativ är att konstruera särskilda *modelleringsspråk*. Sådana språk bör:

- Ha tillräcklig uttryckskraft för att typiska villkorsproblem skall kunna formuleras inom språket.
- Vara oberoende av vilken villkorslösare vi använder.
- Vara lätt att använda, även för icke-programmerare. Problem skall kunna modelleras på hög abstraktionsnivå med hjälp av ett litet antal fundamentala primitiv.
- Inte vara beräkningsfullständigt, d.v.s. vi kan undvara sådana finesser som rekursion, loopar och kvantifiering över obegränsade universa.

En möjlighet är att bygga på *relationer*. Detta leder till så kallad *relationsmodellering*, d.v.s. man formulerar en modell där beslutsvariablerna är (matematiska) relationer. Villkoren kan då uttryckas på ett naturligt sätt med hjälp av predikatlogik.

Ett sådant språk har konstruerats vid Uppsala Universitet, Institutionen för Informationsteknologi. Språket har givits namnet ESRA (Executable Symbolism for Relational Algebra).

En kompilator för ESRA

Om ESRA modeller ska kunna exekveras av en dator måste de först kompileras till ett språk som en villkorslösare kan förstå. Det finns villkorslösare för många programmeringsspråk, t.ex. Java, C++, OCaml och SICStus Prolog. Vårt val föll på SICStus Prolog. Anledningen till att vi valde just SICStus Prolog är att forskarna här har mycket erfarenhet av det språket. Dessutom stöder SICStus Prolog en mycket stor uppsättning globala villkor.

SICStus Prolog saknar variabler av relationstyp, så en viktig del av projektet är att representera relationsvariablerna med någon form av datastrukturer. Relationer kan representeras på många olika sätt. I min kompilator väljer jag att använda matrisrepresentationer där varje relation representeras av en matris av sanningsvärden. Kompilatorn skapar bara en uppsättning representationer för varje modell och är därför *deterministisk*.

Under arbetets gång har jag stött på olika problem att lösa. Hur översätter man till exempel kvantifierade formler i ett språk utan loopar? Hur kommunicerar man mellan delrutiner i ett språk utan globala variabler? Jag har också uteslutit vissa detaljer i specifikationen av ESRA för att göra uppgiften enklare.

Den färdiga kompilatorn har testats gentemot 6 testmodeller och funnits generera korrekt kod för dessa. Jag har också gjort benchmarks för att testa effektiviteten. I vissa fall var effektiviteten sämre än vad vi hoppats på.

Fortsatt arbete

I fortsättningen kommer andra individer att vidareutveckla kompilatorn. Det finns mycket att göra.

- Implementera de element som jag har utelämnat.
- Göra kompilatorn *icke-deterministisk* och införa fler representationer för relationer. Då blir det *kompilatorns* sak att hjälpa modelleraren att hitta den mest optimala representationen för en viss klass modeller och instansdata.
- Optimeringar av den genererade koden. T.ex. kan det vara möjligt att reducera djupet i en nästlad iteration genom att använda något globalt villkor, t.ex. "scalar product."
- Automatisk detektering och brytning av symmetrier.
- Automatisk generering av implicerade villkor.

Contents

1 Introduction **6**
1.1 Constraint Satisfaction Problems (CSPs) 6
1.2 The Relational Modeling Language ESRA 7
1.3 Solving CSPs in SICStus Prolog 9
1.4 The Goal of this Thesis 11

2 Assumptions on the Source Language (a Subset of ESRA) **12**
2.1 General Assumptions 12
2.2 Domains 13
2.3 Declarations 15
2.4 Primitive Expressions 18
2.5 Application Expressions 20
2.6 Quantification 23
2.7 The Data File 25

3 Sample Models **27**
3.1 The Balanced Incomplete Block Design Problem 27
3.2 The Magic Square Problem 27
3.3 The n Queens Problem 28
3.4 The Social Golfer's Problem 28
3.5 The Progressive Party Problem 29
3.6 The Warehouse Location Problem 30

4 Translation **31**
4.1 Representing Domains 31
4.2 Representing Relations 33
4.3 The Translation Algorithm, some Preliminaries 41
4.4 Translation of Simple Expressions 44
4.5 Translation of Quantified Expressions 48
4.6 Translation of the Objective 53
4.7 Input and Output 54
4.8 The Branch Tree 56
4.9 A Complete Example 57

5 The Compiler, the Final Product **66**
5.1 Installation 66
5.2 The ESRA Libraries 66
5.3 Compiling and Running a Model 66
5.4 Using the Result API 67

6 Testing and Benchmarking **67**
6.1 Running the Test Suite 67
6.2 Some Benchmarks 68

7 Conclusion **71**
7.1 Summary 71
7.2 Related Work 71
7.3 Future Work 72

8 Grammar **74**

1 Introduction

1.1 Constraint Satisfaction Problems (CSPs)

Combinatorial problems and discrete optimization problems are becoming more and more important in various application domains from industry to finance. Important domains include *scheduling*, cost optimization, and financial problems such as *the portfolio optimization problem* [5, 11]. Problems of this type are often hard to solve efficiently. Typically they are *NP-complete*, which means that no polynomial time algorithms are available for their solutions. One technique for solving such problems is called *constraint programming over finite domains* [1].

Consider a finite set of variables, called *decision variables*, varying over finite domains and a finite set of *constraints* on these variables, i.e. a set of formulas expressible in first order predicate logic constraining the possible values of the variables. A *constraint satisfaction problem*, or CSP in brief, is the task to find a complete instantiation of the variables which satisfies all constraints. A special class of CSPs is the *constraint optimization problems* or COPs in brief. In a COP the goal is to optimize a function of the variables such that no constraint is violated.

A software system capable of solving such problems is called a *constraint solver* and the process of expressing a problem as a CSP is referred to as *modeling* the problem. In order to model a problem as a CSP one has to define the decision variables of the problem and their domains and express the constraints in terms of the chosen variables. This of course can be done in many different ways, some of them leading to more efficient solutions than others. The art of modeling is generally not very well understood and is based on experience rather than theory. A far sighted goal of constraint programming research is to develop *heuristics* that can be used by the *solver* to chose the most efficient model for a particular class of problems and instance data. But we are not there yet.

Constraint solvers first make an attempt to narrow the domains of the decision variables by noting that certain values always violate some constraints. This process is called *constraint propagation* and is very important for efficient solving of CSPs. When the domains cannot be narrowed any more the solver resorts to a search through the problem *state space* , i.e. the set of all possible instantiations of the decision variables. When an *inconsistent state* is reached, i.e. a state where some of the domains have become empty, the solver *backtracks* to an earlier state and tries new values. This process continues until either a solution is found or the entire state space becomes exhausted.

Constraint solvers can be implemented in various ways. Traditionally solvers are built into general purpose programming languages, usually an extension to a *logic programming language.* The advantage is of course that backtrack search is already a part of such a language, which makes the solver easy to integrate with the syntax and semantics of the language. Another approach, which is becoming increasingly more common, is to implement the solver as a library within a conventional programming language, for instance Java. The Koalog constraint solver (see `www.koalog.com/php/jcs.php`) is a commercial Java class library for solving CSPs. The CLPFD library [2] in SICStus Prolog is another example which is treated in section 1.3.

Yet another approach is to develop a special solver independent *modeling* *language.* This is a language which must be compiled into executable code in some other language. Such languages allows the user to model his problem on a much higher level of abstraction than most conventional languages does. Typically such a language is *declarative*, and does not aim for computational completeness. A commercial example of such a language is OPL [15, 14], a trademark of ILOG, which works in conjunction with a solver written in C++. OPL allows the model itself to be *separated from the instance data.* This is achieved by providing a separate *data file*, which provides values to variables which has been declared in the model but not assigned any value. The separation of model and data is highly desirable because it makes maintenance of a large set of models and instances much more manageable.

One problem with OPL is that it forces the modeler to chose concrete representations of the instance data and decision variables. In OPL the data are stores as arrays. Another more abstract approach is to provide an ADT[1] based view based on the mathematical concept of *relations.* This leads for instance to *relational modeling languages*, see [4] for a discussion of the benefits of relational modeling. It is now the responsibility of the *compiler* to chose most efficient representations of the abstract data types based on some heuristics [9, 17], relieving the modeler from this task.

1.2 The Relational Modeling Language ESRA

The research group ASTRA[2] (`http://www.it.uu.se/research/group/astra`) at Uppsala University, has developed a relational modeling language called ESRA[3]. This section serves as a brief introduction to this language. Fur a full discussion see [4] for a general introduction and [3] for a technical discussion of the syntax and semantics of ESRA. There is also a predecessor of ESRA based on functional variables, here called *functional* ESRA. There already exists a compiler, written in Java, compiling functional ESRA into OPL [17]. See also [9] for a general discussion of how to represent functional variables and *heuristics* for choosing the most suitable one for a certain class of problems and instance data. This MSc thesis is an extension of this work to relational ESRA.

The language is based on the concept of *relations.* A relation between two finite sets A and B is a set of *ordered pairs* (a, b) where $a \in A$ and $b \in B$. More generally a relation between n sets $A_1, A_2, ..., A_n$ is a set of *ordered tuples*, or *tuples* in brief, $(a_1, a_2, ..., a_n)$ such that $a_1 \in A_1, a_2 \in A_2, ..., a_n \in A_n$.

We also introduces the concept of *multiplicity* or *cardinality constraint* on a relation. The notation

$$\text{var } R : A \ {}^{m}\!\times^{n} \ B$$

where m and n are non negative integers, declares R as a relation between the sets A and B, such that every element in A is related to exactly m elements in B, while every element in B is related to exactly n elements in A. As a generalization we may allow m and n to be sets. The expression

$$\text{var } R : A \ {}^{M_1}\!\times^{M_2} \ B$$

[1]Abstract Data Type
[2]Analysis, Synthesis, and Transformation/Reformulation of Algorithms
[3]Executable Symbolism for Relational Algebra

where M_1 and M_2 are finite, non negative integer sets, declares R as a relation between A and B such that each element in A is related to m_1 elements in B, where $m_1 \in M_1$, and each element in B is related to m_2 elements in A, where $m_2 \in M_2$.

The constraints are expressed by a relational calculus based on predicate logic with counting. Rather than giving a formal specification here I will clarify the language with an example. For a full specification see [3].

Consider the *balanced incomplete block design problem*. Let V be any set of v elements, called *varieties*. A *balanced incomplete block design* (BIBD) is a bag of b subsets of V, called *blocks*, each of size k (constraint C_1), such that each pair of distinct varieties occurs together in exactly λ blocks (C_2), with $2 \leq k < v$. An implied constraint is that each variety occurs in the same number of blocks (C_3), namely $r = \lambda(v-1)/(k-1)$. A BIBD is parameterized by a 5-tuple $\langle v, b, r, k, \lambda \rangle$. Originally intended for the design of statistical experiments, BIBDs also have applications in cryptography and other domains, for instance *the portfolio optimization problem* [11]. See Problem 28 at `http://www.csplib.org` for more information. I now present an ESRA model for this problem.

dom $\mathit{Varieties}, \mathit{Blocks}$
cst $r, k, \lambda \in \mathbb{N}$
var $\mathit{BIBD} \in \mathit{Varieties}\ {}^{r}\!\times^{k}\ \mathit{Blocks}$
solve
  $\forall (v_1 < v_2 \in \mathit{Varieties})\ \mathrm{count}(\lambda)(j \in \mathit{Blocks} \mid \mathit{BIBD}(v_1, j) \wedge \mathit{BIBD}(v_2, j))$

Let me step through this model line by line. The line

$$\text{dom } Varieties, Blocks$$

declares the domains $Varieties$ of varieties and *Blocks* of blocks as instance data. The actual domains must be supplied via a data file at run time. The next line

$$\text{cst } r, k, \lambda \in \mathbb{N}$$

declares 3 run time initialized natural number constants (the two remaining $BIBD$ parameters v and b are implicitly given as the cardinalities of $Varieties$ and *Blocks*). Now the instance data are done with; next I declare the single relational decision variable $BIBD$.

$$\text{var } \mathit{BIBD} \in \mathit{Varieties}\ {}^{r}\!\times^{k}\ \mathit{Blocks}$$

This line declares a relation between $Varieties$ and *Blocks* with multiplicity constraints r and k. The interpretation of the atom $BIBD(v, b)$ is that variety v is in block b. Note that the multiplicities on the relation $BIBD$ automatically takes care of constraints C_1 and C_3. This way to express cardinality constraints *declaratively* is a very powerful feature of ESRA. The declarations are now done with and the constraints follow after the keyword ‘solve’. Consider the line

$$\forall(v_1 < v_2 \in \textit{Varieties})\ \mathrm{count}(\lambda)(j \in \textit{Blocks} \mid \textit{BIBD}(v_1, j) \wedge \textit{BIBD}(v_2, j))$$

This is a counting expression nested within a universal quantifier. Let me describe the counting expression first. The formula

$$\mathrm{count}(\lambda)(j \in \textit{Blocks} \mid \textit{BIBD}(v_1, j) \wedge \textit{BIBD}(v_2, j))$$

means: there's exactly λ values of $j \in$ *Blocks* such that $\textit{BIBD}(v_1, j) \wedge \textit{BIBD}(v_2, j))$, i.e. there's exactly λ blocks in *Blocks* that contain both v_1 and v_2. Finally when putting back the counting expression under the universal quantifier we get the meaning: for all pairs of distinct varieties v_1 and v_2, there's exactly λ blocks containing both v_1 and v_2. This is just constraint C_2 which concludes the discussion of the *BIBD* problem.

1.3 Solving CSPs in SICStus Prolog

1.3.1 Prolog and CLPFD

Prolog is a *logic programming* language, originally designed for processing of natural languages. To understand this report the reader should have a thorough understanding of this language. SICStus Prolog is a trademark of SICS [4] (`http://www.sics.se`), a commercial implementation of the Prolog language. The library CLPFD, Constraint Logic Programming over Finite Domains, is a sub system of SICStus Prolog [2], including a constraint solver. Because I will make frequent use of this library throughout this document I will give a brief introduction in this section.

Solving a CSP in SICStus Prolog consists of two phases. First we have to model the problem by *posting constraints* to the solver; this is the *initialization phase*. In the second phase, the *labeling phase*, we invoke the solver usually by a call to the labeling/2 predicate. The solver will attempt to find a solution to the CSP. When a solution is found it is printed and the user may try to find alternative solutions by backtracking. Prolog answers no if there are no more solutions. Now I give a description of all CLPFD predicates and constraints I will use throughout the report. The reader will find additional documentation on line `http://www.sics.se/sicstus`.

1.3.2 Decision Variables and their Domains

Decision variables are tied to domains. CLPFD has no complex or aggregate decision variables, so all decision variables range over integer sets, denoted by *range expressions* which are either intervals or enumerated sets like {1,2,3}. An interval expression, e.g. 1..10, denotes the set of all integers between the lower and upper endpoint respectively. There are also infinite intervals, e.g. 1..sup and inf..7, where sup and inf stand for plus infinity and minus infinity respectively. Sets are also denoted by enumeration of elements with the usual mathematical syntax, e.g. {1, 4, 7, 11, -5}. We ties range expression to the decision variable X by the goal: 'X : Range'. To declare X as a natural number variable use the following goal:

[4] Swedish Institute for Computer Science

Val is an integer, List is a list of integers or decision variables, Count an integer or a decision variable, and RelOp is a relational symbol as in section 1.3.3. True if N is the number of elements of List that are equal to Val and N RelOp Count.

As an example suppose Xs is a list of boolean values, zeros and ones, and we want to state the conjunction of all values. The following goal makes the trick:

```
length(Xs, NumElements),
sum(Xs, #=, NumElements).
```

The explanation is that the sum is equal to the number of elements if and only if each element is equal to one.

1.3.5 Labeling

When all constraints have been stated (posted) it is time to ask the solver to start a *backtrack search* for a solution to the problem. This is often achieved by a call to the built in predicate labeling/2. Consider first a pure CSP with decision variables in the list Xs. The goal

```
labeling([], Xs).
```

starts a backtrack search for the solution. When a solution has been found it is printed by the system and the user is prompted for (at the toplevel only) backtracking to find alternative solutions. Now consider a COP with the objective function bound to the variable M and the decision variables we are doing labeling on in the list Xs. The goal

```
labeling([maximize(M)], Xs).
```

tries to find a solution where M is maximized. The goal

```
labeling([minimize(M)], Xs).
```

searches a solution where M is minimized.

1.4 The Goal of this Thesis

The goal of this MSc thesis is to develop a compiler for relational ESRA, compiling models into executable SICStus Prolog programs. The compiler shall use a binary matrix representation of relational variables with an optimization for total functions. I was given free hand making any reasonable simplifying assumption on the source language which would help me to complete the task. The compiler is implemented in Objective Caml, a functional programming language in the ML family [10] using a parser already developed by Magnus Ågren.

In the rest of the thesis I first give a detailed definition of the source language *actually* implemented, section 2. Some examples of ESRA models can be found in section 3. In section 4 a detailed account of the actual translation algorithm is given. Section 5 is a short summary of how to use the final program. Finally some tests and benchmarks are presented in section 6.

2 Assumptions on the Source Language (a Subset of ESRA)

The purpose of this section is to define the source language of the compiler. All differences between the source language and the full ESRA specification are presented here.

2.1 General Assumptions

Most assumptions on the source language appear throughout the rest of this major section, but there are some outstanding issues that don't fit elsewhere. These issues are addressed in this sub section.

2.1.1 Syntactic Issues

There's only a few points where I disagree with the syntax of full ESRA.

- Identifiers must begin with a letter of either case (*may start with lower case*). The rest of the identifier consists of letters, digits or underscores. *Quoted ascii strings are not supported.* Moreover so called *symbolic constants*, *enumerations* or *atoms* or whatever they are called, *are not supported.*
- Using a naked `#,` `->` or `+>` in relational domain expressions is not allowed. The square brackets in `[#]` etc are mandatory. The reason is that the inclusion of the naked operators makes the grammar unparsable. So for instance the expression `D -> D` is illegal but `D [->] D` is correct.
- In expressions like `forall(i:D)(<formula>)` or `sum(i:D)(<expression>)`, there are mandatory parentheses around `<formula>` and `<expression>`. Trying to discard these parentheses makes the grammar unparsable.

2.1.2 No Set Comprehensions

In the full ESRA specification, there are so called *set comprehensions*, that is a set given by a filtering condition on a variable. For instance $\{i \in D | i < 10\}$ is a set comprehension meaning all elements in D which are less than ten. *Set comprehensions are not supported!*. Implementing set comprehensions require some loop constructions, similar to the ones used to implement quantified expressions. My original intention was to support set comprehensions but now I'm left with no more time so I have to forbid them.

