ECL^{i}PS^e Obsolete Libraries Manual

Release 6.0

Pascal Brisset Hani El Sakkout Thom Frühwirth
Carmen Gervet Warwick Harvey Micha Meier
Stefano Novello Thierry Le Provost Joachim Schimpf
Kish Shen Mark Wallace

February 18, 2013

Contents

Chapter 1

Introduction

This manual contains documentation for libraries which are still part of the ECL^{i}PS^e distribution, but whose use is deprecated. Typically, the libraries have been replaced by newer implementation which provide similar or extended functionality. The old documentation is provided here mainly to ease the task of porting existing code to the newer libraries. Documentation for the new libraries can be found in the *Constraint Library Manual* and the *Reference Manual*. Here is a short overview of the obsolete libraries and where to find replacement functionality:

fd The numeric functionality of the finite-domain library is subsumed by the **ic** interval solver library. The symbolic domain constraints are provided by the **ic_symbolic** library. The branch-and-bound functionality can now be found in more generic form in the **branch_and_bound** library.

conjunto Most of this set solver's functionality is available in the new **ic_sets** library.

ic_eplex, range_eplex These libraries have now been removed and replaced by standalone eplex. Chapter@4 describes how to port your existing code from ic/range eplex to standalone eplex.

Chapter 2

The Finite Domains Library

The library **fd.pl** implements constraints over finite domains that can contain integer as well as atomic (i.e. atoms, strings, floats, etc.) and ground compound (e.g. *f(a, b)*) elements. Modules that use the library must start with the directive

:- use_module(library(fd)).

2.1 Terminology

Some of the terms frequently used in this chapter are explained below.

domain variable A domain variable is a variable which can be instantiated only to a value from a given finite set. Unification with a term outside of this domain fails. The domain can be associated with the variable using the predicate **::/2**. Built-in predicates that expect domain variables treat atomic and other ground terms as variables with singleton domains.

integer domain variable An integer domain variable is a domain variable whose domain contains only integer numbers. Only such variables are accepted in inequality constraints and in rational terms. Note that a non-integer domain variable can become an integer domain variable when the non-integer values are removed from its domain.

integer interval An integer interval is written as

$$Min \; .. \; Max$$

with integer expressions $Min \leq Max$ and it represents the set

$$\{\text{Min}, \text{Min} + 1, \ldots, \text{Max}\}.$$

linear term A linear term is a linear integer combination of integer domain variables. The constraint predicates accept linear terms even in a non-canonical form, containing functors +, - and *, e.g.

$$5 * (3 + (4 - 6) * Y - X * 3).$$

If the constraint predicates encounter a variable without a domain, they give it a default domain -10000000..10000000. Note that arithmetic operations on linear terms are performed with standard machine word integers without any overflow checks. If the domain ranges or coefficients are too large, the operation will not yield correct results. Both the maximum and minimum value of a linear term must be representable in a machine word, and so must the maximum and minimum value of every $c_i x_i$ term.

rational term A rational term is a term constructed from integers and integer domain variables using the arithmetic operations $+, -, *, /$. Besides that, every subexpression of the form *VarA/VarB* must have an integer value in the solution. The system replaces such a subexpression by a new variable *X* and adds a new constraint *VarA #= VarB * X*. Similarly, all subexpressions of the form *VarA*VarB* are replaced by a new variable *X* and a new constraint *X #= VarA * VarB* is added, so that in the internal representation, the term is converted to a linear term.

constraint expression A constraint expression is either an arithmetic constraint or a combination of constraint expressions using the logical FD connectives **#/\/2, #\//2, #=>/2, #<=>/2, #\+/1**.

2.2 Constraint Predicates

?Vars :: ?Domain

Vars is a variable or a list of variables with the associated domain *Domain*. *Domain* can be a closed integer interval denoted as *Min .. Max*, or a list of intervals and/or atomic or ground elements. Although the domain can contain any compound terms that contain no variable, the functor *../2* is reserved to denote integer intervals and thus *1..10* always means an interval and *a..b* is not accepted as a compound domain element.

If *Vars* is already a domain variable, its domain will be updated according to the new domain; if it is instantiated, the predicate checks if the value lies in the domain. Otherwise, if *Vars* is a free variable, it is converted to a domain variable. If *Vars* is a domain variable and

Domain is free, it is bound to the list of elements and integer intervals representing the domain of the variable (see also **dvar_domain/2** which returns the actual domain).

When a free variable obtains a finite domain or when the domain of a domain variable is updated, the **constrained** list of its **suspend** attribute is woken, if it has one.

integers(+Vars)

This constrains the list of variables Vars to have integer domains. Any non-domain variables in Vars will be given the default integer domain.

::(?Var, ?Domain, ?B)

B is equal to 1 iff the domain of the finite domain variable *Var* is a subset of *Domain* and 0 otherwise.

atmost(+Number, ?List, +Val)

At most *Number* elements of the list *List* of domain variables and ground terms are equal to the ground value *Val*.

constraints_number(+DVar, -Number)

Number is the number of constraints and suspended goals currently attached to the variable *DVar*. Note that this number may not correspond to the exact number of *different* constraints attached to *DVar*, as goals in different suspending lists are counted separately. This predicate is often used when looking for the most or least constrained variable from a set of domain variables (see also **deleteffc/3**).

element(?Index, +List, ?Value)

The *Index*'th element of the ground list *List* is equal to *Value*. *Index* and *Value* can be either plain variables, in which case a domain will be associated to them, or domain variables. Whenever the domain of *Index* or *Value* is updated, the predicate is woken and the domains are updated accordingly.

fd_eval(+E)

The constraint expression *E* is evaluated on runtime and no compile-time processing is performed. This might be necessary in the situations where the default compile-time transformation of the given expression is not suitable, e.g. because it would require type or mode information.

indomain(+DVar)

This predicate instantiates the domain variable *DVar* to an element of its domain; on backtracking the subsequent values are taken. It is used, for example, to find a value of *DVar* which is consistent with all currently imposed constraints. If *DVar* is a ground term, it succeeds. Otherwise, if it is not a domain variable, an error is raised.

is_domain(?Term)
Succeeds if *Term* is a domain variable.

is_integer_domain(?Term)
Succeeds if *Term* is an integer domain variable.

min_max(+Goal, ?C)
If *C* is a linear term, a solution of the goal *Goal* is found that minimises the value of *C*. If *C* is a list of linear terms, the returned solution minimises the maximum value of terms in the list. The solution is found using the *branch and bound* method; as soon as a partial solution is found that is worse than a previously found solution, failure is forced and a new solution is searched for. When a new better solution is found, the bound is updated and the search restarts from the beginning. Each time a new better solution is found, the event 280 is raised. If a solution does not make *C* ground, an error is raised, unless exactly one variable in the list *C* remains free, in which case the system tries to instantiate it to its minimum.

minimize(+Goal, ?Term)
Similar to **min_max/2**, but *Term* must be an integer domain variable. When a new better solution is found, the search does not restart from the beginning, but a failure is forced and the search continues. Each time a new better solution is found, the event 280 is raised. Often **minimize/2** is faster than **min_max/2**, sometimes **min_max/2** might run faster, but it is difficult to predict which one is more appropriate for a given problem.

min_max(+Goal, ?Template, ?Solution, ?C)

minimize(+Goal, ?Template, ?Solution, ?Term)
Similar to **min_max/2** and **minimize/2**, but the variables in *Goal* do not get instantiated to their optimum solutions. Instead, *Solutions* will be unified with a copy of *Template* where the variables are replaced with their minimized values. Typically, the template will contain all or a subset of *Goal*'s variables.

min_max(+Goal, ?C, +Low, +High, +Percent)

minimize(+Goal, ?Term, +Low, +High, +Percent)
Similar to **min_max/2** and **minimize/2**, however the branch and bound method starts with the assumption that the value to be minimised is less than or equal to *High*. Moreover, as soon as a solution is found whose minimised value is less than *Low*, this solution is returned. Solutions within the range of *Percent* % are considered equivalent and so the search for next better solution starts with a minimised value

Percent % less than the previously found one. *Low*, *High* and *Percent* must be integers.

min_max(+Goal, ?C, +Low, +High, +Percent, +Timeout)

minimize(+Goal, ?Term, +Low, +High, +Percent, +Timeout)
Similar to **min_max/5** and **minimize/5**, but after *Timeout* seconds the search is aborted and the best solution found so far is returned.

min_max(+Goal, ?Template, ?Solution, ?C, +Low, +High, +Percent, +Timeout)

minimize(+Goal, ?Template, ?Solution, ?Term, +Low, +High, +Percent, +Timeout)
The most general variants of the above, with all the optinal parameters.

2.3 Arithmetic Constraint Predicates

?T1 #\= ?T2 The value of the rational term *T1* is not equal to the value of the rational term *T2*.

?T1 #< ?T2 The value of the rational term *T1* is less than the value of the rational term *T2*.

?T1 #<= ?T2 The value of the rational term *T1* is less than or equal to the value of the rational term *T2*.

?T1 #= ?T2 The value of the rational term *T1* is equal to the value of the rational term *T2*.

?T1 #> ?T2 The value of the rational term *T1* is greater than the value of the rational term *T2*.

?T1 #>= ?T2 The value of the rational term *T1* is greater than or equal to the value of the rational term *T2*.