The card/1 operator to take the cardinality of a set is most typically used in conjunction with set comprehensions, so I decided to omit this operator as well. There is a way to simulate the card/1 operator using a sum expression. For instance

```
card {i:D | F(i) < 100}
```

may be replaced by

```
sum(i:D | F(i) < 100) (1).
```

The trick is to sum a number of ones, one for each element in `{i:D | F(i) < 100}`.

2.2 Domains

2.2.1 Primitive Domains

Primitive domains contain primitive values only. I recognize two kinds of primitive values: integers and booleans. In particular *tuples are not primitive values*. Integers and booleans must not be intermixed with each other in the same domain. A primitive domain is thus either a subset of the integers or a subset of the booleans `{true, false}`. Observe that the atoms inf and sup are not integers. They just represent the lower and upper endpoints of an unbounded interval.

Primitive domains *must not contain decision variables* , but may contain run time initialized constants (see 2.3.2) and expressions depending on such values. I say that domains are *ground* , i.e. they are fully instantiated at run time before the problem is passed to the solver.

A primitive domain may be denoted extensionally as a list of values enclosed in braces, for instance `{1, 5, 7, -1}`. It may also be given intensionally as an interval with lower and upper endpoints separated by `'..'`, for instance `1..sup`. The atoms `'int'` and `'nat'` are syntax sugar for `inf..sup` and `0..sup` respectively. The singleton interval `n..n` may be abbreviated to `n`, but *only when it is used as a multiplicity set*. It is *not allowed for an interval to have an upper bound that is less than its lower bound*. The bounds of an interval must either be *ground* integer values or one of the atoms inf and sup.

To illustrate the rules I now give examples of several correct and incorrect primitive domain expressions.

```
    {1, 7, 5, -11, 2}
```

Correct! This is a domain containing 5 integers.

```
    {1, g, 5, -11*d, 2}
```

Correct under the assumption that `g` and `d` has previously been declared as integer constants.

```
    {true, false}
```

Correct! This is the *boolean domain* .

```
    {1, 2, sup, -12}
```

Illegal! This domain contains the illegal atom `'sup'`.

```
    {1, 5, true}
```

Illegal! This domain is *inhomogeneous* .

```
    1..false
```

Illegal! The atom 'false' is not allowed as end point to an interval.

```
    {1..10}
```

Syntax error! Interval expression must not be enclosed in braces.

```
     {1, (2,3), 4}
```

Illegal! This domain contains a tuple.

2.2.2 Relational Domains

Relational domains are constructed from primitive domains by the binary infix operator `[#]` and the unary suffix operator `[]`. To define their semantics I must first define the notion of a *multiplicity set*. A multiplicity set is just a finite domain of ground non-negative integer values.

Assume that `M1`, `M2` are multiplicity sets and `D1`, `D2` are primitive domains. The *relational domain*

```
D1 [M1#M2] D2
```

is the domain of *relations* in the cartesian product of `D1` and `D2`, such that each element in `D1` is related to m1 elements in `D2` where m1 $\in$ `M1` and each element in `D2` is related to m2 elements in `D1` where m2 $\in$ `M2`.

Assume that `D` is a primitive domain and `M` is a multiplicity set. The *set domain*

```
D[M]
```

is the domain of subsets of `D` with cardinalities in `M`.

I also introduce some handy short notations. `[#]`, `[->]` and `[+>]` are syntax sugar for `[0..sup # 0..sup]`, `[1 # 0..sup]` and `[{0,1} # 0..sup]` respectively. These are the domains of unconstrained relations, total functions and partial functions respectively.

Now I have to worry a bit about what happens when these expressions are nested. What is the meaning of `(D1[#]D2) [#] (D3[#]D4)`? Obviously it makes no sense to interprete this as a set of relations between relations because ESRA has no higher order objects, so I have to interprete it as the set of relations in the four fold cartesian product `D1` $\times$ `D2` $\times$ `D3` $\times$ `D4`. The parentheses in relational expressions should not have any essential semantic significance. Parentheses however are important for the parser. The operator `[#]` associates from left to right so `D1 [#] D2 [#] D3` parses as `(D1 [#] D2) [#] D3`. So what significance has the parse tree if I am going to flatten it out anyway?

First I have to introduce the idea of the *primary functor* of a relational domain expression. It is just the root of the parse tree. So for instance the primary functor of `(D1 [->] D2) [#] (D3 [->] D4)` is `[#]`. I say that a non-trivial *cardinality constraint* is attached to a [`M1#M2`] operator, if one of the multiplicity sets is different from the (*trivial*) set 0..sup. A cardinality constraint of 0..sup (the naturals) is called a *trivial constraint*, because it is really not a constraint at all.

Now I make the following assumptions on relational domains:

- Each relational domain expression may be completely factorized into a cartesian product of primitive domains.
- Non trivial cardinality constraints may only be given to the primary functor of the relational domain parse tree.
- The relational domain parse tree is conceptually flattened to a depth of two levels. I will refer to the primitive domains on the left side of the

primary functor as the *left hand side domains* or for short the LHS domains. Similarly the domains to the right of the primary functor are the *right hand side domains* or RHS domains. I will call the number of LHS domains the *left arity* of the relational domain. Similarly the number of RHS domains is the *right arity* of the relational domain. A relational domain is a *domain of total functions* if the left hand side multiplicity set is $\{1\}$.

The reasons for flattening out the sub structure of the relational domain parse tree are threefold. Firstly I am not quite sure what this sub structure should mean in a language without tuple values. Secondly I do not think such a sub structure is really needed for relational modeling. Thirdly by ignoring such sub structure the task of implementation is vastly simplified.

With this simplification any relational domain can be compiled into a quadruple (LeftDoms, m1, m2, RightDoms) where LeftDoms and RightDoms, the LHS and RHS domains respectively, are lists of primitive domains.

I now give some examples of correct and illegal relational domain expressions.

```
D1 [2#1] D2
```

Correct! This is a domain of relations in `D1 x D2` which relates each element in `D1` with exactly two elements in `D2` and each element in `D2` with exactly 1 element in `D1`.

```
1..10 [->] nat
```

Correct! This is a domain of total functions from 1..10 to `nat`.

```
(1..5 [2#] D) [->] nat
```

Illegal! A non trivial cardinality constraint on a non primary functor.

```
1..10[nat]
```

Correct! This is the domain of all sub sets of `1..10`.

2.3 Declarations

The purpose of declarations in any formal language, is to bind identifiers to their domains (or *types*). In my implementation of ESRA I require that *identifiers be declared before they are used*. By this assumption it is possible to implement the compiler in one pass and I need not to worry about circular dependency of constant definitions. *Only one identifier may be declared in the same declaration.* In ESRA there are 3 kinds of declarations: domain, constant and variable declarations.

2.3.1 Domain Declarations

A domain declaration binds an identifier to a domain expression. A domain expression may later be referred to by this identifier. In the specification of the full ESRA language it is allowed to declare an un specified domain identifier which is to be bound at run time. *This is not allowed in this implementation.* The only form of domain declaration which is recognized by the compiler is therefore

```
dom <identifier> = <domain expression>
```

Note that domain declarations begin with the keyword `'dom'`. It is also *forbidden to bind an identifier to a relational domain.* I now give examples of correct and illegal domain declarations.

```
dom Warehouses = 1..5
```

Correct!

```
dom Warehouses = {'London', 'Stockholm', 'Berlin'}
```

Illegal! Enumerated (or symbolic) constants are not supported.

```
dom Stores
```

Illegal! Does not bind the identifier to a domain at compile time.

```
dom Relations = Warehouses [#] Stores
```

Illegal! Binds a relational domain.

The reasons for these assumptions are again simplicity (and time shortage). Allowing domains to be instantiated at run time makes static type checking impossible. I disallow the case of relational domain binding for technical reasons, it complicates the flattening of the parse tree in my one pass compiler. To overcome these limitations it is necessary to let all domains be represented at run time by an object. The compiler may only refer to a domain by its name because the actual domain is unknown at compile time. A dynamic type check is now necessary each time a relation typed to this domain is referred to. I leave this work to the next developer.

2.3.2 Constant Declarations

A *constant declaration* binds a constant to an identifier. Constant declarations begin with the keyword `'cst'`. There are two types of constants in ESRA. First constants which are known at compile time. These must be bound in line by a declaration like this:

```
cst <identifier> = <value> : <domain>
```

This declaration binds the identifier to its domain at the same time as it is given its value. A consistency check is made by the compiler to ensure that the value does not violate the domain. from this point on the compiler knows the value of the constant and can substitute the value for the identifier whenever it is referred to.

The other kind of constants ESRA support are run time instantiated constants. Such constants must be instantiated at run time from a data file. The syntax for declaring a run time constant is:

```
cst <identifier> : <domain>
```

This binds the identifier to its domain. The compiler does not know the value of such constants, so it will have to refer to them by name.

Constants can be relations. There is no syntax for in line declaration of a relation, so these have to be declared as run time constants. This is just good because it makes it possible to separate the model from its instance data.

I now give examples of some correct and some illegal constant declarations.

```
cst g = 10 : nat
```

Correct! Binds the value 10 to g.

```
cst g = -10 : nat
```

Illegal! Domain error, -10 is not in `nat`

```
cst SypplyCost : (Warehouses [#] Stores) [->] nat
```

Correct! Declares instance data array for the warehouse location problem. Note that the parentheses are really necessary here.

```
cst Capacity = 8 : Warehouses [->] nat
```

Illegal! A relational constant must not be assigned a value in line.

2.3.3 Variable Declarations

A variable declaration allocates a decision variable and binds it to a domain. Variable declarations begin with the keyword `'var'`. The syntax for a variable declaration is:

```
var <identifier> : <domain>
```

Examples:

```
var v : 1..10
```

Correct!

```
var Supplies : Warehouses [#1] Stores
```

Correct! Declares a binary relation between `Warehouses` and `Stores`. This relation is many to one according to the cardinality constraint. Each store can be supplied by one warehouse only but each warehouse may supply several stores.

2.4 Primitive Expressions

Primitive expressions are built from identifiers and integer and boolean constants and variables interconnected by binary and unary operators. Quantified expressions and application expressions are not treated in this section.

2.4.1 Type Rules

There is only two kinds of primitive values: integer values and boolean values. ESRA is a typed language which means that I will not accept a value with a certain type to be used in the wrong context. Integer values are either *integer constants* like 103, or declared variables and constants belonging to an integer domain. A variable belonging to the *boolean domain* might be used in any context requiring a boolean value. An integer variable however *must not be used in a boolean context.*

In the examples that follows I assume that identifiers starting with b are boolean while those starting with n are integers.

```
b1 = 301
```

Incorrect! A boolean can't be compared to an integer.

```
(12 < 7) * 58
```

Incorrect! 12<7 is a boolean value and cannot be multiplied by an integer.

2.4.2 Numeric Expressions

Primitive numeric expressions are built from identifiers and integer constants using one of the *numeric operators* `+`, `-`, `*`, `/`, `%` (remainder) and abs (*absolute value*). The first five of these are binary infix operators while abs is a unary prefix operator. The type rules (see 2.4.1) must be respected. The result of these operations is itself a numeric value and may be used in other numeric expressions. Normal precedence rules from mathematics are respected as is grouping sub expressions by parentheses. In the following examples identifiers starting with 'n' are integer variables.

```
123 * (n1 - 2)
```

Correct!

```
abs(nx*ny - 4*(n+1)) - n
```

Correct!

```
2*n1 - true
```

Illegal! Type error.

2.4.3 Comparison Expressions

Primitive comparison expressions are built from identifiers, integer and boolean constants using one of the *comparison operators* `=, !=, <, =<, >, and >=`. Note that 'less than or equal to' is denoted by '`=<`' as in Prolog. Integer as well as boolean values can be compared using = and `!=`. I *do not support equality or inequality between complex values such as tuples and sets.* Only integers can be compared with the operators `<, >, =< and >=`. The result of a comparison is always a boolean value which cannot participate in numeric expressions. In the following examples identifiers starting with `'n'` are integer variables and those starting with other letters are of non integer types.

```
n1*33 =< 0
```

Correct!

```
true > 0
```

Illegal! Type error.

```
{1, 2, 3} = s
```

Illegal! Set expressions cannot be compared.

```
(2,3) != (6,7)
```

Illegal! Tuples cannot be compared.

2.4.4 Propositional Expressions

ESRA supports all the usual connectives of propositional logic except negation. I also support these connectives. Although negation is not supported it can be simulated by using an implication since the formula `not F` is equivalent to `false => F`. A (primitive) propositional expression is built from identifiers, integer and boolean constants and the operators /\ (conjunction), \/ (disjunction), =>, <= (implication) and <=> (equivalence). All comparison operators have higher precedence than the propositional operators (see the grammar in section 8). In the examples that follow, I assume that identifiers starting with b are booleans while those starting with n are integers.

```
n=5 => abs(n1-n1) =< 100
```

Correct!

```
n => nn*5 < 0
```

Illegal! Type error.

```
(false => b1) /\ (b2 \/ b3)
```

Correct! Emulating negation of b1.

2.5 Application Expressions

2.5.1 Total, Partial and Functional Applications

Relations are said to be *applied* to their arguments. An application is said to be *total* if the number of arguments is the same as the total *arity* of the relation (see section 2.2.2 for a discussion of arity). Suppose R is binary relation between the sets A and B. The atom $R(a, b)$, where $a \in A$ and $b \in B$, is true if and only if (a, b) is in R. Total applications create boolean values.

A *partial application* is a relational application on a subset of its parameters. With R as above the meaning of the expression $R(a)$, where $a \in A$, is the *set* $\{b \in B \mid R(a, b)\}$.

A *functional application*, finally, is an application of a *total function* on all its LHS arguments (section 2.2.2). Suppose F is a total function from the set A to the set B. The expression $F(a)$, where $a \in A$, evaluates to the unique element $b \in B$ such that $F(a, b)$. We say that a is *mapped to* b *under* F. There are two issues with this terminology which I have to explain a bit more.

First one could argue that F(a) is a functional application even if F is a *partial function*. Mathematically this is true, but I want to interpret F(a) as a number and this is impossible if the F is allowed to be partial because then F(a) may be the empty set. The empty set cannot appear in numeric expressions and we have a big trouble. Therefore I don't allow functional notation to be used on partial functions.

Secondly a functional application is technically partial, but I nevertheless will distinguish between partial and functional applications, so when I speak about the former I will always mean a non-functional partial application, resulting in a non-singleton set. Note that a functional application, when the function has an integer range, is a numeric expression, that can itself be used in bigger expressions.

2.5.2 Assumptions on Relations and Application Expressions

The following assumptions are fundamental for my implementation.

- *Partial applications are not supported.*
- Functional applications are supported on relational objects with a LHS multiplicity `{1}` (see section 2.2.2), i.e. total functions. Functional application must not be used on partial functions or on relations that are not functions.
- All *total functions are supposed to have a primitive range*, i.e. the right arity must be equal to 1. Relations with left multiplicity `{1}` but right arity bigger than 1 *are treated as if they where relations*, in particular functional application is forbidden on such objects. This rule is necessary because such applications create tuple values which does not exist in this implementation.
- A relation declared with a LHS multiplicity `{1}` and right arity 1 is always a total function. For technical reasons I cannot allow total application to be used with a function. *Use functional application in this case*!

- An unbounded domain, like 1..sup, is only allowed on the right hand side of (see section 2.2.2) a total function with right arity 1. It is *forbidden to use unbounded intervals in all other cases.*
- Each argument of a relational application must be a primitive value consistent with the domain of the relation. For instance a relation declared on the domain `1..10 [1#0..sup] 1..3` may be applied on `(5, 2)` but not on `(true, 9)`.

Now follows several examples of correct and illegal application expressions and relation declarations to clarify the rules.

```
cst F : 1..10 [->] nat
```

Correct! A total function may have 'nat' on the right side. This is just an "array" of 10 natural numbers.

```
cst F : nat [->] 1..10
```

Illegal! Unbounded domains are *never allowed on the LHS.* This would require an infinite size representation which is impossible.

```
cst R : nat [nat#1] 1..10
```

Illegal! There's a 'nat' on the left hand side. It might be possible to represent `R` as an array, but I don't allow it (time shortage! this is a special case I haven't treated – the "transposed function").

```
var R : 1..10 [#] nat
```

Illegal! The 'nat' is only allowed on total functions. This cannot be represented finitely.

```
cst F : 1..10 [->] (1..10 [#] 1..10)
```

Correct! Beware this is not treated as a function. Use relational notation here!

```
var F : 1..10 [->] nat
```

Correct! Yes, even variable functions may have an unbounded range. Beware that this can cause a runtime error if the domains have not been narrowed to finite by constraint propagation prior to the labeling phase.

```
dom D = 1..5
var R : D [#] D
var S : D[nat]
solve
   S = R(5)
```

Illegal! Partial application of R; set values cannot be compared.

```
dom D = 1..5
var F : D [1#] D
solve
     F(4, 5)
```

Illegal! F is a total function by multiplicity constraints. Use functional notation here: F(4) = 5.

```
dom D = 1..5
var R : D [->] (D [#] D)
solve
     R(1, 3, 2)
```

Correct! Relational notation is correct because R is not treated as a total function. It has right arity 2. R(2) = (3, 4) is forbidden notation.

```
var R : {1,2} [#] 1..10
solve
     R(1, 2, 5)
```

Illegal! To many arguments.

```
var F : 1..10 [->] 1..10
solve
     F(2)*F(3) + 4 =< 100
```

Correct! The values of F are integers, hence they may be used in a numeric expression.

```
var F : 1..10 [+>] 1..10
solve
     F(5) = 5
```

Illegal! Functional notation illegal on partial functions because the type of the function "image" is indeterminate (may be the empty set). Use relational notation here: F(5, 5).

2.5.3 The Problem with Unbounded Domains

The presence of the keywords `nat` and `sup` in the source language presents some difficulties to the implementor. This means that domains can be infinite in certain circumstances. It was clear early on that `nat` often occurs on the RHS of function declarations (see for instance the Progressive Party Problem in section 3.5). I was forced to accommodate this case somehow without having to resort to infinite representations which is quite impossible.

Suppose that we want to represent a function declared on the domain `1..10 [->] nat`. In a pure binary matrix representation this would require that we allocate a ten by infinity boolean matrix which of course is impossible. To overcome this problem I invented what I call *the total function optimization*. This means that total functions (with right arity 1) are represented by an integer matrix with a depth equal to the left arity of the function. Now it is possible to index on the LHS arguments of the function and get a value which is in the range of the function.

Another problem with unbounded domains is that they might render the state space of the problem infinite. Suppose that a primitive variable `X` is defined on the domain 1..sup. If `X` appears in the set of variables sent to `labeling/2` and its domain hasn't been narrowed down to a finite size by constraint propagation, then an exception is raised by the Prolog system: “instantiation error in the first argument to labeling/2”. I ignore this problem. *It is the responsibility of the user to make sure that such a situation does not occur.*

Yet another difficulty is that quantification over non finite domains is never acceptable. But that is taken care of by the type checking routines.

In a fuller implementation of ESRA there might be *set comprehensions*, e.g. $\{i : D \mid F(i) < 10\}$. This presents a new range of difficulties, for instance the compiler might not even be able to determine if such a set is finite or not at compile time. It might even be impossible, in some cases, to determine the finiteness of a set at all. Due to such difficulties I decided not to support set comprehensions in this implementation.

2.6 Quantification

2.6.1 Local Variable Specifiers

Quantifiers bind variables within a scope depending on the syntax of the respective quantifier. These variables are called *local variables* and are declared by what I call a *local variable specifier*. Each local variable ranges over a primitive domain. A local variable specifier is just a comma separated sequence of `<variable>:<domain>` expressions, for instance `v1:D1, v2:D2, v3:D3`.

When several local variables range over the same domain they can be declared as a single ampersand separated sequence like `v1&v2&v3:D`.

Yet another format is possible. With `v1<v2:D` we mean all possible pairs of `v1` and `v2` with `v1<v2` where `v1` and `v2` both ranges over the primitive domain `D`. The `<` may be substituted by one of the other comparison operators. Observe that both sides of the `v1<v2` expression must be a local variable.

An important point to stress is that each local variable ranges over a ground, primitive and *finite* domain. *Infinite domains like 1..sup are not allowed in conjunction with quantification.* This is in accord with the original specification of ESRA. Now I give some examples of correct and illegal local variable specifiers.

```
i : D1, j : D2
```

Correct!

```
i*i-1 : D
```

Illegal! A general expression is not allowed in this context.

```
i<10 : D
```

Illegal! Not a local variable: ’10’.

```
i&j : D, i : E
```

Illegal! The local variable ’i’ must not be redefined. within the same scope.

2.6.2 The Forall Expression

The universal quantifier is denoted by the keyword `'forall'` in ESRA. The general syntax is:

```
forall(<lclvarspec> | <condition>) (<formula>)
```

Here `<lclvarspec>` is a local variable specifier, `<condition>` a formula and `<formula>` another formula. The meaning is: for all instantiations of the local variables according to `<lclvarspec>` such that `<condition>` is true, the `<formula>` has to be true. The `<formula>` of course depends on the local variables.

The `<formula>` *has to be enclosed in parentheses.* This has to be done because taking it away from the grammar makes it un parsable. For instance the expression `forall(i:D) f b` introduces a conflict. Should the function application f b be reduced before the forall expression or vice versa. Some examples follow:

```
forall (i:D) (true)
```

Correct!

```
forall(i:D) i*i < 10
```

Illegal! The formula must be enclosed in parentheses.

2.6.3 The Count Expression

The count quantifier is a generalization of the existential quantifier of predicate logic. It has the syntax:

```
count(<multiplicity>) (<lclvarspec>| <condition>)
```

Here `<multiplicity>` is a multiplicity set (see section 2.2.2), and `<condition>` a formula. The meaning is that the set of all possible instantiations of the local variables according to `<lclvarspec>` such that `<condition>` is fulfilled, has a cardinality which is in the set `<multiplicity>`. Syntactically a singleton multiplicity set `{<expression>}` may be abbreviated to `<expression>`. The expression:

```
exists (<lclvarspec> | <condition>)
```

is syntax sugar for

```
count(1..sup) (<lclvarspec> | <condition>)
```

Here are some examples.

```
count(1) (i : D | i < 10)
```

Correct! There is exactly one `i < 10` in `D`.

```
count(0) (I:D | i*5)
```

Illegal! `i*5` is not a formula.

2.6.4 The Sum Expression

The sum operator is not a quantifier, but has a similar syntax. It operates on numeric expressions rather than on formulas. The syntax is:

```
sum (<lclvarspec> | <condition>) (<numexpr>)
```

Here `<lclvarspec>` is a local variable specifier, `<condition>` a formula and `<numexpr>` a numeric expression. The meaning is: substitute each local variable in `<numexpr>` with its value for each possible instantiation of `<lclvarspec>` such that `<condition>` is true. Then sum all resulting values to obtain the value of the entire expression. This is just the usual sum notation used in mathematics. Examples:

```
sum (i:1..10) (i)
```

Correct! Computes the sum of all numbers between 1 and 10.

```
sum (i : 1..10 | f(i)%2 = 0) (f(i))
```

Correct! Sum all even `f(i)` where `i` is between `1` and `10`.

```
sum (i : D) i*i-1
```

Illegal! The summand has to be enclosed in parentheses.

2.7 The Data File

2.7.1 What is the Data File?

One main objective of relational modeling is to separate the model from the instance data. The commercial modeling language OPL [14] achieves this by providing a separate data file, where all instance data are supposed to be declared. ESRA uses the same paradigm. Each constant which is declared but not instantiated (section 2.3.2) must get its value from the data file. The language of the data file is constructed within Prolog, so the data file is really a Prolog program. The data file is a sequence of *data constructors*, each of them a Prolog term terminated by a full stop. Primitive variables are instantiated by primitive data constructors, while relation variables get their values from what I call *matrix data constructors*.

2.7.2 Primitive Data Constructors

A primitive data constructor is a single line of the form `<identifier> = <value>`. The value must be literal and not an expression or another variable. Although the compiler allows variable names to begin with a lower case letter, Prolog does not so, I require *each runtime initialized constant to begin with a capital letter.* Otherwise Prolog will give you an exception. The data file, as it is implemented now, do not distinguish between integers and booleans, so the correct way to instantiate a boolean is to assign it the value 0 or 1. If you try to put the line `B = true.` in the data file Prolog will crash your program. The program behaves this way because I have not had time to change it. You will probable not instantiate boolean variables in the data file very often.

2.7.3 The Matrix Data Constructor

A matrix data constructor has the following syntax:

```
matrix( <list> ).
```

Here <list> is a list of lists of ... of lists (a "matrix") of integer values. Total functions, which must have right arity 1 (see section 2.2.2), are represented by an integer matrix with a depth of left arity, where the integer values range over the range of the function. Relations which are not total functions are represented by binary matrices (matrices with elements 0 or 1) with a depth of the full arity of the relation. For example the following matrix data constructor

```
R = matrix([[1,0,1],
            [0,0,1],
            [1,1,0]]).
```

assigns R a value of a binary relation between two sets of cardinality 3. The actual domains and cardinality constraints cannot be specified in the data file but must be declared in the model itself. See section 4.2 for more information on matrix representations of relations.

The following example defines a total function of left arity 1, i.e. a *one dimensional array.*

```
Array = matrix([2, 7, 11, 5, 11, 0, 38]).
```

As with relations the actual domain of Array is not defined here but in the model itself.

2.7.4 An Example

As an example I now give a sample data file for the warehouse location problem, to be presented in section 3.6.

```
NStores = 10.

NWareHouses = 5.

FixedCost = 30.

Capacity = matrix([1, 4, 2, 1, 3]).

SupplyCost = matrix([[20, 24, 11, 25, 30],
                     [28, 27, 82, 83, 74],
                     [74, 97, 71, 96, 70],
                     [2, 55, 73, 69, 61],
                     [46, 96, 59, 83, 4],
                     [42, 22, 29, 67, 59],
                     [1, 5, 73, 59, 56],
                     [10, 73, 13, 43, 96],
                     [93, 35, 63, 85, 46],
                     [47, 65, 55, 71, 95]]).
```

The first two lines define the number of stores and the number of warehouses respectively. The third line is the fixed maintenance cost per warehouse. The fourth line is the array of capacities of each warehouse. Finally the fifth line defines the matrix SupplyCost which gives the cost for each warehouse to supply each store.

3 Sample Models

All sample models are parts of my major test suite. All these models with suitable instance data compiles and runs on my system (see section 6.1). The sample models will be used as examples throughout the rest of this document.

3.1 The Balanced Incomplete Block Design Problem

Let V be any set of v elements, called *varieties*. A *balanced incomplete block design* (BIBD) is a bag of b subsets of V, called *blocks*, each of size k (constraint C_1), such that each pair of distinct varieties occurs together in exactly λ blocks (C_2), with $2 \leq k < v$. An implied constraint is that each variety occurs in the same number of blocks (C_3), namely $r = \lambda(v-1)/(k-1)$. A BIBD is parameterized by a 5-tuple $\langle v, b, r, k, \lambda \rangle$ of parameters. Originally intended for the design of statistical experiments, BIBDs also have applications in cryptography and other domains. See Problem 28 at `http://www.csplib.org` for more information.

This problem may be modeled in ESRA as below. Note that declaring the domains `Varieties` and `Blocks` and the single decision variable `BIBD` automatically takes care of constraints C_1 and C_3. The interpretation of the atom `BIBD(v, b)` is simply that variety v is in block b. The constraint C_2 is taken care of by the formula in the solve clause of the model.

```
cst V : nat
cst K : nat
cst Lambda : nat
cst B : nat
cst R : nat
dom Varieties = 1..V
dom Blocks = 1..B
var BIBD : Varieties [R#K] Blocks

solve
    forall (v1 < v2 : Varieties)
       (count (Lambda) (i : Blocks | BIBD(v1,i) /\ BIBD(v2,i)))
```

3.2 The Magic Square Problem

A *magic square* is an n by n grid of *distinct* integers, such that the sum of the integers in each row, each column and the two main diagonals always is the same. One can show that the constant value of the sum is $(n^2+1)n/2$. Below is an ESRA model for the problem to find such a magic square. The possible values are between 1 and n^2 (because the integers are distinct). In the model below N is the size of the grid, Sequence the set 1..N, Val the set of integer values and

S, the unique decision variable is a total function (array) mapping the set of grid squares on their values. The four sums in the solve clause are the sum of each row, each column and the two diagonals respectively. Here N is a run time constant that must be supplied by a data file.

```
cst N : 1..sup
dom Sequence = 1..N
dom Val = 1..N*N
cst SumVal = (N*N+1)*N/2 : nat
var S : (Sequence [#] Sequence) [->1] Val

solve
   forall (I : Sequence)
          ((sum(J : Sequence) (S(I,J)) = SumVal)
          /\
          (sum(J : Sequence) (S(J,I)) = SumVal))
          /\
          sum (I : Sequence) (S(I,I)) = SumVal
          /\
          sum (I : Sequence) (S(I,N-I+1)) = SumVal
```

3.3 The n Queens Problem

The n queens problem is the problem how to place n queens on a $n \times n$ chessboard such that no queen attacks another queen. To model this problem we declare a bijection Q of the rows. The interpretation is that if $Q(i) = j$ a queen stands on the square (i, j) on the chessboard. Note that the bijectivity of Q is taken care of by the cardinality constraint [->1]. By modeling it is clear that each pair of queens stand on different rows and columns, so the only remaining requirement is that they must also stand on different diagonals. The two inequalities in the solve clause of the model below takes care of that.

```
cst N : nat
dom Rows = 1..N
var Q : Rows [->1] Rows

solve
    forall (I<J : Rows)
           ((Q(I) - I != Q(J) - J) /\
           (Q(I) - J != Q(J) - I))
```

3.4 The Social Golfer's Problem

In a golf club, there are N players, each of whom plays golf once a week (constraint C_1) and always in G groups of size S (C_2), hence $N = GS$. The objective is to determine whether there is a schedule of W weeks of play for these golfers, such that there is at most one week where any two distinct players are scheduled to play in the same group (C_3). An implied constraint is that every group occurs exactly SW times across the schedule (C_4). See Problem 10 at `http://www.csplib.org` for more information.

The instance data can be declared as the three natural-number constants G, S, and W, as well as the three domains *Players*, *Weeks*, and *Groups*, as below. A unique decision variable, *Schedule*, can then be declared, immediately taking care of the constraints C_1 (because of the totality of the function) and C_4. The first forall expression in the solve clause takes care of the constraint C_3 while the second one takes care of C_2. The three run time constants G, S and W must be defined in the data file.

```
cst G : nat
cst S : nat
cst W : nat
dom Players = 1..G*S
dom Weeks = 1..W
dom Groups = 1..G
var Schedule : (Players [#] Weeks) [->{S*W}] Groups

solve
    forall (P_1 < P_2 : Players)
            (count (0..1) (V : Weeks | Schedule(P_1,V) = Schedule(P_2,V)))
    /\
    forall (H : Groups, V : Weeks)
            (count(S) (P : Players | Schedule(P,V) = H))
```

3.5 The Progressive Party Problem

The problem is to timetable a party at a yacht club. Certain boats are designated as hosts, while the crews of the remaining boats are designated as guests. The crew of a host boat remains on board throughout the party to act as hosts, while the crew of a guest boat together visits host boats over a number of periods. The spare capacity of any host boat is not to be exceeded at any period by the sum of the crew sizes of all the guest boats that are scheduled to visit it then (constraint C_1). Any guest crew can visit any host boat in at most one period (C_2). Any two distinct guest crews can visit the same host boat in at most one period (C_3). See Problem 13 at `http://www.csplib.org` for more information.

The three run time constants `NumGuests`, `NumHosts` and `NumPeriods` are declared as natural numbers and the domains `Guests`, `Hosts` and `Periods` are declared as below. A unique functional decision variable, `Schedule`, can then be declared as below. The first forall formula in the solve clause takes care of constraint C_2, the second of C_1 and the last one of C_3.

```
cst NumGuests : nat
cst NumHosts : nat
cst NumPeriods : nat

dom Guests = 1..NumGuests
dom Hosts = 1..NumHosts
dom  Periods = 1..NumPeriods
cst SpareCapacity : Hosts -> nat
cst CrewSize : Guests -> nat
var Schedule : (Guests [#] Periods) -> Hosts

solve
    forall (G : Guests, H : Hosts)
         (count(0..1) (P : Periods | Schedule(G,P) = H))
    /\
    forall (P : Periods, H : Hosts)
         (sum(G : Guests | Schedule(G,P) = H) (CrewSize(G))
           =< SpareCapacity(H))
    /\
    forall (G_1 < G_2 : Guests)
         (count (0..1)(P : Periods | Schedule(G_1,P) = Schedule(G_2,P)))
```

3.6 The Warehouse Location Problem

At last I give an example of a constraint optimization problem (COP). Suppose that a number of warehouses are supporting a set of stores. Each store must be supplied by exactly one warehouse (constraint C_1). Each warehouse has a capacity telling the maximum number of stores it can supply (constraint C_2). Each warehouse has a fixed maintenance cost and a supply cost for each store. The problem is to determine which warehouses should be open and which ones should be closed such that the total maintenance cost be as small as possible.

The model below is essentially the same as the one presented in the OPL book [14]. The auxiliary total function (array) `Open` tells which warehouses are open. `Open(i) = 1` means that warehouse number `i` is open while `Open(i) = 0` means that warehouse number `i` is closed. The instance data consists of the arrays Capacity and SupplyCost as well as the numeric constants NWareHouses, NStores and FixedCost. The binary integer decision variable Supply tells weather warehouse w supplies store s or not. It must be an integer function rather than a binary relation, because we want to do arithmetic on it. A sample data file can be found in section 2.7.4. The cost function can now be modeled as below using Open(w) as a weight factor. The first forall expression in the such that clause takes care of constraint C_1, the second one is a channeling constraint ensuring the consistency of the Open array, while the third takes care of constraint C_2.

```
cst NStores : nat
cst NWareHouses : nat
dom Stores = 1..NStores
dom Warehouses = 1..NWareHouses
cst Capacity : Warehouses [->] nat
cst SupplyCost : (Stores [#] Warehouses) [->] nat
cst FixedCost : nat
var Supply : (Stores [#] Warehouses) [->] {0, 1}
var Open : Warehouses [->] {0,1}


minimise
    sum(W : Warehouses) (FixedCost * Open(W)) +
        sum(W : Warehouses, S : Stores) (
                SupplyCost(S,W) * Supply(S,W)
        )

    such that

        forall(S : Stores)(
                count(1) (W : Warehouses | Supply(S,W) = 1)
        )

        /\

        forall(W : Warehouses, S : Stores) (
                Supply(S,W) =< Open(W)
        )

        /\

        forall(W : Warehouses) (
                sum(S : Stores) (Supply(S,W)) =< Capacity(W)
        )
```

4 Translation

4.1 Representing Domains

Domains are static objects represented by *ground* Prolog terms. Their representations are described in this sub section. Because domains are statically instantiated in this implementation, the compiler literally generates these terms and puts them into the generated code.

4.1.1 Primitive Domains

Primitive domains are either *intervals* or *set domains* (see 2.2.1).

- **Intervals** are represented by terms like `interval(Lower, Upper)`, where `Lower` and `Upper` are the lower and upper bound respectively. The bounds can be either integer constants or one or both of the atoms `inf` and `sup`.
- **Set domains**, i.e. domains given by ESRA expressions like `{1,2,3}`, are represented by terms like `list(List)`, where List is a Prolog list of *unique* integer constants.

Before a set domain is used a run time "uniquification" is performed to ensure that the list doesn't contain any duplicate elements. This is necessary because the compiler doesn't always know all elements when the domain is first created due to run time constants. Consider the following declarations

```
cst g : nat
cst r : nat
dom D = {10, g, r, 5, r+g+1}
```

The compiler don't know the values of g and r. Suppose g and r are assigned the values 5 and 4 respectively. The set now becomes `{10, 5, 4}`. This uniquification must be done at run time when the values of g and r have become known.

Intervals are also error checked to make sure they fit the needs of the situation. For instance sometimes a *finite* interval is required. In the following examples several mappings between ESRA and Prolog are shown, assuming the `-->` sign means "maps to".

```
1..10           -->  interval(1,10)
0..sup          -->  interval(0,sup)
inf..-5         -->  interval(inf,-5)
nat             -->  interval(0,sup)
int             -->  interval(1,sup)
{1,3,5,7,9}     -->  list([1,3,5,7,9])
{1}             -->  list([1])
5..5            -->  interval(5,5)
```

4.1.2 Multiplicity Sets

Multiplicity sets are represented directly by the CLPFD range expressions they represent. So for instance {1,3,7} is actually represented by {1,3,7} and 1..sup by 1..sup. That is *the mapping from* ESRA *to Prolog is the identity mapping!*.