2.4 Logical Constraint Predicates

The logical constraints can be used to combine simpler constraints and to build complex logical constraint expressions. These constraints are preprocessed by the system and transformed into a sequence of evaluation constraints and arithmetic constraints. The logical operators are declared with the following precedences:

```
:- op(750, fy, #\+).
:- op(760, yfx, #/\).
:- op(770, yfx, #\/).
:- op(780, yfx, #=>).
:- op(790, yfx, #<=>).
```

#\+ +E1 *E1* is false, i.e. the logical negation of the constraint expression *E1* is imposed.

+E1 #/\+E2 Both constraint expressions *E1* and *E2* are true. This is equivalent to normal conjunction *(E1, E2)*.

+E1 #\/+E2 At least one of constraint expressions *E1* and *E2* is true. As soon as one of *E1* or *E2* becomes false, the other constraint is imposed.

+E1 #=> +E2 The constraint expression *E1* implies the constraint expression *E2*. If *E1* becomes true, then *E2* is imposed. If *E2* becomes false, then the negation of *E1* will be imposed.

+E1 #<=> +E2 The constraint expression *E1* is equivalent to the constraint expression *E2*. If one expression becomes true, the other one will be imposed. If one expression becomes false, the negation of the other one will be imposed.

2.5 Evaluation Constraint Predicates

These constraint predicates evaluate the given constraint expression and associate its truth value with a boolean variable. They can be very useful for defining more complex constraints. They can be used both to test entailment of a constraint and to impose a constraint or its negation on the current constraint store.

?B isd +Expr *B* is equal to 1 iff the constraint expression *Expr* is true, 0 otherwise. This predicate is the constraint counterpart of **is/2** — it takes a constraint expression, transforms all its subexpressions into calls to predicates with arity one higher and combines the resulting boolean values to yield *B*. For instance,

B isd X #= Y

is equivalent to

#=(X, Y, B)

#<(?T1, ?T2, ?B) *B* is equal to 1 iff the value of the rational term *T1* is less than the value of the rational term *T2*, 0 otherwise.

#<=(?T1, ?T2, ?B) *B* is equal to 1 iff the value of the rational term *T1* is less than or equal to the value of the rational term *T2*, 0 otherwise.

#=(?T1, ?T2, ?B) *B* is equal to 1 iff the value of the rational term *T1* is equal to the value of the rational term *T2*, 0 otherwise.

#\=(?T1, ?T2, ?B) *B* is equal to 1 iff the value of the rational term *T1* is different from the value of the rational term *T2*, 0 otherwise.

#>(?T1, ?T2, ?B) *B* is equal to 1 iff the value of the rational term *T1* is greater than the value of the rational term *T2*, 0 otherwise.

#>=(?T1, ?T2, ?B) *B* is equal to 1 iff the value of the rational term *T1* is greater than or equal to the value of the rational term *T2*, 0 otherwise.

#/\(+E1, +E2, ?B) *B* is equal to 1 iff both constraint expressions *E1* and *E2* are true, 0 otherwise.

#\/(+E1, +E2, ?B) *B* is equal to 1 iff at least one of the constraint expressions *E1* and *E2* is true, 0 otherwise.

#<=>(+E1, +E2, ?B) *B* is equal to 1 iff the constraint expression *E1* is equivalent to the constraint expression *E2*, 0 otherwise.

#=>(+E1, +E2, ?B) *B* is equal to 1 iff the constraint expression *E1* implies the constraint expression *E2*, 0 otherwise.

#\+(+E1, ?B) *B* is equal to 1 iff *E1* is false, 0 otherwise.

2.6 CHIP Compatibility Constraints Predicates

These constraints, defined in the module **fd_chip**, are provided for CHIP v.3 compatibility and they are defined using native ECL^{i}PS^e constraints. Their source is available in the file **fd_chip.pl**.

?T1 ## ?T2 The value of the rational term *T1* is not equal to the value of the rational term *T2*.

alldistinct(?List) All elements of *List* (domain variables and ground terms) are pairwise different.

deleteff(?Var, +List, -Rest) This predicate is used to select a variable from a list of domain variables which has the smallest domain. *Var* is the selected variable from *List*, *Rest* is the rest of the list without *Var*.

deleteffc(?Var, +List, -Rest) This predicate is used to select the most constrained variable from a list of domain variables. *Var* is the selected variable from *List* which has the least domain and which has the most constraints attached to it. *Rest* is the rest of the list without *Var*.

deletemin(?Var, +List, -Rest) This predicate is used to select the domain variable with the smallest lower domain bound from a list of domain variables. *Var* is the selected variable from *List*, *Rest* is the rest of the list without *Var*.

List is a list of domain variables or integers. Integers are treated as if they were variables with singleton domains.

dom(+DVar, -List) *List* is the list of elements in the domain of the domain variable *DVar*. The predicate **::/2** can also be used to query the domain of a domain variable, however it yields a list of intervals.

NOTE: This predicate should not be used in ECL^{i}PS^e programs, because all intervals in the domain will be expanded into element lists which causes unnecessary space and time overhead. Unless an explicit list representation is required, finite domains should be processed by the family of the **dom_*** predicates in sections 2.14.2 and 2.14.3.

maxdomain(+DVar, -Max) *Max* is the maximum value in the domain of the integer domain variable *DVar*.

mindomain(+DVar, -Min) *Min* is the minimum value in the domain of the integer domain variable *DVar*.

2.7 Utility Constraints Predicates

These constraints are defined in the module **fd_util** and they consist of useful predicates that are often needed in constraint programs. Their source code is available in the file **fd_util.pl**.

#(?Min, ?CstList, ?Max) The cardinality operator. *CstList* is a list of constraint expressions and this operator states that at least *Min* and at most *Max* out of them are valid.

dvar_domain_list(?Var, ?List) *List* is the list of elements in the domain of the domain variable or ground term *DVar*. The predicate **::/2** can also be used to query the domain of a domain variable, however it yields a list of intervals.

outof(?Var, +List) The domain variable *Var* is different from all elements of the list *List*.

labeling(+List) The elements of the *List* are instantiated using the **indomain/1** predicate.

2.8 Search Methods

A library of different search methods for finite domain problems is available as **library(fd_search)**. See the Reference Manual for details.

2.9 Domain Output

The library **fd_domain.pl** contains output macros which cause an **fd** attribute as well as a domain to be printed as lists that represent the domain values. A domain variable is an attributed variable whose **fd** attribute has a **print** handler which prints it in the same format. For instance,

```
[eclipse 4]: X::1..10, dvar_attribute(X, A), A = fd{domain:D}.

X = X{[1..10]}
D = [1..10]
A = [1..10]
yes.
[eclipse 5]: A::1..10, printf("%mw", A).
A{[1..10]}
A = A{[1..10]}
yes.
```

2.10 Debugging Constraint Programs

The ECL^{i}PS^e debugger is a low-level debugger which is suitable only to debug small constraint programs or to debug small parts of larger programs. Typically, one would use this debugger to debug user-defined constraints and Prolog data processing. When they are known to work properly, this debugger may not be helpful enough to find bugs (correctness debugging) or to speed up a working program (performance debugging). For this, the **display_matrix** tool from tkeclipse may be the appropriate tool.

2.11 Debugger Support

The ECL^{i}PS^e debugger supports debugging and tracing of finite domain programs in various ways. First of all, the debugger commands that handle suspended goals can be used to display suspended constraints (**d**, **^**, **u**) or to skip to a particular constraint (**w**, **i**). Note that most of the constraints are displayed using a write macro, their internal form is different.

Successive updates of a domain variable can be traced using the debug event **Hd**. When used, the debugger prompts for a variable name and then it skips to the port at which the domain of this variable was reduced. When a newline is typed instead of a variable name, it skips to the update of the previously entered variable.
A sequence of woken goals can be skipped using the debug event **Hw**.

2.12 Examples

A very simple example of using the finite domains is the *send more money* puzzle:

```
:- use_module(library(fd)).

send(List) :-
    List = [S, E, N, D, M, O, R, Y],
    List :: 0..9,
    alldifferent(List),
    1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E #=
        10000*M+1000*O+100*N+10*E+Y,
    M #\= 0,
    S #\= 0,
    labeling(List).
```

The problem is stated very simply, one just writes down the conditions that must hold for the involved variables and then uses the default *labeling* procedure, i.e. the order in which the variables will be instantiated. When executing **send/1**, the variables S, M and O are instantiated even before the labeling procedure starts. When a consistent value for the variable E is found (5), and this value is propagated to the other variables, all variables become instantiated and thus the rest of the labeling procedure only checks groundness of the list.
A slightly more elaborate example is the *eight queens* puzzle. Let us show a solution for this problem generalised to N queens and also enhanced by a cost function that evaluates every solution. The cost can be for example *coli - rowi* for the i-th queen. We are now looking for the solution with the smallest cost, i.e. one for which the maximum of all *coli - rowi* is minimal:

```
:- use_module(library(fd)).

% Find the minimal solution for the N-queens problem
cqueens(N, List) :-
    make_list(N, List),
```

```
    List :: 1..N,
    constrain_queens(List),
    make_cost(1, List, C),
    min_max(labeling(List), C).

% Set up the constraints for the queens
constrain_queens([]).
constrain_queens([X|Y]) :-
    safe(X, Y, 1),
    constrain_queens(Y).

safe(_, [], _).
safe(X, [Y|T], K) :-
    noattack(X, Y, K) ,
    K1 is K + 1 ,
    safe(X, T, K1).

% Queens in rows X and Y cannot attack each other
noattack(X, Y, K) :-
    X #\= Y,
    X + K #\= Y,
    X - K #\= Y.

% Create a list with N variables
make_list(0, []) :- !.
make_list(N, [_|Rest]) :-
    N1 is N - 1,
    make_list(N1, Rest).

% Set up the cost expression
make_cost(_, [], []).
make_cost(N, [Var|L], [N-Var|Term]) :-
    N1 is N + 1,
    make_cost(N1, L, Term).

labeling([]) :- !.
labeling(L) :-
    deleteff(Var, L, Rest),
    indomain(Var),
    labeling(Rest).
```