Special care must be taken to ensure that multiplicity sets don't contain non negative values. "Uniquification" is also necessary.

4.1.3 Relational Domains

Relational domains are represented by the term `reldom(LeftDoms, Mult1, Mult2, RightDoms)`, where `LeftDoms` and `RightDoms` are *lists of primitive domains* while `Mult1` and `Mult2` are multiplicity sets.

A special form is necessary for *set domains*, which have right arity zero. The RightDoms are replaced by `[]` and `Mult2` by the atom `nil`.

Here follow examples of the ESRA to Prolog mapping of relational domain expressions.

```
1..10[2] -->
reldom([interval(1,10)],
       {2},
       nil,
       [])
(Comment: This is a set domain, the domain of 2 element
 subsets of 1..10)

1..10 [->] 1..10  -->
reldom([interval(1,10)],
       1..1,
       0..sup,
       [interval(1,10)])

(Assume
dom D = 1..5
dom V = {1,5,7,9})
(D [#] V) [->] nat -->
reldom( [interval(1,5), list([1,5,7,9])],
        1..1,
        0..sup,
        [interval(0, sup)] )

(Assume
dom D = {1,2,3,4,5}
dom E = D)
(D [2#2] E) -->
reldom( [list([1,2,3,4,5])],
        {2},
        {2},
        [list([1,2,3,4,5])] )

(1..10 [#] 1..10) [->1] 1..100 -->
reldom( [interval(1,10), interval(1,10)],
        1..1,
        1..1,
        [interval(1,100)] )

1..10 [+>] 1..10 -->
reldom( [interval(1,10)],
        {0,1},
        0..sup,
        [interval(1,10)] )
```

4.2 Representing Relations

4.2.1 Various Representations

The representation of relational variables in SICStus Prolog is a central point in my thesis. Prolog has no aggregate decision variables, so it's clear these have to be packed into data structures of some kind. The abstract concept *array or matrix* can be implemented in several ways. The most obvious way is to use lists of lists of ... lists. There are also other possibilities such that the amatrix representation (see below). There is an implementation of "arrays" in SICStus Prolog permitting logarithmic access time using some kind of tree representation. My choice for this compiler is the list (lmatrix) representation described in the next sub section.

Which concrete representation of the abstract relation variable we choose may have a dramatic impact on the performance of the solver, so it's important to choose a good one. Unfortunately it's not easy to know a priori which is the best one, so the modeler may wish to experiment with several representations. One solution is to make the compiler *non deterministic*, which means that there will be several generated programs per input model, each one using a particular representation for certain variables in the model.

Even better would it be if the compiler itself can make the choices based on some *heuristics*, see for instance [9]. It would then be the responsibility of the compiler to choose the best representation of each decision variable relieving the modeler from this difficult and time consuming task.

I have not implemented neither a non deterministic compiler nor any heuristics, but simply make the choice deterministically based on the nature of the relation. Nevertheless before I present my solution, I wish to make a short digression here on several kinds of representations.

The linked matrix (lmatrix) representation. This is the representation actually implemented which is described in great detail in the subsequent sections. In short it means to represent the relation by a multi dimensional "matrix". The "matrix" is implemented as a list of lists of ... lists, with a depth equal to the number of dimensions of the "matrix". In the pure binary lmatrix representation each element is 0 or 1. This representation is presented in more detail in section 4.2.2. The access time to reach an element is linear in the sum of the dimensions of matrix influencing the time required to post the constraints but not the actual solving time.

The total function optimization. Suppose that we have an array of instance data, for instance [1,4,6,8,9]. It would be a waste of space to allocate a 5x5 binary matrix to store this structure. Instead we should store it as a list of integers. This is the *l matrix representation with the total function optimization.* This representation is presented in section 4.2.3.

The total function optimization in reverse. Consider a relation on the domain `D [#1] E`. This can be regarded as a total function from E to D, i.e. a "reversed" or "transposed" function. Such a function can be represented by a one dimensional array of decision variables or integers. I have not implemented this optimization.

The amatrix representation. Prolog has a set of operators to build and decompose terms. The `arg` operator, for instance, extracts the nth argument of a term in *constant time.* A matrix can be represented as a nested term. For instance the *lmatrix* `[[1,1,0], [0,1,1]]` can be represented by the structured term `m(m(1,1,0), m(0,1,1))`. The great advantage with this representation is that an element can be accessed in *constant time.* I call this representation the *amatrix representation*, where the *a* stands for `arg`. The problem with this representation is that *Prolog doesn't allow terms with more than 256 arguments.* The compiler must check the number of arguments and then decide if the amatrix representation can be used.

Using association lists. SICStus Prolog provides a library data structure called `'assoc'`, the association structure. It's a dictionary with logarithmic access time implemented as an AVL tree. It's possible to index a matrix directly by the domain elements bypassing the domain lookup step (section 4.2.5). In this representation, the matrix would be represented by a set of assoc dictionaries with the domain elements serving as keys.

The flat representation. If the cardinality of the relation (the number of tuples in the relation) is known a priori and never changes, then the relation can be represented by a flat list of elements. I call this *the flat representation.* The compiler decides if a flat representation is possible by looking at the cardinality constraints. The lmatrix representation with the total function optimization is actually a special case of the flat representation, where the matrix tree has been partially flattened reducing the depth by one. Consider a relation in `1..10 [2#] nat`. Each element on the left hand side is related to exactly 2 elements on the right hand side. This gives a totality of 20 tuples in the relation, so 20 decision variables should be allocated and stored in a flat list. Note that the `nat` on the right hand side is not a problem here. I have not implemented the flat representation.

4.2.2 The Linked Matrix Representation

A way to represent a relation is to use a multi dimensional array, or "matrix", of boolean values (zeroes or ones). So if R is an n-ary relation and M is the matrix representation of R, then $M_{a_1,a_2,..,a_n} = 1$ if and only if $R(a_1, a_2, ..., a_n)$. It's tacitly assumed here that the domains of the relations are intervals from 1 to a maximum value, so that the matrix can be indexed directly by the domain elements; but a generalization will be given in 4.2.5.

One method to represent this "matrix" is to use iterated list structures. A two dimensional matrix can be represented by a list of lists of booleans, i.e. a list containing *the rows of the matrix.* I will call this the *linked matrix* (*lmatrix* for short) representation. I will say that an lmatrix has *depth* d, if it's nested d levels deep. The *length vector* of an $A_1 \times A_2 \times ... \times A_n$ matrix is the list $[A_1, A_2, ..., A_n]$. For instance `[[[1,0], [0,1]], [[0,0], [1,0]]]` is an lmatrix of depth 3, a $2 \times 2 \times 2$ matrix more specifically with length vector `[2,2,2]`.

I now present simplified Prolog predicates for creating lmatrices and accessing their values. The real predicates, defined in the module lmatrix.pl, are a bit more complex for technical reasons, but here I want to keep the focus on what's of primary interest. The predicate `new/2` is used to create a new lmatrix while `value/3` accesses its elements.

```
%% new(+Lengths, -Matrix)
%% Length is an integer list, the length vector of Matrix.
%% Matrix must be un instantiated when new is called.
%% True if Matrix is an lmatrix with length vector
%% Lengths consisting of decision variables in 0..1
new([L], M) :-
     length(M, L),
     domain(M, 0, 1).
new([L|Ls], M) :-
     length(M, L),
     new2(L, M).
new2([], _).
new2(Ls, [M|Ms]) :-
     new(Ls, M),
     new2(Ls, Ms).


%% value(?Indexes, +Matrix, ?Value)
```

```
M = [
      [
        [[1,0], [1,1]],
        [[0,1], [1,0]]
      ],
      [
        [[1,1], [1,0]],
        [[0,1], [1,1]]
      ]
    ].
```

Figure 1: Example of a Linked Matrix. This is a $2 \times 2 \times 2 \times 2$ matrix.

```
%% True if Matrix([Indexes]) = Value
%% Procedurally: look up element indexed by Indexes.
value([I], Row, Value) :-
     lists:nth(I, Row, Value).

value([I|Is], Matrix, Value) :-
     lists:nth(I, Matrix, SubMatrix),
     value(Is, SubMatrix, Value).
```

I will spend the rest of this section developing some declarative notation used in the subsequent sections to reason about lmatrices. I will often refer to the lmatrix in Figure 1 in the following. Now I give several definitions about lmatrices.

Flattening. Let M be an lmatrix. A *flattening* of M, denoted $\mathcal{F}[M]$, is a list containing all elements of M in some order.

D-Flattening. A *d-flattening* of an lmatrix M with respect to the depth d, denoted $\mathcal{F}_d[M]$, is a list of flattenings of all sub lists at nesting depth d. Any d-flattening is an lmatrix of depth 2. I will refer to the elements of $\mathcal{F}_d[M]$ as *the rows of M with respect to the depth d.*

Projection. Let M be an lmatrix *of depth 2* with length vector $[a, b]$. The *projection of M on column j*, denoted $\mathcal{P}_j[M]$, is defined by the list comprehension $[x_i \mid x_i = M_{i,j} \wedge 1 \leq i \leq a]$. Stated in words, the projection on j is the list of all jth elements in the elements of M, or the *jth column* of M.

Transpose. Let M be an lmatrix with depth 2 and length vector $[a, b]$. The *transpose of* M, denoted, $\mathcal{T}[R]$, is defined by the list comprehension $[L_j \mid L_j = \mathcal{P}_j[M] \wedge 1 \leq j \leq b]$. In words the transpose is built by taking all first elements in the elements of M, then the second elements, and so fourth. I will call the elements of $\mathcal{T}[M]$ the *columns of* M. Note that we only define transpose for lmatrices of depth 2.

Columns. Let M be an arbitrary lmatrix. The *columns of M with respect to depth d* are the columns of $\mathcal{F}_d[M]$.

Length. The length of a list L is denoted by $\mathcal{L}[L]$. That is the number of elements in L.

Counting. Let L be a list of primitive elements. We say that $\mathcal{C}_v[L, n]$, if the value v occurs n times in L. The operator $\mathcal{C}_v$ is the *counting operator.*

The 2-flattening of the matrix M in Figure 1 is `[[1,0,1,1], [0,1,1,0], [1,1,1,0], [0,1,1,1]]`. Note how the example has been formatted to emphasize the 2-rows. The columns of depth 2 (see definition of columns above) are `[[1,0,1,0], [0,1,1,1], [1,1,1,1], [1,0,0,1]]`.

4.2.3 Optimization for Total Functions

Consider the problem of representing the array [2,3,2,1] as a total function in the lmatrix representation. The domain may be chosen as `reldom([interval(1,4)], {1}, nat, [list([2,3,1])])`. The lmatrix representing this function is

```
[
 [1,0,0],
 [0,1,0],
 [1,0,0],
 [0,0,1]
].
```

Here the second index of the lmatrix ranges over [2,3,1] rather than over 1..3, see section 4.2.5 for a discussion of domain lookup. In total 12 primitive values have to be allocated to store 4 integers. This is a gross waste of space. Only 4 integers should have to be allocated. The solution is to flatten the matrix to depth 1 and store the values directly, i.e. allocate the matrix [2,3,2,1] instead, which can be indexed directly by an element in the LHS domain 1..4. This reduction of the depth of the lmatrices is the *total function optimization.*

Turning to the general case, let F be a total function in

$$reldom(LDoms, \{1\}, Mult, Range),$$

where *Range* is a primitive domain and $d = \mathcal{L}[LDoms]$ is the left arity of F. Let M be an lmatrix of depth d, with values in *Range*, such that $F(a_1, a_2, ..., a_d) = v$ if and only if $M_{a_1,a_2,...,a_d} = v$, where $v \in Range$. It's assumed here that the LHS domains are intervals with left endpoint 1, but see 4.2.5 for a generalization.

An important point to stress is that the total function optimization is only made for *functions with right arity 1.* It would certainly be possible to represent tuple valued functions in this way also, but it's more complicated and beyond the scope of this implementation. Instead tuple valued functions have to be represented as full binary lmatrices. With this assumption, the elements in the range set is just integer values or decision variables.

The type of representation used is uniquely determined by the relational domain. If `Mult1 = {1}` and right arity $=1$, then the representation is optimized, otherwise it's binary. The Prolog predicate `get_type` below, is used to deduce the representation type.

```
%% get_type(+Mult, +RightArity, -Type)
%% True if Type is the representation type consistent
%% with Mult and RightArity.
%% Type is 'binary' for binary lmatrix and 'integer'
%% for lmatrix with the total function optimization.
get_type(Mult, RightArity, Type) :-
       ( RightArity > 1 ->
           Type = binary
       ;
           Mult = 1..1 ->
           Type = integer
       ;
           Mult = {1} ->
           Type = integer
       ;
           Type = binary).
```

4.2.4 The Cardinality Constraints

In this sub section the cardinality constrains, section 2.2.2, will be defined in the context of the lmatrix representation. I will make heavy use of the declarative notation of section 4.2.2 in this sub section.

Consider a relation R in the domain *reldom(LeftDoms, Mult1, Mult2, RightDoms)*. Assume that $\mathcal{L}[LeftDoms] = d$ and let M be the lmatrix representing R. Consider those elements of M, which are indexed by some fixed values of the left hand side indices. These elements connect one element in the cartesian product of the LHS domains to various elements on the right hand side. But these elements are precisely the elements in $\mathcal{F}_d[M]$, i.e. the d-rows of M. To affect the LHS cardinality constraint of R, it's enough to state that the number of ones in each d-row of M belongs to $Mult1$.

Now consider those elements of M, which are indexed by some fixed values of the RHS indices. These elements connect one element in the cartesian product of the RHS domains to various elements on the left hand side. These elements are in $\mathcal{T}[\mathcal{F}_d[M]]$, the d-columns of M. To affect the RHS cardinality constraint of R, it's enough to ensure that the number of ones in each d-column of M belongs to $Mult2$. I now state these assertions in a somewhat more formal language, which I hope should be obvious. The construct $A \longleftarrow B$ should be read as *A is true if B is true.*

```
Cardinality($M, LeftArity, Mult1, Mult2$) $\longleftarrow$
**Let** $d = LeftArity$
  $\forall(L \in \mathcal{F}_d[M],\ C \in \mathcal{T}[\mathcal{F}_d[M]])$
  **begin**
    $\mathcal{C}_1[L, m1]$
    $\mathcal{C}_1[C, m2]$
    $m1 \in Mult1$
    $m2 \in Mult2$
  **end**
```

Although this notation is purely declarative, the actual computation of the cardinality in lmatrix.pl follows the pseudo syntax closely. There actually are predicates named `transpose` and `dflatten`, computing $\mathcal{T}[\mathcal{F}_d[M]]$ and $\mathcal{F}_d[M]$ respectively.

It remains to discuss how to affect the cardinality constraints in the case of the total function optimization. The first thing to observe is that the LHS cardinality constraint of a total function, $\{1\}$, is implicitly taken care of by the choice of representation, because each (array) element is *uniquely* indexed by the LHS indices. Only the RHS cardinality must be enforced. Moreover, *if the RHS multiplicity set is* $\mathbb{N}$*, the constraint is trivial and may be disregarded.*

Assume F is a total function in $reldom(LDoms, \{1\}, Mult2, Range)$, where *Range* is a *primitive* domain. Let M be the lmatrix of F in the total function optimization. Let d be the depth of M, i.e. $d = \mathcal{L}[LDoms]$. Each element in $\mathcal{F}[M] \in Range$. So it's sufficient to require that for each element v in *Range*, the number of occurrences of v in $\mathcal{F}[M]$ belongs to $Mult2$. Note that we must require that the cardinality of *Range* is finite unless $Mult2 = \mathbb{N}$. The following pseudo syntax procedure enforces the cardinality constraint on total functions with finite ranges.

```
CardinalityFun($M, Range, Mult$) $\longleftarrow$
%% Range must be a finite primitive domain.
    $\forall(v \in Range)$
    begin
        $\mathcal{C}_v[\mathcal{F}[M], m]$
        $m \in Mult$
    end
```

4.2.5 Domain Lookup

So far it's have been tacitly assumed that all domains are of the form interval(1, Max). So the arguments of relational applications are mapped identically on matrix indices. In general, however, some of the domains may be other intervals or general lists of integers. When a relational application is encountered, a translation of the arguments to indices must be done, before the value can be retrieved from the lmatrix. Because this lookup must be made each time the relation is applied, *the domain should be made part of the relational object itself.* Therefore, from now on, a relational object with domain Domain and lmatrix M, will be represented by the Prolog term

```
relation(Domain, lm_rel(M))
```

Some examples will clarify the process of domain lookup.

```
(Lookup in a set domain)
F = relation(reldom([list([-3, 7, -90, 4, 55])],
                    {1},
                        0..sup,
                        [inf..sup]),
                lm_rel([4, 8, 11, -100, 9])
               ).
F(-90) = ?
lists:nth(N,[-3,7,-90,4,55],-90)
N = 3
lists:nth(3, [4,8,11,-100,9], V)
V = 11
Ergo: F(-90) = 11

(Lookup in an interval domain)
F = relation(reldom([interval(5,10)],
                        {1},
                        0..sup,
                        [0..sup]),
                lm_rel([3, 8, 2, 8, 3])
               ).
F(7) = ?
Translation: 7 => 7 + (1-5) = 3
lists:nth(3, [3, 8, 2, 8, 3], V)
V = 2
Ergo: F(7) = 2
```

To be able to translate between application arguments and lmatrix indices, a Prolog predicate `translate(?Arg, +Domain, ?Index)` must be defined. Such a predicate is implemented in lmatrix.pl. A new predicate value/4 can now be rewritten, replacing value/3 from section 4.2.2 to take care of domain lookup.

```
%% value(?Args, +Domains, +Matrix, ?Value)
%% The value predicate with domain lookup.
value([Arg], Domain, Row, Value) :-
     translate(Arg, Domain, Index),
     clpfd:element(Index, Row, Value).
value([Arg|Args], [Domain|Domains], Matrix, Value) :-
     translate(Arg, Domain, Index),
     lists:nth(Index, Matrix, SubMatrix),
     value(Args, Domains, SubMatrix, Value).
```

4.2.6 Creating Constant Relations

Section 2.7.3 introduced a matrix data constructor, used to enter constant relational objects via the data file. This section addresses the problem of parsing such data constructors. In order to parse the data correctly, the relational domain is needed. Observe that the item `M` in `matrix(M)` is already supposed to be an lmatrix, so it has only to be verified that it's consistent with the domain. The representation type is inferred by a call to `get_type`. Assume there is a predicate `parse_relation(+Matrix, +Domain, -Relation)` (actually defined in esra.pl). The signature of this predicate is:

```
parse_relation(matrix(+Matrix), +Domain, -Relation) :-
%% Parses a relational data constructor from the data file.
%% At entry relation must be un instantiated
%% True if Relation is the correct value of the constructor.
```

Now follows a step by step description of what `parse_relation` has to do.

- Decompose Domain. Domain = reldom(LDoms, M1, M2, RDoms)
- *Uniquify all primitive domains* in LDoms and RDoms, that is remove any duplicate values. This is necessary because the domains may depend on run time constants, unknown at compile time, but instantiated by now. *Domains are sets and must not contain duplicates.*
- Call `get_type` (section 4.2.5) to determine the representation type.
- Check the dimensions of Matrix against all domains. The length of each partial list of Matrix must match the length of the corresponding domain. Fail if there is an unexpected error, i.e. list expected, but found something else.
- Check the primitive elements of Matrix. They must be integers.
- Apply the clpfd:domain/3 predicate to all values to make sure they are in the correct domain. The domain is 0..1 for the binary representation and the range of the total function in the optimized case.
- Apply Cardinality (section 4.2.4) to Matrix to ensure that no cardinality constraint is violated.
- If we come so far unify Relation with `relation(Domain, lm_rel(Matrix))`.

4.3 The Translation Algorithm, some Preliminaries

The rest of this major section will focus on the mapping of ESRA syntax into SICStus Prolog. This sub section will treat some preliminary concepts, necessary to understand how the compiler works.

4.3.1 Traversing a Parse Tree

I assume that the typical reader has a fair knowledge of translation algorithms, but I will devote this section to a brief discussion of parse tree traversal for the benefit of the casual reader. The parser generates a *parse tree*, which is an abstract representation of the program syntax. The higher up in the tree, the bigger syntactic entities will be found. The root represents the entire model, the intermediate nodes expressions and the leaves primitive values.

The translation algorithm performs a complete traversal of the parse tree, visiting all nodes. At each node a recursive call is made for each direct child of that node, returning some code chunk or other item. The different items are combined in some way and passed back to the caller. *The parse tree is processed bottom up.*

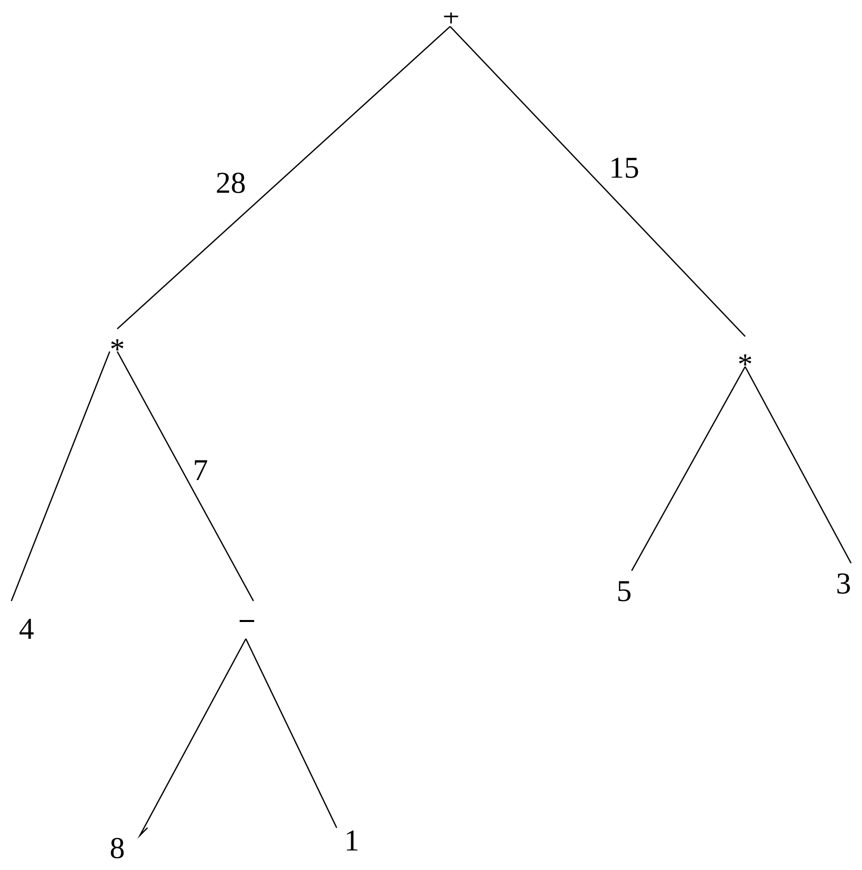

Figure 2: Parse Tree of $4 * (8 - 1) + 5 * 3$

As an example consider the numeric expression $4{*}(8{-}1){+}5{*}3$. its parse tree is depicted in Figure 2. Translation is the same as evaluation in this example. The value returned by each computation node is labeled on the branch leading to it. So at the "-" node 8-1 = 7 is computed and passed back to its parent. Then 4*7 = 28 is computed at the left "*" node and passed back to the root. On the right side of the tree 5*3 = 15 is computed at the right "*" node and passed back. Finally, at the root, the result 28+15 = 43 is computed. All translation algorithms work in a similar way.

4.3.2 The Conjunctive Context

Consider the formula $F_1 \wedge F_2 \wedge F_3$. If this is true then F_1, F_2 and F_3 are also true individually (by elimination of $\wedge$). In this case it's enough to state each of the Fs individually as a constraint. Assume on the contrary that $F \Rightarrow (A_1 \wedge A_2)$ is true. Then it is not generally true that A_1 is true. The reason is that the conjunction $A_1 \wedge A_2$ is only *conditionally true*, i.e. it's true if F is true. In this case we cannot state A_1 as a constraint by its own but must compute the conjuction $A_1 \wedge A_2$ and *reify it*. The correct way stating that formula is: $T \Leftrightarrow A_1 \wedge A_2, F \Rightarrow T$, using an intermediate boolean variable T. In the first case I will say that the conjunction is evaluated in a *conjunctive context*, while in the second case the implication operator *cuts off* the conjunctive context, making the context *disjunctive.* In a conjunctive context it's permissible to *replace a conjunction with each conjunct separately.* In a disjunctive context, every conjunctive formula *has to be reified* and stored in a new variable.

The conjunctive context is implemented as an optimization. Without it, any conjunction would *always have to be reified.* The compiler passes the *conjunct flag* to the translation function which is true in a conjunctive context and otherwise false. As soon as a disjunctive node is reached in the parse tree, the conjunct flag is set to false and the conjunctive context is cut off for all descendants of this node.

Consider the ESRA expression `forall(i : 1..10) (F(i) = 5)`. This is just the iterated conjunction `F(1) = 5 /\ ... /\`$F(10) = 5$. In a conjunctive context each of these equalities can be stated as is. In a disjunctive context, on the contrary, each conjunct in the iteration must be stored away in a list. When the iteration is complete this list is passed back to the ancestor node in the parse tree and treated there.

Note that the count operator, along with all non conjunctive connectives ($\vee, \Rightarrow, \Leftarrow$ *and* $\Leftrightarrow$), cuts off the conjunctive context; meaning that all expressions nested inside a count operator lives in a disjunctive context.

4.3.3 Communicating Variables between Sub Predicates

In order to map ESRA quantified expressions on SICStus Prolog, it's necessary to create sub predicates (see section 4.5). This gives rise to a new problem, because *Prolog doesn't have any global variables.* So how to create communication between the same variables in different sub predicates? One solution is to state all variables as facts, but I actually adopted another solution: the *vlist mechanism.*

The compiler maintains a variable `vlist`, which is the list of names of all user supplied variables used in the model. before any sub predicates are created this list must be unified with the name `Vlist`. The variable `Vlist` is then passed as an argument to all sub predicates. Inside the sub predicate the `Vlist` argument must again be unified with `vlist`. This mechanism is best illustrated by an example. In the following program fragment a sub predicate `sub` is being called.

```
...
Vlist = [A, B, Potatoes, N1],
sub(, , ,Vlist),
....

sub(, , ,Vlist) :-
          Vlist = [A, B, Potatoes, N1],
          ...
```

The unification of Vlist with [A, B, Potatoes, N1] makes sure that the variables A, B, Potatoes and N1 refer to the same objects in the caller as in sub. A similar mechanism is used for local variables. The local variables, active in the caller's scope, are maintained in a list called `Locals` and passed to all sub goals within this scope.

Finally all *domain expressions*, which may depend on compiler generated variables, are maintained in a list called Domains. When this list is unified with the corresponding argument in a sub predicate, all temporaries within the domain expression become unified as well. An example will clarify this mechanism (here the focus is on Domains, so Vlist and Locals are omitted).

```
...
Domains = [interval(1,T1), list([1, T2, 3, T3, 5])],
sub(, , ,Domains),
....

sub(, , ,Domains) :-
       Domains = [interval(1,T1), list([1, T2, 3, T3, 5])],
       ...
```

The unification of Domains with the list expression above makes sure that T1, T2 and T3 become defined within sub. I will often refer to the unification of variable and list as a *packing instruction* on the caller's side and as an *unpacking instruction* in the callee.

4.4 Translation of Simple Expressions

Simple expressions are such ones not containing quantifiers or other looping constructs. This include numeric and propositional expressions, applications as well as declarations. The purpose of this sections is to define the ESRA to prolog mapping of these expressions.

4.4.1 Compile time Evaluation of Constant Expressions

While translating expressions, the compiler should evaluate constant expressions like 4*5=20. This works very much like the example in section 4.3.1. Among other benefits this will enable the compiler to evaluate all expressions depending on declared and initialized constants. An example will clarify this procedure.

```
cst G = 8
dom D = {7*G, -1, G, 5}
(Comment: the compiler evaluates D = {56, -1, 8, 5})
cst B = 10+G
(Comment: the compiler evaluates B = 18)
etc
```

It's clear that if there are no run time initialized constants, the compiler will be able to evaluate every ground expression at compile time. This works because I require that *each identifier must be declared before it's used.*

4.4.2 Application Expressions

The predicate 'value/3', introduced in section 4.2, is used to apply a relation on a tuple of values. Here, only the declarative aspects of value/3 are of interest, the actual implementation details are discussed in section 4.2. The meaning of value/3 is:

let R be an n-ary relation. If $value([a_1, a_2, .., , a_n], R, 1)$, then $R(a_1, a_2, ..., a_n)$ holds. Similarly if $value([a_1, a_2, .., , a_n], R, 0)$, then $R(a_1, a_2, ..., a_n)$ is false.

Let F be a total function with left arity $n-1$ and right arity 1. If it's true that $value([a_1, a_2, ..., a_{n-1}], F, V)$, then $F(a_1, a_2, ..., a_{n-1}) = V$ is also true.

The predicate 'new/2', which is used to create relational decision variables, is also important. The predicate `new(+Domain, -Relation)` is true if `Relation` is a relation with domain `Domain`. Here `Domain` must be a fully instantiated relational domain expression while `Relation` must be an un instantiated variable. `Relation` becomes unified with a new fresh relation.

The subsequent sections will show many examples of how to use these two predicates. For instance each relation application expression maps into a call to value/3.

4.4.3 Numeric and Comparison Expressions

The translation of numeric and comparative expressions into Prolog follows the general outline of section 4.3.1. At each node in the parse tree, a couple of code chunks are assembled by recursive calls to the translation function and concatenated together with the new code generated at that node and finally passed back to the caller.

In the following I will focus attention on the *mapping* between ESRA and Prolog, so from now on the discussion will be declarative omitting the details of the translation process.

The compiler will generate many so called *temporary variables* to hold intermediary results which have no names in the source model. I will adopt the convention here to call these variables T1, T2, T3, and so fourth. The mapping is straight forward, just replace each ESRA operator with its Prolog counter part. In the examples I will sometimes use the (Assume ...) construction which means that the text within the parentheses does not participate in the mapping but carries essential information, such as declarations. The actual ESRA text which is mapped will be enclosed in angular brackets. It's assumed that each expression to be mapped lives in a conjunctive context, so the final constraint will not be reified. If the result is just a numeric value it will be left in a temporary. Embedded application expressions will be treated as in section 4.4.2. Note that the CLPFD operators #= etc are used, because it's decision variables which are dealt with here. The sign `-->` means "maps to". If the reader have problems to understand why the instructions are generated in a particular order he is encouraged to draw a parse tree for himself and do the translation by hand.

```
< X*24 = I*F(I-1) > -->
T1 #= X*24,
T2 #= I-1,
value([T2], F, T3),
T4 #= I*T3,
T1 #= T4.

< F(3*F(I*F(I))) > -->
value([I], F, T1),
T2 #= I*T1,
value([T2], F, T3),
T4 #= 3*T3,
value([T4], F, T5).
(result in T5)

< X*Y-1 != 0 > -->
T1 #= X*Y,
T2 #= T1-1,
T2 #\= 0.

< X*(F(X) - 1) =< G(X-1, Y) > -->
value([X], F, T1),
T2 #= X*T1,
T3 #= T2-1,
T4 #= X-1,
value([T4, Y], G, T5),
T3 #=< T5.
```

4.4.4 Propositional Expressions

Propositional expressions are almost as straight forward to map, but are slightly complicated by the conjunctive context.

```
< F => X=5 > -->
X #= 5 #<=> T1,
F #=> T1.
```

The sub expression X=5 is evaluated in disjunctive context because of the implication operator, so the result is reified into T1. Then the implication is translated in conjunctive context (which is assumed).

```
< false => R(I,J) >
value([I,J], R, T1),
0 #=> T1

< BIBD(v1,j) /\ BIBD(v2,j) > -->
value([v1, j], BIBD, 1),
value([v2, j], BIBD, 1),
```

4.4.5 Declarations

Declaration causes two kinds of effects. Firstly a symbol table entry is created for every declared identifier. The translator uses this information for type checking purposes. Secondly *code may be generated by the declaration* for various reasons.

When a primitive decision variable is declared its domain must be made known to CLPFD. This produces a line of code like `X in 1..10`. Constants may be initialized in line by an expression that may contain run time constants.

Such expressions has to be evaluated and temporaries must be created to hold their values. Sometimes a domain declaration also contain expressions which must be evaluated resulting in new temporaries.

When a relational decision variable is declared, a call to `new/2` must be generated (section 4.4.2) to create the relation. Constant declarations will not be considered here, see section 4.7.1 instead. Recall that (Assume ...) in the mapping syntax means that the thing within parentheses is not part of the ESRA expression being mapped, but is needed as contextual information.

```
< var X : 0..10 > -->
X in 0..10

< var X : {1, 7, 4} > -->
X in {1, 7, 4}

(Assume G, R are integer run time constants)
< var X : 1..G*R > -->
T1 #= G*R,
X in 1..T1

(Assume A, B, C and E are integer run time constants)
< dom D = {2, 3-A, B, 4*C, 55, 9-E} > -->
T1 #= 3-A,
T2 #= 4*C,
T3 #= 9-E.
(Comment: The domain D becomes known by the compiler,
no additional code produced for this purpose! The variables
T1, T2 and T3 contains the results of computations of 3-A,
4*C and 9-E. All references to D in the following are literally
substituted by the term 'list([2, T1, B, T2, 55, T3])' by the
compiler. Therefore the T variables need to be remembered. The
D identifier binding is not stored in the generated code. This is
a design decision.)

< var F : (1..10 [#] 1..5) [->] nat > -->
new(reldom([interval(1,10), interval(1,5)],
           1..1,
           0..sup,
           [interval(0, sup)]),
    F).
(Comment: A total function is created in the
declared domain.)

< var S : 1..10[5] > -->
new(reldom([interval(1,10)],
           {5},
           nil,
           []),
    S).
```

In the last example a *set domain* expression is constructed, see section 4.1.3.

4.5 Translation of Quantified Expressions

4.5.1 Loops in Prolog

The quantified expressions are not so strait forward to map into Prolog as the stuff we have treated so far. The reason is that it's necessary to loop over the domains of each local variable. The quantifiers and sums are really the ESRA counter part of loops in ordinary programming languages. But Prolog doesn't support loops, only *recursion*, so I have to create several recursive predicates, in fact *one for each local variable.*

Before treating this problem in detail, I want to give examples of some simple Prolog looping techniques. The first example is a double loop over two intervals, the Prolog equivalent to a double for loop. The loop body simply prints the values of the local variables at each turn.

```
%% A double for loop in Prolog.
%% Call with loop1(Ibegin, Jbegin, Max)
%% Printing all pairs i,j with Ibegin =< I =< Max and
%% Jbegin =< j =< Max
loop1(I, J, Max) :-
      ( I =< Max ->
          Inew is I+1,
          loop2(I, J, Max),
          loop1(Inew, J, Max)
      ;
          true
      ).
loop2(I, J, Max) :-
      ( J =< Max ->
          Jnew is J+1,
          format(``~d, ~d~n'', [I,J]),
          loop2(I, Jnew, Max)
      ;
          true
      ).
```

This program uses iteration over the natural numbers. It's also possible to iterate over lists in a similar way. Suppose D1 and D2 are two lists. The following example loops over the entire "cartesian product" of D1 and D2 and prints all pairs.

```
%% D1 and D2 are integer lists
%% Print the ``cartesian product'' of D1 and D2
loop1([], _).
loop1([A1 | A1s], D2) :-
    loop2(A1, D2),
    loop1(A1s, D2).
loop2(_, []).
```

```
loop2(A1, [A2 | A2s]) :-
    format("~d, ~d~n", [A1, A2]),
    loop2(A1, A2s).
```

Primitive domains are either intervals or lists. Because it's slightly more convenient to iterate over lists I decided to *always use lists* when iterating over primitive domains. Therefore, I convert every interval which appears in a local variable specifier to a list, before starting the iteration. The predicate `domain_to_list/2` makes that trick (see code in esra.pl). The signature is

```
%% domain_to_list(+Domain, -List) :-
%% Domain is a domain expression, which must be ground and
%% finite.
%% List is a list, which must be uninstantiated at entry.
%% True if List contains the elements of Domain.
```

4.5.2 Iteration Schemas

The sub predicates, generated by the compiler, follow fixed patterns. I call these patterns *iteration schemas*. Note that there is exactly one predicate for each local variable and each predicate has two clauses, one *base clause* and one *recursive clause*. In this sub section I will use abstract pseudo syntax names, such as as $loop_1, loop_2, loop_{i-1}, loop_n$, for all sub predicates.