The approach is similar to the previous example: first we create the domain variables, one for each column of the board, whose values will be the rows. We state constraints which must hold between every pair of queens and

finally we make the cost term in the format required for the **min_max/2** predicate. The labeling predicate selects the most constrained variable for instantiation using the **deleteff/3** predicate. When running the example, we get the following result:

```
[eclipse 19]: cqueens(8, X).
Found a solution with cost 5
Found a solution with cost 4

X = [5, 3, 1, 7, 2, 8, 6, 4]
yes.
```

The time needed to find the minimal solution is about five times shorter than the time to generate all solutions. This shows the advantage of the *branch and bound* method. Note also that the board for this 'minimal' solution looks very nice.

2.13 General Guidelines to the Use of Domains

The *send more money* example already shows the general principle of solving problems using finite domain constraints:

- First the variables are defined and their domains are specified.
- Then the constraints are imposed on these variables. In the above example the constraints are simply built-in predicates. For more complicated problems it is often necessary to define Prolog predicates that process the variables and impose constraints on them.
- If stating the constraints alone did not solve the problem, one tries to assign values to the variables. Since every instantiation immediately wakes all constraints associated with the variable, and changes are propagated to the other variables, the search space is usually quickly reduced and either an early failure occurs or the domains of other variables are reduced or directly instantiated. This labeling procedure is therefore incomparably more efficient than the simple *generate and test* algorithm.

The complexity of the program and the efficiency of the solving depends very much on the way these three points are performed. Quite frequently it is possible to state the same problem using different sets of variables with different domains. A guideline is that the search space should be as small as possible, and thus e.g. five variables with domain 1..10 (i.e. search space size is 10^5) are likely to be better than twenty variables with domain 0..1 (space size 2^{20}).

The choice of constraints is also very important. Sometimes a redundant constraint, i.e. one that follows from the other constraints, can speed up the search considerably. This is because the system does not propagate *all* information it has to all concerned variables, because most of the time this would not bring anything, and thus it would slow down the search. Another reason is that the library performs no meta-level reasoning on constraints themselves (unlike the CHR library). For example, the constraints

```
X + Y #= 10, X + Y + Z #= 14
```

allow only the value 4 for Z, however the system is not able to deduce this and thus it has to be provided as a redundant constraint.
The constraints should be stated in such a way that allows the system to propagate all important domain updates to the appropriate variables.
Another rule of thumb is that creation of choice points should be delayed as long as possible. Disjunctive constraints, if there are any, should be postponed as much as possible. Labeling, i.e. value choosing, should be done after all deterministic operations are carried out.
The choice of the labeling procedure is perhaps the most sensitive one. It is quite common that only a very minor change in the order of instantiated variables can speed up or slow down the search by several orders of magnitude. There are very few common rules available. If the search space is large, it usually pays off to spend more time in selecting the next variable to instantiate. The provided predicates **deleteff/3** and **deleteffc/3** can be used to select the most constrained variable, but in many problems it is possible to extract even more information about which variable to instantiate next.
Often it is necessary to try out several approaches and see how they work, if they do. The profiler and the statistics package can be of a great help here, it can point to predicates which are executed too often, or choice points unnecessarily backtracked over.

2.14 User-Defined Constraints

The **fd.pl** library defines a set of low-level predicates which allow the user to process domain variables and their domains, modify them and write new constraint predicates.

2.14.1 The *fd* Attribute

A domain variable is a metaterm. The **fd.pl** library defines a metaterm attribute

fd{**domain** : **D**, **min** : **Mi**, **max** : **Ma**, **any** : **A**}

which stores the domain information together with several suspension lists. The attribute arguments have the following meaning:

- **domain** - the representation of the domain itself. Domains are treated as abstract data types, the users should not access them directly, but only using access and modification predicates listed below.
- **min** - a suspension list that should be woken when the minimum of the domain is updated
- **max** - a suspension list that should be woken when the maximum of the domain is updated
- **any** - a suspension list that should be woken when the domain is reduced no matter how.

The suspension list names can be used in the predicate **suspend/3** to denote an appropriate waking condition.
The attribute of a domain variable can be accessed with the predicate **dvar_attribute/2** or by unification in a matching clause:

```
get_attribute(_{fd:Attr}, A) :-
    -?->
    Attr = A.
```

Note however, that this matching clause succeeds even if the first argument is a metaterm but its **fd** attribute is empty. To succeed only for domain variables, the clause must be

```
get_attribute(_{fd:Attr}, A) :-
    -?->
    nonvar(Attr),
    Attr = A.
```

or to access directly attribute arguments, e.g. the domain

```
get_domain(_{fd:fd{domain:D}}, Dom) :-
    -?->
    D = Dom.
```

The **dvar_attribute/2** has the advantage that it returns an attribute-like structure even if its argument is already instantiated, which is quite useful when coding **fd** constraints.
The attribute arguments can be accessed by macros from the **structures.pl** library, if e.g. **Attr** is the attribute of a domain variable, the max list can be obtained as

arg(max of fd, Attr, Max)

or, using a unification

Attr = fd{max:Max}

2.14.2 Domain Access

The domains are represented as abstract data types, the users are not supposed to access them directly, instead a number of predicates and macros are available to allow operations on domains.

dom_check_in(+Element, +Dom) Succeed if the integer *Element* is in the domain *Dom*.

dom_compare(?Res, +Dom1, +Dom2) Works like **compare/3** for terms. *Res* is unified with

- = iff *Dom1* is equal to *Dom2*,
- < iff *Dom1* is a proper subset of *Dom2*,
- > iff *Dom2* is a proper subset of *Dom1*.

Fails if neither domain is a subset of the other one.

dom_member(?Element, +Dom) Successively instantiate *Element* to the values in the domain *Dom* (similar to **indomain/1**).

dom_range(+Dom, ?Min, ?Max) Return the minimum and maximum value in the integer domain *Dom*. Fails if *Dom* is a domain containing non-integer elements. This predicate can also be used to test if a given domain is integer or not.

dom_size(+Dom, ?Size) *Size* is the number of elements in the domain *Dom*.

2.14.3 Domain Operations

The following predicates operate on domains alone, without modifying domain *variables*. Most of them return the size of the resulting domain which can be used to test if any modification was done.

dom_copy(+Dom1, -Dom2) *Dom2* is a copy of the domain *Dom1*. Since the updates are done in-place, two domain variables must not share the same physical domain and so when defining a new variable with an existing domain, the domain has to be copied first.

dom_difference(+Dom1, +Dom2, -DomDiff, ?Size) The domain *DomDifference* is *Dom1* $\setminus$ *Dom2* and *Size* is the number of its elements. Fails if *Dom1* is a subset of *Dom2*.

dom_intersection(+Dom1, +Dom2, -DomInt, ?Size) The domain *DomInt* is the intersection of domains *Dom1* and *Dom2* and *Size* is the number of its elements. Fails if the intersection is empty.

dom_union(+Dom1, +Dom2, -DomUnion, ?Size) The domain *DomUnion* is the union of domains *Dom1* and *Dom2* and *Size* is the number of its elements. Note that the main use of the predicate is to yield the most specific generalisation of two domains, in the usual cases the domains become smaller, not bigger.

list_to_dom(+List, -Dom) Convert a list of ground terms and integer intervals into a domain *Dom*. It does not have to be sorted and integers and intervals may overlap.

integer_list_to_dom(+List, -Dom) Similar to **list_to_dom/2** , but the input list should contain only integers and integer intervals and it should be sorted. This predicate will merge adjacent integers and intervals into larger intervals whenever possible. typically, this predicate should be used to convert a sorted list of integers into a finite domain. If the list is known to already contain proper intervals, **sorted_list_to_dom/2** could be used instead.

sorted_list_to_dom(+List, -Dom) Similar to **list_to_dom/2**, but the input list is assumed to be already in the correct format, i.e. sorted and with correct integer and interval values. No checking on the list contents is performed.

2.14.4 Accessing Domain Variables

The following predicates perform various operations:

dvar_attribute(+DVar, -Attrib) *Attrib* is the attribute of the domain variable *DVar*. If *DVar* is instantiated, *Attrib* is bound to an attribute with a singleton domain and empty suspension lists.

dvar_domain(+DVar, -Dom) *Dom* is the domain of the domain variable *DVar*. If *DVar* is instantiated, *Dom* is bound to a singleton domain.

var_fd(?Var, +Dom) If *Var* is a free variable, it becomes a domain variable with the domain *Dom* and with empty suspension lists. The domain *Dom* is copied to make in-place updates logically sound. If *Var* is already a domain variable, its domain is intersected with the domain *Dom*. Fails if *Var* is not a variable.

dvar_msg(+DVar1, +DVar2, -MsgDVar) *MsgVar* is a domain variable which is the most specific generalisation of domain variables or ground values *Var1* and *Var2*.