There are two types of iteration schemas, the *blank schema* and the *schema with collection*. In the blank schema, the main body of the loop just posts a constraint, no result is computed. The blank schema is used in conjunctive context. In the schema with collection, on the other hand, no constraint is posted directly, instead the constraint is *reified* and added to an accumulation list at each turn of the loop. The accumulated list is then passed back to the caller. This schema *collects* the result of a computation. Both schemas exhibit a strong similarity to the looping examples of section 4.5.1.

Suppose the local variables are $A_1, A_2, \ldots, A_n$, belonging to the primitive domains $D_1, D_2, \ldots, D_n$, where one or more of the D_i may be identical. There will be one predicate, named $loop_i$, associated with each local variable A_i. There will also be an initial call to $loop_1$ starting the n fold iteration. Only $loop_n$ will compute anything. The whole purpose of $loop_1$, $loop_2$, …, and $loop_{n-1}$, is to start the next sub iteration. In the schema definitions which follow, I will replace the arguments Vlist, Locals and Domains of section 4.3.3 with an ellipsis, in the interest of clarity. Suppose also that the domains D_i already are *lists*.

Consider the blank schema first. Both base clause and recursive clause are given for each predicate. In the base case, the empty list, is supposed to match the recursive argument in the other clause.

Definition: Blank Schema

Initial goal: $loop_1(D_1, D_2, \ldots, D_n, \ldots)$

$loop_1$(`[]`, _, ...,, ...).

```
$loop_1([A_1|A_1s], A_2, ..., A_n, ...) :-$
    $loop_2(A_1, D_2, ..., D_n, ...),$
    $loop_1(A_1s, D_2, ..., D_n, ...).$

$loop_2(\_, [], \_, ..., ...).$
$loop_2(\_, [A_2|A_2s], A_3, ..., A_n, ...) :-$
    $loop_3(A_1, A_2, D_3, ..., D_n, ...),$
    $loop_2(A_1, A_2s, D_3, ..., D_n, ...).$

⋮

$loop_n(\_, \_, ..., [], ...).$
$loop_n(\_, [A_n|A_ns], ...) :-$
    $(Do\ something\ here),$
    $loop_3(A_1, A_2, ..., A_{n-1}, A_ns, ...).$
```

The schema with collection is slightly more complicated. There are three compiler generated variables, always named X, Y and Z, used as accumulators to scope up the computed result. X,Y is the primary accumulator pair, but Z is used for intermediate lists.

Definition: Schema With Collection

Initial goal: $loop_1(D_1, D_2, \ldots, D_n,\ [], Res, \ldots)$
(Comment: At exit, *Res* is the list of computed results, one result per turn)

```
$loop_1([], \_, ...,\ X, X,\ ...).$
$loop_1([A_1|A_1s], D_2, ..., D_n,\ X, Y,\ ...) :-$
    $loop_2(A_1, D_2, ..., D_n,\ X, Z,\ ...),$
    $loop_1(A_1s, D_2, ..., D_n,\ Z, Y,\ ...).$

$loop_2(\_, [], \_, ...,\ X, X,\ ...).$
$loop_2(A_1, [A_2|A_2s], D_3, ..., D_n,\ X, Y,\ ...) :-$
    $loop_3(A_1, A_2, D_3, ..., D_n,\ X, Z,\ ...),$
    $loop_2(A_1, A_2s, D_3, ..., D_n,\ Z, Y,\ ...).$

⋮

$loop_n(\_, \_, ..., [],\ X, X,\ ...).$
$loop_n(\_, [A_n|A_ns],\ X, Y,\ ...) :-$
    $(Compute\ something\ here => T),$
    $loop_n(A_1, A_2, ..., A_{n-1}, A_ns,\ [T|X],\ ...).$
```

4.5.3 The Forall Quantifier

Consider the expression

```
forall (LocalVariables | Filter) (Formula)
```

The first to do is to rewrite this expression to

```
forall (LocalVariables) (Filter => Formula),
```

taking care of the filter clause automatically. This would be overkill if `Filter` is ground because a simple Prolog conditional would suffice. But in the interest of simplicity I always make this transformation in my compiler.

The next step is to generate an iteration schema for `LocalVariables` and fill it with the translation of `Formula`. If the schema is collecting, the result is a list of boolean values, which have to be *reified.* Suppose that `List` contains the results of the computation. To reify `List`, use the CLPFD sum constraint.

```
length(List, N),
clpfd:sum(List, #=, Sum),
Sum #= N.
```

This makes the entire forall expression true, because this really is equivalent to putting all elements of List equal to 1. If the expression is computed in a disjunctive context, however, the last line above have to be replaced by `Sum #= N #<=> Result`, to reify the expression instead.

4.5.4 The Count Quantifier

Consider the expression

```
count (Multiplicity) (LocalVariables | Formula)
```

The first thing to do is to generate a schema for `LocalVariables`. For the count quantifier, the schema must *always be collecting*, because we cannot handle the counting constraint before we have seen all values to be counted. Therefore the conjunctive context has to be cut off before the Formula is translated. Suppose that `List` contains the results of the computation. To enforce the counting constraint on `List`, use the CLPFD count constraint.

```
clpfd:count(1, List, #=, Count),
Count in Multiplicity
```

Just counting the number of true values in List and making sure it's in Multiplicity. This can be reified if the last line above is replace by

```
Count in Multiplicity #<=> Result
```

4.5.5 The Sum Operator

Consider the expression

```
sum (LocalVariables | Filter) (Expression)
```

The first to do is to rewrite this expression to

```
sum (LocalVariables) (Filter*Formula),
```

Here Filter *must* be computed in a *disjunctive context*, enforcing reification. Then Filter functions as a binary weight factor, to decide which terms in the sum to omit and which ones to include, in the final result. Then a schema is created for `LocalVariables`. The schema for the sum operator is *always collecting*, because all values have to be computed before the sum can be computed. Assume that List contains the results of the computation. To compute the sum of List, use the CLPFD sum constraint.

```
clpfd:sum(List, #=, Result)
```

The resulting sum value is now stored in `Result`.

4.5.6 Examples

The section is concluded with several examples of ESRA to Prolog mappings. The *vlist mechanism* is assumed, but I'm not interested here in the actual values of the lists, so I replace them by ellipses and ignores how the variables in the sub predicates get their values. See section 4.8 for an example of a complete vlist mechanism. Also note that the domains are declared as temporary variables beginning with 'T' in the sub predicates. This is always done. From now on, the convention to name sub predicates after the same base name as the model itself is adopted. For instance if the model's name is 'model', the sub predicates are named model1, model2 etc.

```
(Assume dom D = 1..3)
(Assume model name is 'test')
< forall(I&J : D) (I*J < 100) > -->
...
(Comment: The blank schema is used)
test1([1,2,3], [1,2,3], Vlist, Locals, Domains),
...
test1([], _, _, _, _).
test1([I|Is], T2, Vlist, Locals, Domains):-
     test2(I, T2, Vlist, Locals, Domains),
     test1(Is, T2, Vlist, Locals, Domains).
test2(_, [], _, _, _).
test2(I, [J|Js], Vlist, Locals, Domains):-
     Vlist = ...,
     Locals = ...,
     Domains = ...,
          T3 #= I*J,
          T3 #< 100,
          test2(I, Js, Vlist, Locals, Domains).


     (Assume model name is 'test')
     < count(20 (I : 1..3 | F(I) = I) > -->
     ...
     (Comment: Collecting Schema Used)
     test1(T1, [],Results, Vlist, Locals, Domains),
     clpfd:count(1, Results, #=, Count),
     Count in {2},
     ...
     test1([], X,X, _, _, _).
     test1([I|Is], X,Y, Vlist, Locals, Domains):-
          Vlist = ...,
          Locals = ...,
          Domains = ...,
          value([I], F, T2),
          T2 #= I #<=> T3,
          test1(Is, [T3|X], Y, Vlist, Locals, Domains).


     (Assume model name is 'summing')
     < S = sum(I : 1..10) (I*I - 1) > -->
     ...
     (Comment: Collecting Schema Used)
     summing1([1,2,3,4,5,6,7,8,9,10], [],Xs, Vlist, Locals, Domains),
     clpfd:sum(Xs, #=, Sum)
     ...
     summing1([], X,X, _, _, _).
     summing1([_I|_Is], X,Y, Vlist, Locals, Domains):-
         Vlist = ...,
         Locals = ...
         Domains = ...,
         T1 #= I*I,
         T2 #= T1-1,
         summing1(Is, [T2|X],Y, Vlist, Locals, Domains).
```

4.6 Translation of the Objective

When all constraints have been posted the problem have to be submitted to the solver for a solution. This is the *objective* of the model. There are two kinds of objectives, `solve` and `optimise` respectively. Consider the objective `solve <formula>`. After `<formula>` has been translated, a call to the library predicate *all_variables/2*, defined in esra.pl is generated. The signature is

```
%% all_variables(+ESRAVariables, -CLPFDVariables)
%% ESRAVariables is a list of all ESRA decision variables
%% True if CLPFDVariables is the list of all CLPFD decision variables,
%% which has been allocated.
```

The lines of code that invokes the solver to solve a CSP, are always (if we assume that V1, V2, ..., Vn are all decision variables in the ESRA model).

```
esra:all_variables([V1, V2, ..., Vn], AllVariables),
clpfd:labeling([ffc], AllVariables)
```

The ellipsis shall be figuratively interpreted here, in an actual generated program n is always known, so the compiler generates the list of known names.

If we are dealing with a COP instead, and the objective function is bound to the CLPFD variable T, then the corresponding Prolog lines are

```
esra:all_variables([V1, V2, ..., Vn], AllVariables),
clpfd:labeling([ffc, Optimize], AllVariables)
```

Here Optimize is one of the atoms maximize or minimize. Note the option `[ffc]` sent to the labeling/2 predicate. This means "first failed with the most constrained heuristic". This leftmost variable with the smallest domain is always chosen first breaking ties by (a) selecting the variable that has the most constraints suspended on it and (b) selecting the leftmost one. This default can be overridden by changing the compiler variable 'options' defined in generate.ml.

4.7 Input and Output

4.7.1 The Data File, and Run Time Constants

Run time constants have to be initialized by the data file. The data file actually is a Prolog program, so each data constructor is a valid Prolog term (section 2.7). if the compiler encounters run time constants it generates a call to the predicate `read_file/2`, defined in esra.pl, which reads the file and stores each data constructor in a dictionary named Unknowns. The signature is

```
%% read_file(+FileName, -Unknowns)
%% FileName is an atom, the name of the data file.
%% Unknown must be un instantiated when this predicate
%% is called.
%% Reads all run time constants from the file FileName and
%% stores them in the dictionary Unknowns.
```

Later, each run time constant is retrieved from Unknowns by a call to `find_variable/4`, which is also defined in esra.pl. This predicate has the signature

```
%% find_variable(+Key, +Unknowns, -Value, +Domain)
%% True if Key is bound to Value in Unknowns.
%% Key is an atom, the ESRA name of the constant.
%% Unknowns is the dictionary containing all run time
%% constants.
%% Domain is the domain of the constant.
%% Value is the name of the constant.
```

When `find_variable` encounters a relational data constructor it calls `parse_relation/3` of section 4.2.6 to construct the object. Now I give some examples of the mapping of ESRA constant declarations to Prolog.

```
< cst K : nat > -->
find_variable('K', Unknowns, K, interval(0, sup))

< cst Array : 1..10 [->] nat > -->
find_variable('Array', Unknowns, Array,
              reldom([interval(1,10)],
                     {1},
                     0..sup,
                     [interval(0,sup)])
             )

< cst Set : 1..10 [{2,5}] > -->
find_variable('Set', Unknowns, Set,
              reldom([interval(1,10)],
                     {2,5},
                     nil,
                     [])
             )
(Comment: This is a set constant)
```

4.7.2 Printing and Results

After a solution to the problem has been found, the predicate `make_results/2` (defined in esra.pl) is called to create a dictionary containing all decision variables of the problem. This dictionary is the *result parameter*. The signature of `make_results/2` is

```
%% make_results(+List, -Result)
%% List is a D-list of pairs Key-Value, one for each decision
%% variable of the problem.
%% Result must be un instantiated by entry.
%% True if Result is a dictionary of all (Key,Value) pairs.
%% The variable Result holds the result parameter on exit.
```

The next step is to call `print_results/2` (defined in esra.pl), to print all resulting decision variables on the screen according to a default printing format. The signature is

```
%% print_result(+Result)
%% Result is a D-list of pairs Key-Value, one for each
%% decision variable.
%% Prints the results of the problem in a default format
%% on the screen.
```

The result parameter contains all decision variables of the problem. The user can extract this information by using the *result API*, which is described in section 5.4.

4.8 The Branch Tree

Before presenting the final example of a real world translation, there's a necessary subtlety to consider. When translating a quantified expression a new predicate is created. The program text of this predicate is to inserted later in the code stream. If we are writing instruction to a file one by one, we are getting into problems. The solution is that the compiler maintains a data representation of the program which I call a *branch tree* in memory during the translation. Only when the translation is complete, is the branch tree converted into the final program. To explain the branch tree, consider the following ESRA code chunk

```
N = sum(I : 1..5) (I*I) /\
forall(I:1..5) (I < 10).
```

A branch tree for this code chunk is

```
(Assume: program name is ‘‘test’’)
(Comment: Vlist mechanism takes care of N)
(Comment: Domains and Locals uninteresting now, replace by ellipsis.)
Vlist = [N],
test1([1,2,3,4,5], [],T1, ...),
Branch(
         test1([], X,X, _, _, _).
         test1([I|Is], X,Y, Vlist, Locals, Domains) :-
               Vlist = [N],
               Locals = ...,
               Domains = ...,
               T3 #= I*I,
               test1(Is, [T3|X],Y, Vlist, Locals, Domains).
      ),
clpfd:sum(T1, #=, T2),
N #= T2,
test2([1,2,3,4,5], ...),
Branch(
         test2([], _, _, _).
         test2([I|Is], Vlist, Locals, Domains) :-
               Vlist = [N],
               Locals = ...,
               Domains = ...,
               I #< 10,
               test2(Is, Vlist, Locals, Domains)
```

```
        ).
```

When the entire model has been translated, the compiler converts the branch tree into a final Prolog program. First all instructions not occurring in any branch are written out, the so called *stem* of the branch tree. Then all branches are written out recursively. The result is a Prolog program with several predicates.

4.9 A Complete Example

Now it's time to present a major example of a full translation of an ESRA model. I have chosen the well known *warehouse location problem*, see section 3.6 for a definition. This model differs from the one of section 3.6 because I think it's better suited to illustrate the subtleties of translation than the other model, but it models the same CSP.

4.9.1 Preliminaries

Before the example can be understood, there are some preliminary notions that must be explained, which have not been encountered yet.

The name mapping. The compiler generates variables with names like Vlist, Domains, AllVariables, X, Y and Z. How can we assure that these variables don't clash with user supplied ones? There's also the problem that Prolog doesn't allow variable names to begin with lower case letters, while ESRA does so. The solution is to *map the names* to an internal format. The mapping is very simple; just prepend an underscore to the name. Thus, the ESRA name `foo` maps to the prolog name `_foo`, while `Bar` maps to `_Bar`.

The computer generated "temporary variables" have so far been called T1, T2 etc. We now rename these variables to `_1, _2` etc, to avoid clashing with user variables beginning with T.

Module prefixes. The familiar predicates `value/3`, `new/2` etc, have acquired a *module prefix*, to show that they belong to one of the ESRA *library* modules: esra.pl and lmatrix.pl. These modules contain all necessary Prolog code to support the generated program at run time.

Once wrappers. Several predicates are wrapped into calls of `once/1`. This is to *suppress backtracking*. A goal called by `once(Goal)` is only invoked once. When backtracking occurs, Goal doesn't leave any choice points on the stack. Without this precaution, the program often backtracks when it's not supposed to, causing various problems.

Files. There must be several files to run the model. Firstly, the generated program itself, named `model.pl`, if `model` is the base name of the model. Secondly, the data file, which has the name `model.in.pl`. Both these files are Prolog programs. Thirdly, the library files `esra.pl` and `lmatrix.pl`, must be in the directory indicated by the module declarations at the head of the generated program.

4.9.2 Stepping Through the Example

The example model is named `ware`, the *warehouse location problem.* The source file is `ware.pl` and the data file `ware.in.pl`. The following ESRA source model has been line numbered to facilitate referencing.

```
  1   cst NStores : nat
  2   cst NWareHouses : nat
  3   dom Stores = 1..NStores
  4   dom Warehouses = 1..NWareHouses
  5   cst Capacity : Warehouses [->] nat
  6   cst SupplyCost : (Stores [#] Warehouses) [->] nat
  7   cst MaintenanceCost : nat
  8   var Supplies : Warehouses [#1] Stores

  9   minimise
  10         MaintenanceCost *
  11        (sum(W : Warehouses | count(1..sup) (
  12             S : Stores | Supplies(W,S))) (1)
  13        )
  14        +
  15        sum(W : Warehouses, S : Stores | Supplies(W,S)) (
  16             SupplyCost(S,W)
  17        )

  18  such that
  19       (forall (W : Warehouses)
  20                    (count (0..Capacity(W))
  21                            (S : Stores | Supplies(W,S))))
```

And here is the translation, ware.pl, also line numbered.

```
1   :- use_module(library(clpfd)).
2   :- use_module(library(lists)).
3   :- use_module('lmatrix.pl').
4   :- use_module('esra.pl').

5   ware(Results):-
6           esra:reset_counters,
7           esra:read_file('ware.in.pl', Unknowns),
8           esra:find_variable('NStores', Unknowns, _NStores,
                                  interval(0, sup)),
9           esra:find_variable('MaintenanceCost', Unknowns,
10                                 MaintenanceCost, interval(0, sup)),
11          esra:find_variable('NWareHouses', Unknowns, _NWareHouses,
                                  interval(0, sup)),
12         lmatrix:new(reldom([interval(1, _NWareHouses)],
                                0..sup, 1..1,
                                [interval(1, _NStores)]),
                         _Supplies),
      esra:find_variable('SupplyCost', Unknowns, _SupplyCost,
                      reldom([interval(1, _NStores),
                             interval(1, _NWareHouses)],
                             {1}, 0..sup,
                             [interval(0, sup)])),
      esra:find_variable('Capacity', Unknowns, _Capacity,
                      reldom([interval(1, _NWareHouses)],
                             {1}, 0..sup,
                             [interval(0, sup)])),
      Vlist = [_NStores, _MaintenanceCost, _NWareHouses, _Supplies,
               _SupplyCost, _Capacity],
      Domains = [interval(1, _NWareHouses), interval(1, _NStores)],
      esra:domain_to_list(interval(1, _NWareHouses), _8),
      once(ware3(_8, [],_14, Vlist, [], Domains)),
      sum(_14, #=, _15),
      _16 #= _MaintenanceCost*_15,
      esra:domain_to_list(interval(1, _NWareHouses), _1),
      esra:domain_to_list(interval(1, _NStores), _2),
      once(ware1(_1, _2, [],_6, Vlist, [], Domains)),
      sum(_6, #=, _7),
      _17 #= _16+_7,
      esra:domain_to_list(interval(1, _NWareHouses), _18),
      once(ware5(_18, Vlist, [], Domains)),
      esra:all_variables([_Supplies], AllVariables),
      esra:report_postings,
      labeling([down, minimize(_17)], AllVariables),
      esra:report_solution(optimum-_17),
      esra:make_results(['Supplies'-_Supplies], Results),
      esra:print_results(['Supplies'-_Supplies]).

ware5([], _, _, _).
ware5([_W|_Ws], Vlist, Locals, Domains):-
      [_NStores, _MaintenanceCost, _NWareHouses, _Supplies, _SupplyCost,
      _Capacity] = Vlist,
      [] = Locals,
      [interval(1, _NWareHouses), interval(1, _NStores)] = Domains,
      lmatrix:value([_W], _Capacity, _19),
      esra:domain_to_list(interval(1, _NStores), _20),
      once(ware6(_20, [],_22, Vlist, [_W], Domains)),
      esra:set_counts(_22, 0.._19),
      once(ware5(_Ws, Vlist, Locals, Domains)).


ware6([], X,X, _, _, _).
ware6([_S|_Ss], X,Y, Vlist, Locals, Domains):-
      [_NStores, _MaintenanceCost, _NWareHouses, _Supplies, _SupplyCost,
       _Capacity] = Vlist,
      [_W] = Locals,
      [interval(1, _NWareHouses), interval(1, _NStores)] = Domains,
      lmatrix:value([_W, _S], _Supplies, _21),
      once(ware6(_Ss, [_21|X], Y, Vlist, Locals, Domains)).


ware1([], _, X,X, _, _, _).
ware1([_W|_Ws], _2, X,Y, Vlist, Locals, Domains):-
      once(ware2(_W, _2, X,Z, Vlist, Locals, Domains)),
      once(ware1(_Ws, _2, Z,Y, Vlist, Locals, Domains)).

ware2(_, [], X,X, _, _, _).
ware2(_W, [_S|_Ss], X,Y, Vlist, Locals, Domains):-
      [_NStores, _MaintenanceCost, _NWareHouses, _Supplies, _SupplyCost,
       _Capacity] = Vlist,
      [] = Locals,
      [interval(1, _NWareHouses), interval(1, _NStores)] = Domains,
      lmatrix:value([_S, _W], _SupplyCost, _4),
      lmatrix:value([_W, _S], _Supplies, _3),
      _5 #= _4*_3,
      once(ware2(_W, _Ss, [_5|X], Y, Vlist, Locals, Domains)).


ware3([], X,X, _, _, _).
ware3([_W|_Ws], X,Y, Vlist, Locals, Domains):-
      [_NStores, _MaintenanceCost, _NWareHouses, _Supplies, _SupplyCost,
       _Capacity] = Vlist,
      [] = Locals,
      [interval(1, _NWareHouses), interval(1, _NStores)] = Domains,
      esra:domain_to_list(interval(1, _NStores), _9),
      once(ware4(_9, [],_11, Vlist, [_W], Domains)),
      esra:set_counts_and_reify(_11, 1..sup, _12),
      _13 #= 1*_12,
      once(ware3(_Ws, [_13|X], Y, Vlist, Locals, Domains)).


ware4([], X,X, _, _, _).
ware4([_S|_Ss], X,Y, Vlist, Locals, Domains):-
      [_NStores, _MaintenanceCost, _NWareHouses, _Supplies, _SupplyCost,
       _Capacity] = Vlist,
      [_W] = Locals,
      [interval(1, _NWareHouses), interval(1, _NStores)] = Domains,
     lmatrix:value([_W, _S], _Supplies, _10),
     once(ware4(_Ss, [_10|X], Y, Vlist, Locals, Domains)).
```

The construction in lines 11–12 of the source model requires an explanation. Part of the explanation is already given in section 2.1.2. The sum expression evaluates to the cardinality of the set

```
{W : Warehouses |
                count(1..sup)
                    (S : Stores | Supplies(W,S))}.
```

This cardinality is the number of open warehouses, so we get the total fixed cost by multiplying this number with the fixed cost per warehouse (`MaintenanceCost`). Now I'm stepping through the example line by line to explain how it works.

Line 1–4
Module declarations. Open modules clpfd, lists, lmatrix and esra. The first two are system modules, while the latter belong to the ESRA *library.* If the module declarations of the library points to another directory, these files should reside there instead of in the current directory. A variable in the compiler decides to which directory the library files belong.

Line 5
The main goal, which starts the generated program. Results is the *result parameter.*

Line 6
Reset the CLPFD statistics counters to zero.

Line 7
Read the data file `ware.in.pl`. Store each run time constant in a Key-Value dictionary named `Unknowns`.

Line 8–11
Lookup the declared run time constants of the program: `NStores`, `MaintenanceCost` and `NWarehouses`, in the dictionary. These constants are integer valued as seen by the domains in the calls to `find_variable`. These lines takes care of the lines 1–2 and 7 in the source model. Note that the domains declared in the source are passed as arguments to `find_variable` as well as the variable names.

Line 12
Create the variable Supplies (line 8 in the source). Note how `var Supplies : Warehouses [#1] Stores` translates to a call to `lmatrix:new/2`, defining the prolog variable `_Supplies`.

Line 13–14
More calls to `find_variable` taking care of lines 5–6 in the model. Now _Capacity and _SupplyCost are instantiated. These are *relational constants.* Relational constants are looked up after all declarations have been processed by the compiler, and all domains are known.

Line 15–16
Now when all declarations have been processed by the compiler, the variables Vlist and Domains can be unified with the list of all variable names and the list of all declared domain expressions respectively. This is the *vlist mechanism.*

Line 17
A call to `domain_to_list/2` converts the domain expression `interval(1, _NWareHouses)` into a Prolog list and stores this list in `_8`. This step is a preparation for computing the first sum (source lines 11–13) in the objective function of the model. It's the `Warehouses` domain which has to be converted to a list.

Line 18
The sub predicate ware3 is invoked in the collecting schema to compute the sum `sum(W:Warehouses | count(1..sup) (S : Stores|Supplies(W,S))) (1)`. This sum is actually a weight factor measuring the cardinality of the set of stores a certain warehouse supplies. Only warehouses supplying some store have a maintenance cost. The result is *collected* in `_14`.

Line 19–20
The sum of the list `_14` is computed and multiplied with the maintenance cost. The result is in `_16`. Now we have computed the first term of the objective function, the *fixed cost*.

Line 21–22
More domain-list conversions. The domains `Warehouses` and `Stores` are converted to the lists `_1` and `_2` respectively. This is preparation for the second sum computation.

Line 23
The sub predicate `ware1` computes the second sum of the objective function in the collecting schema. The collect is in `_14`. Note that Locals is `[]`, because no locals are active in the current scope, the main predicate.

Line 24
The collect is summed, using the CLPFD sum constraint. result in `_7`.

Line 25
The two parts of the objective function are summed, result in `_17`.

Line 26–27
Invoking sub predicate `ware5` to compute the forall expression in the such that clause of the model in the blank schema.

Line 28
The `esra:all_variables/2` predicate takes all decision variables of the model and flattens them to get all CLPFD elementary decision variables, which has been allocated. They are collected into the Prolog list `AllVariables`.

Line 29
Code to print out some statistics, like the number of posted constraints etc.

Line 30
Invoking the CLPFD solver to minimize the value in `_17` with respect to all constraints posted so far.

Line 31 Report that a solution has been found along with statistics such that the number of backtracks and the solving time.

Line 32
Call `make_results/2` to create the *result parameter*.

Line 33
Call `print_solution/2` to print the decision variables in a default format.

Now the main branch of the program has been explained. The branch started by `ware3` is treated nest. It's the branch used to compute the first sum in the objective function. Lines 11–12 in the source model.

Line 65–66
This is just the clause heads of the schema with collection over the local variable specifier (W : Warehouses). The collect will be gathered in Y. The predicate `ware3` is the root of the first branch of the program. The predicate is called in line 18.

Line 67–69
The vlist mechanism.

Line 70–71
Calling `ware4` to compute the nested count expression in the schema with collection. The collect is in the list `_11`. Note that Locals is `[_W]`, because the local variable `_W` is now in the outer scope. this call starts a sub branch of `ware3`.

Line 72
The line

```
  esra:set_counts_and_reify(_11, 1..sup, _12)
```

calls a library predicate, taking care of the counting condition of lines 11–12 in the source model. We *reify* the boolean variable `_12` with the statement that the number of occurrences of 1 in `_11` belongs to 1..sup. Effectively we are measuring the cardinality of the set

```
  {S : Stores | Supplies(W,S)}.
```

Line 73
This line computes the expression under the sum sign of source lines 11–12. The factor `_11`, the result of the count, is used as a weight factor controlling if the current term should contribute to the sum or not. The actual term is "1". This construction is a way to simulate the card/1 ESRA operator, which is not supported by this implementation.

Line 74
The recursive call to `ware3`, processing the rest of the domain. The computed value is pushed on the collect X.

Line 75–76
Predicate `ware4` uses the schema with collection to compute the body of the count expression in source lines 11–12. The collect is gathered in `Y`.

Line 77–79
The *vlist mechanism.* Note that the local variable `_W` is not in the current scope, so it must be *inherited* via the vlist mechanism.

Line 80
Using `value` to compute Supplies(W,S).

Line 81
Recursive call to `ware4`, taking care of the rest of the domain. The collect is pushed onto X.

The first sum in the objective function is now explained. The second sum is computed by `ware1`.

Lines 51–52
Predicate `ware1` starts the second major branch of the program. The goal of this branch is to compute the sum

```
sum(W : Warehouses, S : Stores | Supplies(W,S)) (
             SupplyCost(S,W)
        )
```

The schema with collection is used, with the collect in Y as usual. This predicate is responsible for the local variable W.

Line 53
This call to `ware2` starts an iteration over S, the second level of the two fold iteration over (W:Warehouses, S:Stores). The partial collect is gathered in Z (in the schema with collection, intermediate collect vectors are always using the Z variable).

Line 54
The recursive call to `ware1` is taking care of the rest of the warehouses. The partial collect Z, containing computed values for the current value of W and all possible S, is used as input here. The final collect is gathered in Y.

Lines 55–56
Predicate `ware2` computes the body of the second sum in the objective function. It uses the schema with collection, with the collect in Y.

Line 57–60
The vlist mechanism. Note that Locals is empty; there are no locals in the outer scope here.

Line 61–62
These lines compute Supplies(W,S) and SupplyCost(S,W) respectively.

Line 63
The filtering factor Supplies(W,S) is multiplied by SupplyCost(S,W). The result is in `_5`.

Line 64
The recursive call to `ware2`. The resulting value, `_5`, is pushed onto the collect vector X.

All that remains now is to explain the expression in the such that clause of the model, lines 19–21. The branch starting with `ware5` takes care of that expression.

Line 34–35
Predicate `ware5` starts the third branch of the program, using the blank schema over (W : Warehouses). It enforces the constraint

```
forall(W : Warehouses) (
        count(0..Capacity(W))
                (S : Stores | Supplies(W,S))
        ).
```

Line 36–38
The vlist mechanism.

Line 39
Computing Capacity(W), result in `_19`.

Line 40–41
Invoking `ware6` to compute

```
count(0..Capacity(W)) (
        S : Stores | Supplies(W,S)).
```

The collect is in `_22`.

Line 42
Calling a library predicate to set the counting constraint on `_22`. The number of occurrences of "1s" in `_22` belongs to the interval 1..`_19`. Here `_19` is the result of the computation Capacity(W).

Line 43
Recursive call to `ware5`, taking care of the rest of the warehouses.

Line 44–45
The predicate `ware6` uses a schema with collection over (S : Stores), computing the expression `Supplies(W,S)`. The collect goes into Y.

Line 46–48
The vlist mechanism. Note that `_W` belongs to an outer scope, so it has to be passed via Locals.

Line 49

Computing the value of Supplies(W,S). Result goes into `_21`.

Line 50
Recursive call to `ware6` to take care of the rest of the stores. The collect is pushed onto X.

5 The Compiler, the Final Product

5.1 Installation

To install ESRA cd to the source directory 'src' and type 'make'. That will create an executable file named 'esra'. This executable doesn't depend on any libraries, databases or other files, so it may reside anywhere in your file system. Just place it somewhere and set your PATH accordingly.

5.2 The ESRA Libraries

The system contains two library files, esra.pl and lmatrix.pl. These Prolog module files are necessary to run the compiler generated programs. The first four lines of each generated program contain module declarations. The default location for the ESRA library files is the current directory, i.e. the one you stand in when you run the generated program. This location can be changed in two ways:

- Change the module declaration at the beginning of the generated program to include the full path names of esra.pl and lmatrix.pl. This will only affect this particular program.
- In the compiler, module translate.ml, there is a let binding of the identifier `library_path`. Change this binding to have the compiler placing the ESRA libraries in your favorite directory. This will affect all programs generated by the compiler.

5.3 Compiling and Running a Model

To compile an ESRA model, say `foo.esra`, just type `esra foo.esra`. If there were no compilation errors, you will get a program named `foo.pl`. If the compiler finds an error, it issues an error message and aborts the execution without producing any output. If the model depends on run time initialized constants, you will have to create a data file, which must have the name `foo.in`. Make sure that the module declarations at the top of the program. are consistent with the location of the ESRA libraries. There are two ways to run the generated program.

- Type `prolog` or `sicstus` at the shell prompt. In the interactive Prolog system, just type `[foo].` to consult your program. To run the program, just type the single goal `foo(R).`, where `R` is the *result parameter* (see section 4.7.2). The program will then print the results of the computation in a default format if successful, otherwise it answers with no.

- Use your emacs interface. Opening `foo.pl` in emacs, will get emacs into Prolog Mode. Typing the command C-c C-b (`consult_buffer`) will consult the program and open a Prolog process in a separate buffer, so that you can communicate with the program. Then just type `foo(R).` to start the program, where `R` is the *result parameter* (see section 4.7.2).

5.4 Using the Result API

The *result parameter* is supposed to be opaque. To extract information from it, a special API has been constructed, the so called *result API.* This API is defined in esra.pl. There are three predicates: `get_object/3`, `is_related/2` and `function_value/3` described below.

```
%% get_object(+Results, +Name, -Object) :-
%% Results is a result parameter, Name is an atom, Object is a value
%% True if Object is bound to Name in Results

%% is_related(+Arguments, +Relation) :-
%% Arguments is a list of integers
%% Relation is an ESRA relation
%% True if Relation(c1, c2, ..., cn) where ci = Arguments[i]

%% function_value(+Coordinates, +Function, -Value) :-
%% Arguments is a list of integers
%% Function is an ESRA total function
%% True if Function(c1, c2, ..., cn) = Value where ci = Arguments[i]
```

6 Testing and Benchmarking

6.1 Running the Test Suite

This product comes with a small test suite, located in the directory src/test. The models in this directory has been compiled and tested, so I know that they works. To compile all models in the test suite, cd to src/test and run the command `ctest` at the shell prompt. This will compile all test models. Running the test programs require two steps.
Enter Prolog and type

```
[ctest].
```

This will consult all programs and define a predicate 'run'.

Then type

```
run.
```

This will run all programs in the test suite.

To conclude this section I give an example. The program magic.pl is in the test suite. This is a model of so called *magic squares* (see section 3.2). In the accompanying data file magic.in.pl, the size of the magic square can be specified. The following is a sample session

```
| ?- [magic].
% consulting /home/mano7083/xjob/esra/test/magic.pl...
% consulted /home/mano7083/xjob/esra/test/magic.pl in module user, 0 msec 8 byt
yes
| ?- magic(R).
Initialization done, 179 constraints posted
Lapsed time : 0.01 seconds

Solution found in 0.0 seconds
2 backtracks were tried

S =
[8,3,4]
[1,5,9]
[6,7,2]

R = t('S',relation(reldom([interval(1,3),interval(1,3)],{1},1..1,
                          [interval(1,9)]),
                   lm_rel([[8,3,4],[1,5,9],[6,7,2]])),0,t,t) ?
```

Here we can see the output from the program. The variable S contains a 3x3 magic square. As we can see the sum of each row, each column and the two major diagonals is always 15 as it should be. Note that Prolog types the value of R at the end (the result parameter) followed by a question mark. The user can type a semicolon after the question mark to get more solutions by backtracking. The display of the result parameter is not very useful, but I don't know how to turn it off without at the same time disable backtracking.

6.2 Some Benchmarks

The benchmarks presented in this section seems to indicate some efficiency problems. CSPs are generally hard to solve, but this implementation gives unexpectedly fast growing execution times for most problems. I don't understand the reason for this behavior.

The magic Square Problem

In the *magic square problem*, section 3.2, there is a steep growth of computing time by the grid width n. In the table *constraints* is the number of posted constraints and *backtracks* the number of backtracks the solver had to make.

Magic Square Benchmarks			
n	*time* (s)	*backtracks*	*constraints*
3	0	2	179
4	0	13	413
5	2.03	4974	855
6	$>$ 2 min	?	1613

The Warehouse Location Problem

The computing time of the *warehouse location problem*, sections 3.6 and 4.9.2, is a rapidly growing function of the problem size as could be seen from the benchmarks. In the table below, n_w, n_s, stands for the number of warehouses and the number of stores respectively. The instance data are randomly generated. One can see that the computing time depends on n_w and n_s in different ways. The problem rapidly becomes intractable.

Two different models are compared. The model from section 3.6 uses the redundant array `open` keeping track of which warehouses are open. The model of section 4.9.2 on the other hand disposes of redundant variables. The first model is most efficient as seen by the table below. In the table a time value of the type '$>$ 2 min' means that the program has timed out with a time limit of 2 minutes. The first line of the table refers to a model published in the OPL book [14], the other instances are randomly generated.