2.14.5 Modifying Domain Variables

When the domain of a domain variable is reduced, some suspension lists stored in the attribute have to be scheduled and woken.
NOTE: In the **fd.pl** library the suspension lists are woken precisely when the event associated with the list occurs. Thus e.g. the **min** list is woken if and only if the minimum value of the variable's domain is changed, but it is not woken when the variable is instantiated to this minimum or when another element from the domain is removed. In this way, user-defined constraints can rely on the fact that when they are executed, the domain was updated in the expected way. On the other hand, user-defined constraints should also comply with this rule and they should take care not to wake lists when their waking condition did not occur. Most predicates in this section actually do all the work themselves so that the user predicates may ignore scheduling and waking completely.

dvar_remove_element(+DVar, +El) The element *El* is removed from the domain of *DVar* and all concerned lists are woken. If the resulting domain is empty, this predicate fails. If it is a singleton, *DVar* is instantiated. If the domain does not contain the element, no updates are made.

dvar_remove_smaller(+DVar, +El) Remove all elements in the domain of *DVar* which are smaller than the integer *El* and wake all concerned lists. If the resulting domain is empty, this predicate fails; if it is a singleton, *DVar* is instantiated.

dvar_remove_greater(+DVar, +El) Remove all elements in the domain of *DVar* which are greater than the integer *El* and wake all concerned lists. If the resulting domain is empty, this predicate fails; if it is a singleton, *DVar* is instantiated.

dvar_update(+DVar, +NewDom) If the size of the domain *NewDom* is 0, the predicate fails. If it is 1, the domain variable *DVar* is instantiated to the value in the domain. Otherwise, if the size of the new domain is smaller than the size of the domain variable's domain, the domain of *DVar* is replaced by *NewDom*, the appropriate suspension lists in its attribute are passed to the waking scheduler and so is the **constrained** list in the **suspend** attribute of the domain variable. If the size of the new domain is equal to the old one, no updates and no waking is done, i.e. this predicate does not check an explicit equality of both domains. If the size of the new domain is greater than the old one, an error is raised.

dvar_replace(+DVar, +NewDom) This predicate is similar to **dvar_update/2**, but it does not propagate the changes, i.e. no waking is done. If the

size of the new domain is 1, *DVar* is not instantiated, but it is given this singleton domain. This predicate is useful for local consistency checks.

2.15 Extensions

The **fd.pl** library can be used as a basis for further extensions. There are several hooks that make the interfacing easier:

- Each time a new domain variable is created, either in the **::/2** predicate or by giving it a default domain in a rational arithmetic expression, the predicate **new_domain_var/1** is called with the variable as argument. Its default definition does nothing. To use it, it is necessary to redefine it, i.e. to recompile it in the **fd** module, e.g. using **compile/2** or the tool body of **compile_term/1**.

- Default domains are created in the predicate **default_domain/1** in the **fd** module, its default definition is

  ```
  default_domain(Var) :- Var :: -10000000..10000000.
  ```

  It is possible to change default domains by redefining this predicate in the **fd** module.

2.16 Example of Defining a New Constraint

We will demonstrate creation of new constraints on the following example. To show that the constraints are not restricted to linear terms, we can take the constraint

$$X^2 + Y^2 \leq C.$$

Assuming that X and Y are domain variables, we would like to define such a predicate that will be woken as soon as one or both variables' domains are updated in such a way that would require updating the other variable's domain, i.e. updates that would propagate via this constraint. For simplicity we assume that X and Y are nonnegative. We will define the predicate **sq(X, Y, C)** which will implement this constraint:

```
:- use_module(library(fd)).

% A*A + B*B <= C
sq(A, B, C) :-
    dvar_domain(A, DomA),
    dvar_domain(B, DomB),
```

```
dom_range(DomA, MinA, MaxA),
dom_range(DomB, MinB, MaxB),
MiA2 is MinA*MinA,
MaB2 is MaxB*MaxB,
(MiA2 + MaB2 > C ->
    NewMaxB is fix(sqrt(C - MiA2)),
    dvar_remove_greater(B, NewMaxB)
;
    NewMaxB = MaxB
),
MaA2 is MaxA*MaxA,
MiB2 is MinB*MinB,
(MaA2 + MiB2 > C ->
    NewMaxA is fix(sqrt(C - MiB2)),
    dvar_remove_greater(A, NewMaxA)
;
    NewMaxA = MaxA
),
(NewMaxA*NewMaxA + NewMaxB*NewMaxB =< C ->
    true
;
    suspend(sq(A, B, C), 3, (A, B)->min)
),
wake.                % Trigger the propagation
```

The steps to be executed when this constraint becomes active, i.e. when the predicate **sq/3** is called or woken are the following:

1. We access the domains of the two variables using the predicate **dvar_domain/2** and and obtain their bounds using **dom_range/3**. Note that it may happen that one of the two variables is already instantiated, but these predicates still work as if the variable had a singleton domain.

2. We check if the maximum of one or the other variable is still consistent with this constraint, i.e. if there is a value in the other variable's domain that satisfies the constraint together with this maximum.

3. If the maximum value is no longer consistent, we compute the new maximum of the domain, and then update the domain so that all values greater than this value are removed using the predicate **dvar_remove_greater/2**. This predicate also wakes all concerned suspension lists and instantiates the variable if its new domain is a singleton.

4. After checking the updates for both variables we test if the constraint is now satisfied for all values in the new domains. If this is not the

case, we have to suspend the predicate so that it is woken as soon as the minimum of either domain is changed. This is done using the predicate **suspend/3**.

5. The last action is to trigger the execution of all goals that are waiting for the updates we have made. It is necessary to wake these goals **after** inserting the new suspension, otherwise updates made in the woken goals would not be propagated back to this constraint.

Here is what we get:

```
[eclipse 20]: [X,Y]::1..10, sq(X, Y, 50).

X = X{[1..7]}
Y = Y{[1..7]}

Delayed goals:
sq(X{[1..7]}, Y{[1..7]}, 50)
yes.
[eclipse 21]: [X,Y]::1..10, sq(X, Y, 50), X #> 5.

Y = Y{[1..3]}
X = X{[6, 7]}

Delayed goals:
sq(X{[6, 7]}, Y{[1..3]}, 50)
yes.
[eclipse 22]: [X,Y]::1..10, sq(X, Y, 50), X #> 5, Y #> 1.

X = 6
Y = Y{[2, 3]}
yes.
[eclipse 23]: [X,Y]::1..10, sq(X, Y, 50), X #> 5, Y #> 2.

X = 6
Y = 3
yes.
```

2.17 Program Examples

In this section we present some FD programs that show various aspects of the library usage.

2.17.1 Constraining Variable Pairs

The finite domain library gives the user the possibility to impose constraints on the value of a variable. How, in general, is it possible to impose constraints on two or more variables? For example, let us assume that we have a set of colours and we want to define that some colours fit with each other and others do not. This should work in such a way as to propagate possible changes in the domains as soon as this becomes possible.
Let us assume we have a symmetric relation that defines which colours fit with each other:

```
% The basic relation
fit(yellow, blue).
fit(yellow, red).
fit(blue, yellow).
fit(red, yellow).
fit(green, orange).
fit(orange, green).
```

The predicate **nice_pair(X, Y)** is a constraint and any change of the possible values of X or Y is propagated to the other variable. There are many ways in which this pairing can be defined in ECL^{i}PS^e. They are different solutions with different properties, but they yield the same results.

2.17.1.1 User-Defined Constraints

We use more or less directly the low-level primitives to handle finite domain variables. We collect all consistent values for the two variables, remove all other values from their domains and then suspend the predicate until one of its arguments is updated:

```
nice_pair(A, B) :-
        % get the domains of both variables
    dvar_domain(A, DA),
    dvar_domain(B, DB),
        % make a list of respective matching colours
    setof(Y, X^(dom_member(X, DA), fit(X, Y)), BL),
    setof(X, Y^(dom_member(Y, DB), fit(X, Y)), AL),
        % convert the lists to domains
    sorted_list_to_dom(AL, DA1),
    sorted_list_to_dom(BL, DB1),
        % intersect the lists with the original domains
    dom_intersection(DA, DA1, DA_New, _),
    dom_intersection(DB, DB1, DB_New, _),
        % and impose the result on the variables
    dvar_update(A, DA_New),
```

```
        dvar_update(B, DB_New),
            % unless one variable is already instantiated, suspend
            % and wake as soon as any element of the domain is removed
        (var(A), var(B) ->
            suspend(nice_pair(A, B), 2, [A,B]->any)
        ;
            true
        ).

    % Declare the domains
    colour(A) :-
        findall(X, fit(X, _), L),
        A :: L.
```

After defining the domains, we can state the constraints:

```
[eclipse 5]: colour([A,B,C]), nice_pair(A, B), nice_pair(B, C), A #\= green

B = B{[blue, green, red, yellow]}
C = C{[blue, orange, red, yellow]}
A = A{[blue, orange, red, yellow]}

Delayed goals:
  nice_pair(A{[blue, orange, red, yellow]}, B{[blue, green, red, yellow]})
  nice_pair(B{[blue, green, red, yellow]}, C{[blue, orange, red, yellow]})
```

This way of defining new constraints is often the most efficient one, but usually also the most tedious one.

2.17.1.2 Using the *element* Constraint

In this case we use the available primitive in the fd library. Whenever it is necessary to associate a fd variable with some other fd variable, the **element/3** constraint is a likely candidate. Sometimes it is rather awkward to use, because additional variables must be used, but it gives enough power:

```
nice_pair(A, B) :-
    element(I, [yellow, yellow, blue, red, green, orange], A),
    element(I, [blue, red, yellow, yellow, orange, green], B).
```

We define a new variable **I** which is a sort of index into the clauses of the fit predicate. The first colour list contains colours in the first argument of fit/2 and the second list contains colours from the second argument. The propagation is similar to that of the previous one.
When **element/3** can be used, it is usually faster than the previous approach, because **element/3** is partly implemented in C.

2.17.1.3 Using Evaluation Constraints

We can also encode directly the relations between elements in the domains of the two variables:

```
nice_pair(A, B) :-
    np(A, B),
    np(B, A).

np(A, B) :-
    [A,B] :: [yellow, blue, red, orange, green],
    A #= yellow #=> B :: [blue, red],
    A #= blue #=> B #= yellow,
    A #= red #=> B #= yellow,
    A #= green #=> B #= orange,
    A #= orange #=> B #= green.
```

This method is quite simple and does not need any special analysis; on the other hand it potentially creates a huge number of auxiliary constraints and variables.

2.17.1.4 Using Generalised Propagation

Propia is the first candidate to convert an existing relation into a constraint. One can simply use **infers most** to achieve the propagation:

```
nice_pair(A, B) :-
    fit(A, B) infers most.
```

Using Propia is usually very easy and the programs are short and readable, so that this style of constraints writing is quite useful e.g. for teaching. It is not as efficient as with user-defined constraints, but if the amount of propagation is more important that the efficiency of the constraint itself, it can yield good results, too.

2.17.1.5 Using Constraint Handling Rules

The `domain` solver in CHR can be used directly with the **element/3** constraint as well, however it is also possible to define directly domains consisting of pairs:

```
:- lib(chr).
:- chr(lib(domain)).

nice_pair(A, B) :-
    setof(X-Y, fit(X, Y), L),
```

```
A-B :: L.
```

The pairs are then constrained accordingly:

```
[eclipse 2]: nice_pair(A, B), nice_pair(B, C), A ne orange.

B = B
C = C
A = A

Constraints:
(9) A_g1484 - B_g1516 :: [blue - yellow, green - orange, red - yellow,
yellow - blue, yellow - red]
(10) A_g1484 :: [blue, green, red, yellow]
(12) B_g1516 - C_g3730 :: [blue - yellow, orange - green, red - yellow,
yellow - blue, yellow - red]
(13) B_g1516 :: [blue, orange, red, yellow]
(14) C_g3730 :: [blue, green, red, yellow]
```

2.17.2 Puzzles

Various kinds of puzzles can be easily solved using finite domains. We show here the classical Lewis Carrol's puzzle with five houses and a zebra:

```
Five men with different nationalities live in the first five houses
of a street.  They practise five distinct professions, and each of
them has a favourite animal and a favourite drink, all of them
different.  The five houses are painted in different colours.

The Englishman lives in a red house.
The Spaniard owns a dog.
The Japanese is a painter.
The Italian drinks tea.
The Norwegian lives in the first house on the left.
The owner of the green house drinks coffee.
The green house is on the right of the white one.
The sculptor breeds snails.
The diplomat lives in the yellow house.
Milk is drunk in the middle house.
The Norwegian's house is next to the blue one.
The violinist drinks fruit juice.
The fox is in a house next to that of the doctor.
The horse is in a house next to that of the diplomat.

Who owns a Zebra, and who drinks water?
```

One may be tempted to define five variables Nationality, Profession, Colour, etc. with atomic domains to represent the problem. Then, however, it is quite difficult to express equalities over these different domains. A much simpler solution is to define 5x5 integer variables for each mentioned item, to number the houses from one to five and to represent the fact that e.g. Italian drinks tea by equating Italian = Tea. The value of both variables represents then the number of their house. In this way, no special constraints are needed and the problem is very easily described:

```
:- lib(fd).

zebra([zebra(Zebra), water(Water)]) :-
    Sol = [Nat, Color, Profession, Pet, Drink],
    Nat = [English, Spaniard, Japanese, Italian, Norwegian],
    Color = [Red, Green, White, Yellow, Blue],
    Profession = [Painter, Sculptor, Diplomat, Violinist, Doctor],
    Pet = [Dog, Snails, Fox, Horse, Zebra],
    Drink = [Tea, Coffee, Milk, Juice, Water],

    % we specify the domains and the fact
    % that the values are exclusive
    Nat :: 1..5,
    Color :: 1..5,
    Profession :: 1..5,
    Pet :: 1..5,
    Drink :: 1..5,
    alldifferent(Nat),
    alldifferent(Color),
    alldifferent(Profession),
    alldifferent(Pet),
    alldifferent(Drink),

    % and here follow the actual constraints
    English = Red,
    Spaniard = Dog,
    Japanese = Painter,
    Italian = Tea,
    Norwegian = 1,
    Green = Coffee,
    Green #= White + 1,
    Sculptor = Snails,
    Diplomat = Yellow,
    Milk = 3,
    Dist1 #= Norwegian - Blue, Dist1 :: [-1, 1],
```

```
Violinist = Juice,
Dist2 #= Fox - Doctor, Dist2 :: [-1, 1],
Dist3 #= Horse - Diplomat, Dist3 :: [-1, 1],

flatten(Sol, List),
labeling(List).
```

2.17.3 Bin Packing

In this type of problems the goal is to pack a certain amount of different things into the minimal number of bins under specific constraints. Let us solve an example given by Andre Vellino in the Usenet group comp.lang.prolog, June 93:

- There are 5 types of components:

  glass, plastic, steel, wood, copper

- There are three types of bins:

  red, blue, green

- whose capacity constraints are:

  - red has capacity 3
  - blue has capacity 1
  - green has capacity 4

- containment constraints are:

  - red can contain glass, wood, copper
  - blue can contain glass, steel, copper
  - green can contain plastic, wood, copper

- and requirement constraints are (for all bin types):

  wood requires plastic

- Certain component types cannot coexist:

  - glass exclusive copper
  - copper exclusive plastic

- and certain bin types have capacity constraint for certain components

  - red contains at most 1 of wood
  - green contains at most 2 of wood

- Given an initial supply of: 1 of glass, 2 of plastic, 1 of steel, 3 of wood, 2 of copper, what is the minimum total number of bins required to contain the components?

To solve this problem, it is not enough to state constraints on some variables and to start a labeling procedure on them. The variables are namely not known, because we don't know how many bins we should take. One possibility would be to take a large enough number of bins and to try to find a minimum number. However, usually it is better to generate constraints for an increasing fixed number of bins until a solution is found.
The predicate **solve/1** returns the solution for this particular problem, **solve_bin/2** is the general predicate that takes an amount of components packed into a **cont/5** structure and it returns the solution.

```
solve(Bins) :-
    solve_bin(cont(1, 2, 1, 3, 2), Bins).
```

solve_bin/2 computes the sum of all components which is necessary as a limit value for various domains, calls **bins/4** to generate a list **Bins** with an increasing number of elements and finally it labels all variables in the list:

```
solve_bin(Demand, Bins) :-
    Demand = cont(G, P, S, W, C),
    Sum is G + P + S + W + C,
    bins(Demand, Sum, [Sum, Sum, Sum, Sum, Sum, Sum], Bins),
    label(Bins).
```

The predicate to generate a list of bins with appropriate constraints works as follows: first it tries to match the amount of remaining components with zero and the list with nil. If this fails, a new bin represented by a list

[**Colour**, **Glass**, **Plastic**, **Steel**, **Wood**, **Copper**]

is added to the bin list, appropriate constraints are imposed on all the new bin's variables, its contents is subtracted from the remaining number of components, and the predicate calls itself recursively:

```
bins(cont(0, 0, 0, 0, 0), 0, _, []).
bins(cont(G0, P0, S0, W0, C0), Sum0, LastBin, [Bin|Bins]) :-
    Bin = [_Col, G, P, S, W, C],
    bin(Bin, Sum),
    G2 #= G0 - G,
    P2 #= P0 - P,
    S2 #= S0 - S,
    W2 #= W0 - W,
    C2 #= C0 - C,
    Sum2 #= Sum0 - Sum,
```

```
    ordering(Bin, LastBin),
    bins(cont(G2, P2, S2, W2, C2), Sum2, Bin, Bins).
```

The **ordering/2** constraints are strictly necessary because this problem has a huge number of symmetric solutions.
The constraints imposed on a single bin correspond exactly to the problem statement:

```
bin([Col, G, P, S, W, C], Sum) :-
    Col :: [red, blue, green],
    [Capacity, G, P, S, W, C] :: 0..4,
    G + P + S + W + C #= Sum,
    Sum #> 0,               % no empty bins
    Sum #<= Capacity,
    capacity(Col, Capacity),
    contents(Col, G, P, S, W, C),
    requires(W, P),
    exclusive(G, C),
    exclusive(C, P),
    at_most(1, red, Col, W),
    at_most(2, green, Col, W).
```

We will code all of the special constraints with the maximum amount of propagation to show how this can be achieved. In most programs, however, it is not necessary to propagate all values everywhere which simplifies the code quite considerably. Often it is also possible to use some of the built-in symbolic constraints of ECL^{i}PS^e, e.g. **element/3** or **atmost/3**.