Warehouse Location Problem Benchmarks							
Model of Section 3.6					Model of Section 4.9.2		
n_w	n_s	*time* (s)	*backtracks*	*constraints*	*time* (s)	*backtracks*	*constraints*
5	10	1.3	1098	839	23	58107	3496
8	7	0.61	890	1242	2.3	4109	947
10	10	19	20210	2183	$>$ 2 min	?	1664
5	15	41	87430	1628	$>$ 2 min	?	1244

The model of section 3.6 is more efficient than the other one as can be seen from the benchmark table above. The reason may be that the redundant variables and channeling constraints of the former model triggers more constraint propagation. It's important to notice that this behaviour may be different for different models and instances.

On a suggestion from Pierre Flener, I tried to flatten the Supplies array column wise instead of row wise (the default) before sending it to labeling/2. I also changed the default labeling option from [ffc] to [down]. This modification speeded up the computation. As a comparison I give benchmarks for the "optimized" versions of the same models and instances below.

Warehouse Location Problem Benchmarks

Model of Section 4.9.2 (optimized)				
n_w	n_s	*time* (s)	*backtracks*	*constraints*
5	10	0.8	2582	689
8	7	0.6	737	779
10	10	28	21176	1364
5	15	18	38098	1019
10	15	97	63557	2019
15	15	$>$ 2 min	?	3019

Although the optimized model performs better the scalability is still bad. In particular the 15 $\times$ 15 instance still times out. The model of section 3.6 is not suitable for this type of optimization because it uses two decision variable arrays instead of one and the Supply array is transposed with respect to the one in the other model so it does no sense to flatten it column wise. Therefore only one of the models has been included here.

The n Queens Problem

In the *n queens problem*, the computing time is first very low, but at approximately $n = 16$ it begins to grow rapidly. A strange pattern is that the time is invariably lower for odd values of n. For even ns, the problem becomes intractable at n = 22, but for odd values of n I can drive it a little bit father.

n Queens Benchmarks

n	*time* (s)	*backtracks*	*constraints*
8	0.01	23	464
9	0.01	6	594
10	0.05	23	740
11	0.01	9	902
12	0.04	45	1080
13	0.12	16	1274
14	0.17	296	1484
15	0.34	207	1790
16	2.63	1516	1952
17	1.8	875	2210
18	11.75	5545	2484
19	0.8	321	2774
20	60.42	24386	3080
21	2.37	783	3402
23	6.25	1910	4094
25	12.27	3161	4850
27	151.51	32444	5670
29	525.13	93393	6554

7 Conclusion

7.1 Summary

In this thesis I present an implementation of the modeling language ESRA[4, 3]. The compiler translates ESRA models into SICStus Prolog and submits them to the CLPFD [2] sub system for solution. In the interest of simplicity and rapid development, certain features of ESRA has been excluded from the implementation. This report is divided into two major parts. The first part gives a detailed definition of the language actually implemented, while the second part is an equally detailed presentation of the translation algorithms. The compiler itself is developed in OCaml, an object oriented functional language. The relational variables of ESRA are represented as "matrices", i.e. as iterated list structures simulating multi dimensional arrays. I have abstracted some Prolog code, for instance the code responsible for relational representation, into two separate Prolog library modules. The generated programs together with the library modules makes up an executable system, capable of solving CSPs written in ESRA. I have tested the compiler against several test models. In all cases I have found that the compiler generates correct code and that the code produces the correct answers. Some benchmarks of the warehouse location problem, and other problems, are also presented. As the benchmarks indicate, the implementation is not as efficient as one would wish. I don't understand the reasons for this inefficiency.

7.2 Related Work

Constraint logic programming over finite domains is a vast and important research field. Systems based on high level modeling languages are in the vogue and there are several ongoing projects in this direction all over the world.

There is also a predecessor of ESRA based on functional variables, here called *functional* ESRA. There already exists a compiler, written in Java, compiling functional ESRA into OPL [17, 9]. See also [9] for a general discussion of how to represent functional variables and *heuristics* for choosing the most suitable one for a certain class of problems and instance data.

The goal of the Australian G12 project [13] is to build a software platform for solving large scale industrial combinatorial optimization problems. The system will use Constraint Programming to allow problems to be stated simply, and then solved efficiently. The platform will consist of Zinc, a high level model language, Mercury, an already existing constraint programming language and Cadmium, a mapping language for transforming Zinc models into mercury programs. The complete platform, comprising Zinc, Cadmium and Mercury will be called G12 (since Zinc, Cadmium and Mercury belong to the 12th group in the periodic table of chemical elements).The system will work with several solvers mixing different solving paradigms, such as for instance constraint programming, mixed integer programming and local search. Such work will require the collaboration of researchers from different disciplines: operations researchers, graph algorithms researchers, meta-heuristics researchers, artificial intelligence researchers, and software engineers.

The Artificial Intelligence Group at University of York, UK, does research in the area of automated generation of constraint programs [7]. The project's aim is to develop an automated system that, given a specification of a problem, can generate one or more constraint programs that can solve the problem. The York system consists of two components:a *specification language* called ESSENCE [6] and a *refinement language* CONJURE [8]. ESSENCE goes far beyond ESRA in supplying a vast set of decision variable types such as sets, multisets, relations, functions, partitions which can be *nested to arbitrary depths*, for instance set, set of set, set of partitions, and so forth. Note that the York group have taken another approach to relational modeling than ESRA. A design goal of ESRA is to make the language initially as small as possible, deliberately omitting features such as nested data-types. The second development of the York group is the formulation and automation of a set of rules that can refine constraints on complex variables in an ESSENCE specification into constraints on atomic and atomic set variables, the level of abstraction provided by existing constraint toolkits. This is taken care of by the CONJURE language.

7.3 Future Work

This implementation of ESRA is a minimal one. I have deliberately left out features which I found difficult, unclear or tedious to implement. Now I will give suggestions of future extensions, both such ones that I left out and such ones that was never in the scope of this project.

The type checking is now interlaced with translation in my *one pass compiler*. I think it would be nice to lift it out and perform the type checking act on the parse tree before passing it to the translator. Also the error messages could be more user friendly.

An optimization pass may be introduced between the parser (type checker) and translator which attempts a transformation/simplification of the parse tree in order to produce more efficient code. For instance it would be possible to replace some loops with a *scalar product* constraint. Consider the following code snippet from the warehouse problem model in section 4.9.2

```
sum(W : Warehouses, S : Stores | Supplies(W,S)) (
        SupplyCost(S,W)
).
```

Normally this sum is computed by a double loop over W and S. But the inner loop can be replaced by a scalar product constraint instead. Conceptually this is equivalent to

```
sum(W : Warehouses) (
          scalar_product (Row-W[Supplies], Col-S[SupplyCost])
)
```

This can be implemented directly in SICStus Prolog which has a scalar_product constraint.

Sometimes an entire sub predicate can be eliminated by computing a sum directly on a relation variable rather than copy it row wise (a procedure pointed out by my supervisor Pierre Flener). Consider the following fragment of the warehouse location problem from section 4.9.2.

```
forall(W : Warehouses) (
       count(0..Capacity(W)) (
             S : Stores | Supplies(W,S)
```

```
        )
)
```

The normal way to translate this expression is by a double loop over W and S. The inner loop over S should just copy the array Supplies(W) and pass it back. This can be optimized by letting the outer loop set the counting constraint directly on Supplies(W). This is conceptually equivalent to

```
forall(W : Warehouses) (
        Count-Ones(Row-W[Supplies], Count)
        Count in 0..Capacity(W)
).
```

This is possible because the array Supplies is stored row wise, so that the rows are directly addressable without copying.

Some features that I have left out are really part of ESRA. For instance the card/1 operator and set comprehension notation as well as symbolic (enumerated) constants should be implemented. See section 2 for a full account of what has been missed out of the language.

The compiler should be *non deterministic*, i.e. produce several programs using different representations of the relational variables. This will aid the modeler (or a tool) in the task of choosing the most suitable representation for a certain model/instance data class. The modeler (or a tool) will then be able to *experiment* with different models without having to recompile.

Eventually, more intelligence will be built into the compiler via *heuristics* (such as those of [9]) for the compiler to rank the resulting compiled programs by decreasing likelihood of efficiency, without any recourse to experiments. Certain representations can be shown to be more efficient than others under certain circumstances depending on cardinality constraints and other factors [9, 12, 16].

Symmetry breaking is very important in solving such problems as BIBD which are loaded with symmetries. Automatic symmetry breaking in the compiler would be very desirable although hard to achieve.

Automatic generation of implied constraints is also very desirable to build into the compiler because it is often the key to efficient solving.

8 Grammar

Model

$$\langle\text{Model}\rangle \quad\longrightarrow\quad \langle\text{Decl}\rangle\,\langle\text{Objective}\rangle$$

Declarations

$$\langle\text{Decl}\rangle \longrightarrow \langle\text{DomDecl}\rangle \mid \langle\text{CstDecl}\rangle \mid \langle\text{VarDecl}\rangle \mid \langle\text{Decl}\rangle\,\langle\text{Decl}\rangle$$

Domain Declarations

$$\langle\text{DomDecl}\rangle \quad\longrightarrow\quad \texttt{dom}\,\langle\text{Id}\rangle\,\texttt{=}\,\langle\text{Expr}\rangle$$

Constant Declarations

$$\begin{array}{lcl}
\langle\text{CstDecl}\rangle & \longrightarrow & \texttt{cst}\,\langle\text{Id}\rangle\,\texttt{:}\,\langle\text{Expr}\rangle \\
 & \mid & \texttt{cst}\,\langle\text{Id}\rangle\,\texttt{=}\,\langle\text{Expr}\rangle\,\texttt{:}\,\langle\text{Expr}\rangle
\end{array}$$

Variable Declarations

$$\langle\text{VarDecl}\rangle \quad\longrightarrow\quad \texttt{var}\,\langle\text{Id}\rangle\,\texttt{:}\,\langle\text{Expr}\rangle$$

Objectives

$$\begin{array}{lcl}
\langle\text{Objective}\rangle & \longrightarrow & \texttt{solve}\,\langle\text{Expr}\rangle \\
 & \mid & \texttt{minimise}\,\langle\text{Expr}\rangle\,\texttt{such that}\,\langle\text{Expr}\rangle \\
 & \mid & \texttt{maximise}\,\langle\text{Expr}\rangle\,\texttt{such that}\,\langle\text{Expr}\rangle
\end{array}$$

Expressions

$$\begin{array}{lcl}
\langle\text{Expr}\rangle & \longrightarrow & \langle\text{Name}\rangle \mid \langle\text{Appl}\rangle \mid \langle\text{Tuple}\rangle \mid \langle\text{NumExpr}\rangle \\
 & \mid & \langle\text{SetExpr}\rangle \mid \langle\text{Formula}\rangle
\end{array}$$

$$\langle\text{Appl}\rangle \quad\longrightarrow\quad \langle\text{Id}\rangle\,\langle\text{Expr}\rangle$$

$$\langle\text{Tuple}\rangle \quad\longrightarrow\quad \texttt{(}\,\langle\text{Exprs}\rangle\,\texttt{)}$$

$$\langle\text{Exprs}\rangle \quad\longrightarrow\quad \langle\text{Expr}\rangle \mid \langle\text{Expr}\rangle\,\texttt{,}\,\langle\text{Exprs}\rangle$$

Numeric Expressions

$$\begin{array}{lcl}
\langle\text{NumExpr}\rangle & \longrightarrow & \langle\text{Int}\rangle \\
 & \mid & \langle\text{Expr}\rangle\,\langle\text{ArithBinOp}\rangle\,\langle\text{Expr}\rangle \\
 & \mid & \langle\text{ArithUnaryOp}\rangle\,\langle\text{Expr}\rangle \\
 & \mid & \texttt{sum}\,\texttt{(}\,\langle\text{QuantExpr}\rangle\,\texttt{)}\,\texttt{(}\,\langle\text{Expr}\rangle\,\texttt{)}
\end{array}$$

$$\langle\text{Int}\rangle \quad\longrightarrow\quad \langle\text{Nat}\rangle \mid \texttt{-}\langle\text{Nat}\rangle$$

$$\langle\text{Nat}\rangle \quad\longrightarrow\quad \langle\text{Digit}\rangle \mid \langle\text{Digit}\rangle\,\langle\text{Nat}\rangle$$

$$\langle\text{Digit}\rangle \quad\longrightarrow\quad \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$$

Set Expressions

```
⟨SetExpr⟩  ⟶  int | nat
           |   { ⟨Exprs⟩ }
           |   ⟨Expr⟩ .. ⟨Expr⟩
           |   ⟨Expr⟩ [ ⟨Expr⟩ ]
           |   ⟨Expr⟩ ⟨SetOp⟩ ⟨Expr⟩

⟨SetOp⟩  ⟶  [ ⟨Expr⟩ # ⟨Expr⟩ ]
         |   [ ⟨Expr⟩ #]
         |   [# ⟨Expr⟩ ]
         |   [#]
         |   [-> ⟨Expr⟩ ]
         |   [->]
         |   [+> ⟨Expr⟩ ]
         |   [+>]
```

Formulas

```
⟨Formula⟩  ⟶  true | false
           |   ⟨Expr⟩ ⟨PropOp⟩ ⟨Expr⟩
           |   ⟨Expr⟩ ⟨CompOp⟩ ⟨Expr⟩
           |   forall ( ⟨QuantExpr⟩ )( ⟨Expr⟩ )
           |   exists ( ⟨QuantExpr⟩ )
           |   count ( ⟨Expr⟩ )( ⟨QuantExpr⟩ )

⟨QuantExpr⟩  ⟶  ⟨QvarDecls⟩
             |   ⟨QvarDecls⟩ | ⟨Expr⟩

⟨QvarDecls⟩  ⟶  ⟨LclVarSpec⟩
             |   ⟨LclVarSpec⟩ , ⟨QvarDecls⟩

⟨LclVarSpec⟩  ⟶  ⟨Qvars⟩ : ⟨Expr⟩

⟨Qvars⟩  ⟶  ⟨GuardVars⟩ | ⟨ConjVars⟩

⟨ConjVars⟩  ⟶  ⟨Id⟩ | ⟨Id⟩ & ⟨ConjVars⟩

⟨GuardVars⟩  ⟶  ⟨Id⟩ ⟨CompOp⟩ ⟨Id⟩
```

Identifiers

```
⟨Id⟩  ⟶  ⟨Letter⟩
      |   ⟨Letter⟩ ⟨DigitsLetters⟩

⟨Ids⟩  ⟶  ⟨Id⟩
       |   ⟨Id⟩ , ⟨Ids⟩

⟨Letter⟩ ⟶ A | ... | Z | a | ... | z

⟨DigitsLetters⟩  ⟶  (⟨Digit⟩ | ⟨Letter⟩ | _)
                 |   (⟨Digit⟩ | ⟨Letter⟩ | _) ⟨DigitsLetters⟩
```

Operators

Comparison Operators

$\langle\text{CompOp}\rangle \quad \longrightarrow \quad$ `<` | `=<` | `=` | `>=` | `>` | `!=`

Arithmetic Operators

$\langle\text{ArithBinOp}\rangle \quad \longrightarrow \quad$ `+` | `-` | `*` | `/` | `%`

$\langle\text{ArithUnaryOp}\rangle \quad \longrightarrow \quad$ `-` | `abs`

Propositional Operators

$\langle\text{PropOp}\rangle \quad \longrightarrow \quad$ `/\` | `\/` | `=>` | `<=` | `<=>`

References

[1] K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[2] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proceedings of PLILP'97*, volume 1292 of *LNCS*, pages 191–206. Springer-Verlag, 1997.

[3] P. Flener, J. Pearson, and M. Ågren. The Syntax, Semantics, and Type System of esra. Technical report, ASTRA group, April 2003. Available at `http://www.it.uu.se/research/group/astra/`.

[4] P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In M. Bruynooghe, editor, *LOPSTR'03: Revised Selected Papers*, volume 3018 of *LNCS*, pages 214–232. Springer-Verlag, 2004.

[5] P. Flener, J. Pearson, and L. G. Reyna. Financial portfolio optimisation. In M. Wallace, editor, *Proceedings of CP'04*, volume 3258 of *LNCS*, pages 227–241. Springer-Verlag, 2004.

[6] A. Frisch, M. Grumm, C. Jefferson, and B. M. Hernandez. The essence of essence. In B.Hnich, P.Prosser, and B.Smith, editors, *Proceedings of the Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems, pages 73-88*, oct 2005. Available at `http://www.cs.york.ac.uk/aig/constraints/AutoModel`.

[7] A. Frisch, C. Jefferson, and B. M. Hernandez. The rules of constraint modelling: An overview. In *Proceedings of the 12th Workshop on Automated Reasoning,2005.* Available at `http://www.cs.york.ac.uk/aig/constraints/AutoModel`.

[8] A. Frisch, C. Jefferson, and B. M. Hernandez. The rules of constraint modelling. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, 109-116*, 2005. Available at `http://www.cs.york.ac.uk/aig/constraints/AutoModel`.

[9] B. Hnich. *Function Variables for Constraint Programming.* PhD thesis, Department of Information Science, Uppsala University, Sweden, 2003. Available at `http://publications.uu.se/theses/`.

[10] X. Leroy. *The Objective Caml System release 3.08.* Institut National de Recherche en Informatique et en Automatique, jul 2004. Available at `http://caml.inria.fr`.

[11] O. Sivertsson. Construction of synthetic CDO squared. Master's thesis, Computing Science, Department of Information Technology, Uppsala University, Sweden, 2005. Available as Technical Report 2005-042 at `http://www.it.uu.se/research/publications/reports/2005-042/`.

[12] B. M. Smith. Modelling a permutation problem. Technical Report 18, School of Computing, University of Leeds, UK, 2000. Also in *Proceedings of the ECAI'00 Workshop on Modelling and Solving Problems with Constraints.*

[13] P. Stuckey, M. de la Banda, M. Maher, K. Marriott, J. Slaney, Z. Somogyi, M. Wallace, and T. Walsh. The *g12 project*, mapping solver independent models to efficient solutions. Technical report. Available at `http://www.cs.mu.oz.au/~pjs/papers/g12.pdf`.

[14] P. Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, 1999.

[15] P. Van Hentenryck. Constraint and integer programming in OPL. *INFORMS Journal on Computing*, 14(4):345–372, 2002.

[16] T. Walsh. Permutation problems and channelling constraints. In R. Nieuwenhuis and A. Voronkov, editors, *Proceedings of LPAR'01*, volume 2250 of *LNCS*, pages 377–391. Springer-Verlag, 2001.

[17] S. Wrang. Implementation of the ESRA constraint modelling language. Master's thesis, Computing Science 223, Department of Information Technology, Uppsala University, Sweden, 2002. Available at `ftp://ftp.csd.uu.se/pub/papers/masters-theses/`.