2.17.3.1 Capacity Constraints

capacity(Color, Capacity) should instantiate the capacity if the colour is known, and reduce the colour values if the capacity is known to be greater than some values. If we use evaluation constraints, we can code the constraint directly, using equivalences:

```
capacity(Color, Capacity) :-
    Color #= blue #<=> Capacity #= 1,
    Color #= green #<=> Capacity #= 4,
    Color #= red #<=> Capacity #= 3.
```

A more efficient code would take into account the ordering on the capacities. Concretely, if the capacity is greater than 1, the colour cannot be blue and if it is greater than 3, it must be green:

```
capacity(Color, Capacity) :-
    var(Color),
```

```
        !,
        dvar_domain(Capacity, DC),
        dom_range(DC, MinC, _),
        (MinC > 1 ->
            Color #\= blue,
            (MinC > 3 ->
                Color = green
            ;
                suspend(capacity(Color, Capacity), 3, (Color, Capacity)->inst)
            )
        ;
            suspend(capacity(Color, Capacity), 3, [Color->inst, Capacity->min])
        ).
    capacity(blue, 1).
    capacity(green, 4).
    capacity(red, 3).
```

Note that when suspended, the predicate waits for colour instantiation or for minimum of the capacity to be updated (except that 3 is one less than the maximum capacity and thus waiting for its instantiation is equivalent).

2.17.3.2 Containment Constraints

The containment constraints are stated as logical expressions and this is also the easiest way to medel them. The important point to remember is that a condition like *red can contain glass, wood, copper* actually means *red cannot contain plastic or steel* which can be written as

```
    contents(Col, G, P, S, W, _) :-
        Col #= red #=> P #= 0 #/\ S #= 0,
        Col #= blue #=> P #= 0 #/\ W #= 0,
        Col #= green #=> G #= 0 #/\ S #= 0.
```

If we want to model the containment with low-level domain predicates, it is easier to state them in the equivalent conjugate form:

- glass can be contained in red or blue
- plastic can be contained in green
- steel can be contained in blue
- wood can be contained in red, green
- copper can be contained in red, blue, green

or in a further equivalent form that uses at most one bin colour:

- glass can not be contained in green
- plastic can be contained in green
- steel can be contained in blue
- wood can not be contained in blue
- copper can be contained in anything

```
contents(Col, G, P, S, W, _) :-
    not_contained_in(Col, G, green),
    contained_in(Col, P, green),
    contained_in(Col, S, blue),
    not_contained_in(Col, W, blue).
```

contained_in(Color, Component, In) states that if Color is different from In, there can be no such component in it, i.e. Component is zero:

```
contained_in(Col, Comp, In) :-
    nonvar(Col),
    !,
    (Col \== In ->
        Comp = 0
    ;
        true
    ).
contained_in(Col, Comp, In) :-
    dvar_domain(Comp, DM),
    dom_range(DM, MinD, _),
    (MinD > 0 ->
        Col = In
    ;
        suspend(contained_in(Col, Comp, In), 2, [Comp->min, Col->inst])
    ).
```

not_contained_in(Color, Component, In) states that if the bin is of the given colour, the component cannot be contained in it:

```
not_contained_in(Col, Comp, In) :-
    nonvar(Col),
    !,
    (Col == In ->
        Comp = 0
    ;
        true
    ).
```

```
not_contained_in(Col, Comp, In) :-
    dvar_domain(Comp, DM),
    dom_range(DM, MinD, _),
    (MinD > 0 ->
        Col #\= In
    ;
        suspend(not_contained_in(Col, Comp, In), 2, [Comp->min, Col->any])
    ).
```

As you can see again, modeling with the low-level domain predicates might give a faster and more precise programs, but it is much more difficult than using constraint expressions and evaluation constraints. A good approach is thus to start with constraint expressions and only if they are not efficient enough, to (stepwise) recode some or all constraints with the low-level predicates.

2.17.3.3 Requirement Constraints

The constraint 'A requires B' is written as

```
requires(A, B) :-
    A #> 0 #=> B #> 0.
```

With low-level predicates, the constraint 'A requires B' is woken as soon as some A is present or B is known:

```
requires(A, B) :-
    nonvar(B),
    !,
    ( B = 0 ->
        A = 0
    ;
        true
    ).
requires(A, B) :-
    dvar_domain(A, DA),
    dom_range(DA, MinA, _),
    ( MinA > 0 ->
        B #> 0
    ;
        suspend(requires(A, B), 2, [A->min, B->inst])
    ).
```

2.17.3.4 Exclusive Constraints

The exclusive constraint can be written as

```
exclusive(A, B) :-
    A #> 0 #=> B #= 0,
    B #> 0 #=> A #= 0.
```

however a simple form with one disjunction is enough:

```
exclusive(A, B) :-
    A #= 0 #\/ B #= 0.
```

With low-level domain predicates, the exclusive constraint defines a suspension which is woken as soon as one of the two components is present:

```
exclusive(A, B) :-
    dvar_domain(A, DA),
    dom_range(DA, MinA, MaxA),
    ( MinA > 0 ->
        B = 0
    ; MaxA = 0 ->
        % A == 0
        true
    ;
        dvar_domain(B, DB),
        dom_range(DB, MinB, MaxB),
        ( MinB > 0 ->
            A = 0
        ; MaxB = 0 ->
            % B == 0
            true
        ;
            suspend(exclusive(A, B), 3, (A,B)->min)
        )
    ).
```

2.17.3.5 Atmost Constraints

at_most(N, In, Colour, Components) states that if Colour is equal to In, then there can be at most N Components and vice versa, if there are more than N Components, the colour cannot be In. With constraint expressions, this can be simply coded as

```
at_most(N, In, Col, Comp) :-
    Col #= In #=> Comp #<= N.
```

A low-level solution looks as follows:

```
at_most(N, In, Col, Comp) :-
    nonvar(Col),
```

```
        !,
        (In = Col ->
            Comp #<= N
        ;
            true
        ).
    at_most(N, In, Col, Comp) :-
        dvar_domain(Comp, DM),
        dom_range(DM, MinM, _),
        (MinM > N ->
            Col #\= In
        ;
            suspend(at_most(N, In, Col, Comp), 2, [In->inst, Comp->min])
        ).
```

2.17.3.6 Ordering Constraints

To filter out symmetric solutions we can e.g. impose a lexicographic ordering on the bins in the list, i.e. the second bin must be lexicographically greater or equal than the first one etc. As long as the corresponding most significant variables in two consecutive bins are not instantiated, we cannot constrain the following ones and thus we suspend the ordering on the **inst** lists:

```
    ordering([], []).
    ordering([Val1|Bin1], [Val2|Bin2]) :-
        Val1 #<= Val2,
        (integer(Val1) ->
            (integer(Val2) ->
                (Val1 = Val2 ->
                    ordering(Bin1, Bin2)
                ;
                    true
                )
            ;
                suspend(ordering([Val1|Bin1], [Val2|Bin2]), 2, Val2->inst)
            )
        ;
            suspend(ordering([Val1|Bin1], [Val2|Bin2]), 2, Val1->inst)
        ).
```

There is a problem with the representation of the colour: If the colour is represented by an atom, we cannot apply the **#<=/2** predicate on it. To keep the ordering predicate simple and still have a symbolic representation of the colour in the program, we can define input macros that transform the colour atoms into integers:

```
:- define_macro(no_macro_expansion(blue)/0, tr_col/2, []).
:- define_macro(no_macro_expansion(green)/0, tr_col/2, []).
:- define_macro(no_macro_expansion(red)/0, tr_col/2, []).

tr_col(no_macro_expansion(red), 1).
tr_col(no_macro_expansion(green), 2).
tr_col(no_macro_expansion(blue), 3).
```

2.17.3.7 Labeling

A straightforward labeling would be to flatten the list with the bins and use e.g. **deleteff/3** to label a variable out of it. However, for this example not all variables have the same importance — the colour variables propagate much more data when instantiated. Therefore, we first filter out the colours and label them before all the component variables:

```
label(Bins) :-
    colours(Bins, Colors, Things),
    flatten(Things, List),
    labeleff(Colors),
    labeleff(List).

colours([], [], []).
colours([[Col|Rest]|Bins], [Col|Cols], [Rest|Things]) :-
    colours(Bins, Cols, Things).

labeleff([]).
labeleff(L) :-
    deleteff(V, L, Rest),
    indomain(V),
    labeleff(Rest).
```

Note also that we need a special version of **flatten/3** that works with non-ground lists.

2.18 Current Known Restrictions and Bugs

1. The default domain for integer finite domain variables is -10000000..10000000. Larger domains must be stated explicitly using the ::/2 predicate, however neither bound can be outside the standard integer range for the machine (usually 32 bits).

2. Linear integer terms are processed using machine integers and thus if the maximum or minimum value of a linear term overflows this range (usually 32 bits), incorrect results are reported. This may occur if

large coefficients are used, if domains are too large or a combination of the two.

Chapter 3

The Set Domain Library

Note: As of ECL^{i}PS^e release 5.1, the library described in this chapter is being phased out and replaced by the new set solver library lib(ic_sets). See the corresponding chapters in the *Library Manual* and the Reference Manual for details.

Conjunto is a system to solve set constraints over finite set domain terms. It has been developed using the kernel of ECL^{i}PS^e based on metaterms. It contains the finite domain library of ECL^{i}PS^e. The library **conjunto.pl** implements constraints over set domain terms that contain herbrand terms as well as ground sets. Modules that use the library must start with the directive

```
:- use_module(library(conjunto))
```

For those who are already familiar with the ECL^{i}PS^e constraint library manual this manual follows the finite domain library structure.
For further information about this library, please email to c.gervet@icparc.ic.ac.uk.

3.1 Terminology

The computation domain of **Conjunto** is finite so set domain and set term will stand respectively for finite set domain and finite set term in the following. Here are defined some of the terms mainly used in the predicate description.

Ground set

A known finite set containing only atoms from the Herbrand Universe or its powerset but without any variable.

Set domain

A discrete lattice or powerset D attached to a set variable S. D is defined by $\{S \in 2^{lub\_s} \mid glb\_s \subseteq S\}$ under inclusion specified

by the notation *Glb_s..Lub_s*. *Glb_s* and *Lub_s* represent respectively the intersection and union of elements of D. Thus they are both ground sets. S is then called a **set domain variable**.

Weighted set domain

A specific set domain *WD* attached to a set variable *S* where each element of *WD* is of the form *e(s,w)*. *s* is a ground set representing a possible value of the set variable and *w* is the weight or cost associated to this value. e.g.

```
WD = {e(1,50),e({1,3},20)}..{e(1,50),e({1,3},20),e(f(a),100)}.
```

D would have been:

```
{1,{1,3}}..{1,{1,3},f(a)}.
```

Set expression

A composition of set domain variables or ground sets together with set operator symbols which are the standard ones coming from set theory.

$S ::= S\_1 \cap S\_2 \mid S\_1 \cup S\_2 \mid S\_1 \setminus S\_2$

Set term

Any term of the followings: (1) a ground set, (2) a set domain variable or (3) a set expression. All set built-in predicates deal with set terms thus with any of the three cases.

3.2 Syntax

- A ground set is written using the characters `{` and `}`, e.g. `S = {1,3,{a,g}, f(2)}`
- A domain D attached to a set variable is specified by two ground sets : *Glb_s..Lub_s*
- Set expressions: Unfortunately the characters representing the usual set operators are not available on our monitors so we use a specific syntax making the connection with arithmetic operators:
  - $\cup$ is represented by \/
  - $\cap$ is represented by /\
  - $\setminus$ is represented by \

3.3 The solver

The **Conjunto** solver acts in a data driven way using a relation between *states*. The transformation performs interval reduction over the set domain bounds. The set expression domains are approximated in terms of the domains of the set variables involved. From a constraint propagation viewpoint this means that constraints over set expressions can be approximated in terms of constraints over set variables. A failure is detected in the constraint propagation phase as soon as one domain lower bound *glb_s* is not included in its associated upper bound *lub_s*. Once a solved form has been reached all the constraints which are not definitely solved are delayed and attached to the concerned set variables.

3.4 Constraint predicates

?Svar ‘:: ++Glb..++Lub

attaches a domain to the set variable or to a list of set variables *Svar*. If $Glb \not\subseteq Lub$ it fails. If *Svar* is already a domain variable its domain will be updated according to the new domain; if *Svar* is instantiated it fails. Otherwise if *Svar* is free it becomes a set variable.

set(?Term)

succeeds if *Term* is a ground set.

?S ‘= ?S1

The value of the set term *S* is equal to the value of the set term *S1*.

?E in ?S

The element *E* is an element of *S*. If *E* is ground it is added to the lower bound of the domain of *S*, otherwise the constraint is delayed. If *E* is ground and does not belong to the upper bound of *S* domain, it fails.

?E notin ?S

The element *E* does not belong to *S*. If *E* is ground it is removed from the upper bound of *S*, otherwise the constraint is delayed. If *E* is ground and belongs to the upper bound of the domain of *S*, it is removed from the upper bound and the constraint is solved. If *E* is ground and belongs to the lower bound of *S* domain, it fails.

?S ‘< ?S1

The value of the set term *S* is a subset of the value of the set term **S1**. If the two terms are ground sets it just checks the inclusion and succeeds or fails. If the lower bound of the domain of *S* is not included in the upper bound of *S1* domain, it fails. Otherwise it checks the inclusion over the bounds. The constraint is then delayed.

?S ‘<> ?S1

The domains of S and S1 are disjoint (intersection empty).

all_union(?Lsets, ?S)

Lsets is a list of set variables or ground sets. *S* is a set term which is the union of all these sets. If *S* is a free variable, it becomes a set variable and its attached domain is defined from the union of the domains or ground sets in *Lsets*.

all_disjoint(?Lsets)

Lsets is a list of set variables of ground sets. All the sets are pairwise disjoint.

#(?S,?C)

S is a set term and *C* its cardinality. C can be a free variable, a finite domain variable or an integer. If C is free, this predicate is a mean to access the set cardinality and attach it to C. If not, the cardinality of S is constrained to be C.

sum_weight(?S,?W)

S is a set variable whose domain is a *weighted domain*. *W* is the weight of *S*. If *W* is a free variable, this predicate is a mean to access the set weight and attach it to W. If not, the weight of S is constrained to be W. e.g.

```
S ‘:: {e(2,3)}..{e(2,3), e(1,4)}, sum_weight(S, W)
```

returns W :: 3..7.

refine(?Svar)

If *Svar* is a set variable, it labels *Svar* to its first possible domain value. If there are several instances of *Svar*, it creates choice points. If *Svar* is a ground set, nothing happens. Otherwise it fails.

3.5 Examples

3.5.1 Set domains and interval reasoning

First we give a very simple example to demonstrate the expressiveness of set constraints and the propagation mechanism.

```
:- use_module(library(conjunto)).

[eclipse 2]: Car `:: {renault} .. {renault, bmw, mercedes, peugeot},
    Type_french = {renault, peugeot} , Choice `= Car /\ Type_french.

Choice = Choice{{renault} .. {peugeot, renault}}
Car = Car{{renault} .. {bmw, mercedes, peugeot, renault}}
Type_french = {peugeot, renault}

Delayed goals:
      inter_s({peugeot, renault}, Car{{renault}..{bmw, mercedes,
          peugeot, renault}}, Choice{{renault} .. {peugeot, renault}})
yes.
```

If now we add one cardinality constraint:

```
[eclipse 3]: Car `:: {renault} .. {renault, bmw, mercedes, peugeot},
    Type_french = {renault, peugeot} , Choice `= Car /\ Type_french,
    #(Choice, 2).

Car = Car{{peugeot, renault} .. {bmw, mercedes, peugeot, renault}}
Type_french = {peugeot, renault}
Choice = {peugeot, renault}
yes.
```

The first example gives a set of cars from which we know `renault` belongs to. The other labels `{renault, bmw, mercedes, peugeot}` are possible elements of this set. The `Type_french` set is ground and `Choice` is the set term resulting from the intersection of the first two sets. The first execution tells us that `renault` is element of `Choice` and `peugeot` might be one. The intersection constraint is partially satisfied and might be reconsidered if one of the domain of the set terms involved changes. The cosntraint is delayed. In the second example an additional constraint restricts the cardinality of `Choice` to 2. Satisfying this constraint implies setting the `Choice` set to `{peugeot, renault}`. The domain of this set has been modified so is the intersection constraint activated and solved again. The final result adds `peugeot` to the `Car` set variable. The intersection constraint is now satisfied and removed from the constraint store.

3.5.2 Subset-sum computation with convergent weight

A more elaborate example is a small decision problem. We are given a finite weighted set and a *target* $t \in N$. We ask whether there is a subset s' of S whose weight is t. This also corresponds to having a single weighted set domain and to look for its value such that its weight is t.
This problem is NP-complete. It is approximated in Integer Programming using a procedure which "trims" a list according to a given parameter. For example, the set variable

```
S ‘:: {}..{e(a,104), e(b,102), e(c,201) ,e(d,101)}
```

is approximated by the set variable

```
S’ ‘:: {}..{e(c,201) ,e(d, 101)}
```

if the parameter delta is 0.04 ($0.04 = 0.2 \div n$ where $n = \#S$).

```
:- use_module(library(conjunto)).

% Find the optimal solution to the subset-sum problem
solve(S1, Sum) :-
        getset(S),
        S1 ‘:: {}.. S,
        trim(S, S1),
        constrain_weight(S1, Sum),
        sum_weight(S1, W),
        Cost = Sum - W,
        min_max(labeling(S1), Cost).

% The set weight has to be less than Sum
constrain_weight(S1, Sum) :-
        sum_weight(S1, W),
        W #<= Sum.

% Get rid of a set of elements of the set according to a given delta
trim(S, S1) :-
        set2list(S, LS),
        trim1(LS, S1).

trim1(LS, S1) :-
        sort(2, =<, LS, [E | LSorted]),
        getdelta(D),
        testsubsumed(D, E, LSorted, S1).

testsubsumed(_, _, [], _).
```

```
testsubsumed(D, E, [F | LS], S1) :-
        el_weight(E, We),
        el_weight(F, Wf),
        ( We =< (1 - D) * Wf ->
            testsubsumed(D, F, LS, S1)
        ;
            F notin S1,
            testsubsumed(D, E, LS, S1)
        ).

% Instantiation procedure
labeling(Sub) :-
        set(Sub),!.
labeling(Sub) :-
        max_weight(Sub, X),
        ( X in Sub ; X notin Sub ),
        labeling(Sub).

% Some sample data
getset(S) :- S = {e(a,104), e(b,102), e(c,201), e(d,101), e(e,305),
                  e(f,50), e(g,70),e(h,102)}.
getdelta(0.05).
```

The approach is is the following: first create the set domain variable(s), here there is only one which is the set we want to find. We state constraints which limit the weight of the set. We apply the "trim" heuristics which removes possible elements of the set domain. And finally we define the cost term as a finite domain used in the **min_max/2** predicate. The cost term is an integer. The **conjunto.pl** library makes sure that any modification of an fd term involved with a set term is propagated on the set domain. The labeling procedure refines a set domain by selecting the element of the set domain which has the biggest weight using `max_weight(Sub, X)`, and by adding it to the lower bound of the set domain. When running the example, we get the following result:

```
[eclipse 3]: solve(S, 550).
Found a solution with cost 44
Found a solution with cost 24

S = {e(d, 101), e(e, 305), e(f, 50), e(g, 70)}
yes.
```

An interesting point is that in set based problems, the optimization criteria mainly concern the cardinality or the weight of a set term. So in practice we just need to label the set term while applying the **fd** optimization pred-

icates upon the set cardinality or the set weight. There is no need to define additional optimization predicates.

3.5.3 The ternary Steiner system of order n

A ternary Steiner system of order n is a set of $n*(n-1)\backslash 6$ triplets of distinct elements taking their values between *1* and n, such that all the pairs included in two different triplets are different.
This problem is very well dedicated to be solved using set constraints: (i) no order is required in the triplet elements and (ii) the constraint of the problem can be easily written with set constraints saying that any intersection of two set terms contains at most one element. With a finite domain approach, the list of domain variables which should be distinct requires to be given explicitely, thus the problem modelling is would be bit ad-hoc and not valid for any n.

```
:- use_module(library(conjunto)).

% Gives one solution to the ternary steiner problem.
% n has to be congruent to 1 or 3 modulo 6.

steiner(N, LS) :-
        make_nbsets(N,NB),
        make_domain(N, Domain),
        init_sets(NB, Domain, LS),
        card_all(LS, 3),
        labeling(LS, []).

labeling([], _).
labeling([S | LS], L) :-
        refine(S),
        (LS = []  ; LS = [L2 | _Rest],
        all_distincts([S | L], L2),
        labeling(LS, [S | L])).

% the labeled sets are distinct from the set to be labeled
% this constraint is a disjonction so it is useless to put it
% before the labeling as no information would be deduced anyway
all_distincts([], _).
all_distincts([S1 |L], L2) :-
        distinctsfrom(S1, L2),
        all_distincts(L, L2).

distinctsfrom(S, S1) :-
```

```
        #(S /\ S1,C),
        fd:(C #<= 1).

% creates the required number of set variables according to n
make_nbsets(N,NB) :-
        NB is N * (N-1) // 6.

% initializes the domain of the variables according to n
make_domain(N, Domain) :-
        D :: 1.. N,
        dom(D, L),
        list2set(L, Domain).

init_sets(0, _Domain, []) :- !.
init_sets(NB, Domain, Sol) :-
        NB1 is NB-1,
        init_sets(NB1, Domain, Sol1),
        S `:: {} .. Domain,
        Sol = [S | Sol1].

% constrains the cardinality of each set variable to be equal to V (=3)
card_all([], _V).
card_all([Set1|LSets], V) :-
        #(Set1, V),
        card_all(LSets, V).
```

The approach with sets is the following: first we create the number of set variables required according to the initial problem definition such that each set variable is a triplet. Then to initialize the domain of these set variables we use the fd predicates which allow to define a domain by an implicit enumeration approach 1..n. This process is cleaner than enumerating a list of integer between 1 and n. Once all the domain variables are created, we constrain their cardinality to be equal to three. Then starts the labeling procedure where all the sets are labeled one after the other. Each time one set is labeled, constraints are stated between the labeled set and the next one to be labeled. This constraint states that two sets have at most one element in common. The semantics of $\#(S \cap S\_1, C), C \leq 1$ is equivalent to a disjunction between set values. This implies that in the contraint propagation phase, no information can be deduced until one of the set is ground and some element has been added to the second one. No additional heuristics or tricks have been added to this simple example so it works well for n = 7, 9 but with the value 13 it becomes quite long. When running the example, we get the following result:

```
[eclipse 4]: steiner(7, S).
```

```
6 backtracks
0.75
S = [{1, 2, 3}, {1, 4, 5}, {1, 6, 7}, {2, 4, 6}, {2, 5, 7}, {3, 4, 7}, {3, 5, 6
yes.
```

3.6 When to use Set Variables and Constraints...

The *subset-sum* example shows that the general principle of solving problems using set domain constraints works just like finite domains:

- Stating the variables and assigning an initial set domain to them.
- Constraining the variables. In the above example the constraint is just a built-in constraint but usually one needs to define additional constraints.
- Labeling the variables, *i.e.*, assigning values to them. In the set case it would not be very efficient to select one value for a set variable for the size of a set domain is exponential in the upper bound cardinality and thus the number of backtracks could be exponential too. A second reason is that no specific information can be deduced from a failure (backtrack) whereas if (like in the refine predicate) we add one by one elements to the set till it becomes ground or some failure is detected, we benefit much more from the constraint propagation mechanism. Every domain modification activates some constraints associated to the variable (depending on the modified bound) and modifications are propagated to the other variables involved in the constraints. The search space is then reduced and either the goal succeeds or it fails. In case of failure the labeling procedure backtracks and removes the last element added to the set variable and tries to instanciate the variable by adding another element to its lower bound. In the `subset-sum` example the labeling only concerns a single set, but it can deal with a list of set terms like in the `steiner` example. Although the choice for the element to be added can be done without specific criterion like in the `steiner` example, some user defined heuristics can be embedded in the labeling procedure like in the `subset-sum` example. Then the user needs to define his own `refine` procedure.

Set constraints propose a new modelling of already solved problems or allows (like for the *subset-sum* example) to solve new problems using CLP. Therefore, one should take into account the problem semantics in order to define the initial search space as small as possible and to make a powerful use of set constraints. The objective of this library is to bring CLP to bear on graph-theorical problems like the *steiner* problem which is a hypergraph computation problem, thus leading to a better specification and solving of

problems as, packing and partitioning which find their application in many real life problems. A partial list includes: railroad crew scheduling, truck deliveries, airline crew scheduling, tanker-routing, information retrieval,time tabling problems, location problems, assembly line balancing, political districting,etc.
Sets seem adequate for problems where one is not interested in each element as a specific individual but in a collection of elements where no specific distinction is made and thus where symmetries among the element values need to be avoided (eg. steiner problem). They are also useful when heterogeneous constraints are involved in the problem like weight constraints combined with some disjointness constraints.

3.7 User-defined constraints

To define constraints based on set domains one needs to access the properties of a set term like its domain, its cardinality, its possible weight. As the set variable is a metaterm i.e. an abstract data structure, some built-in predicates allow the user to process the set variables and their domains, modify them and write new constraint predicates.

3.7.1 The abstract set data structure

A set domain variable is a metaterm. The **conjunto.pl** library defines a metaterm attribute
set{setdom:[Glb,Lub], card:C, weight:W, del_inst:Dinst, del_glb:Dglb, del_lub:Dlub, del_any:Dany}
This attribute stores information regarding the set domain, its cardinality, and weight (null if undefined) and together with four suspension lists. The attribute arguments have the following meaning:

- **setdom** The representation of the domain itself. As set domains are treated as abstract data types, the users should not access them directly, but only using built-in access and modification predicates presented hereafter.
- **card** The representation of the set cardinality. The cardinality is initialized as soon as a set domain is attached to a set variable. It is either a finite domain or an integer. It can be accessed and modified in the same way as set domains (using specific built-in predicates).
- **weight** The representation of the set weight. The weight is intialized to zero if the domain is not a weighted set domain, otherwise it is computed as soon as a weighted set domain is attached to a set variable. it can be accessed and modified in the same way as set domains (using specific built-in predicates).

- **del_inst** A suspension list that should be woken when the domain is reduced to a single set value.
- **del_glb** A suspension list that should be woken when the lower bound of the set domain is updated.
- **del_lub** a suspension list that should be woken when the upper bound of the set domain is updated.
- **del_any** a suspension list that should be woken when any reduction of the domain is inferred.

The attribute of a set domain variable can be accessed with the predicate **svar_attribute/2** or by unification in a matching clause:

```
get_attribute(_{set: Attr}, A) :- -?-> nonvar(Attr), Attr = A.
```

The attribute arguments can be accessed by macros from the ECL^{i}PS^e**structures.pl** library, if e.g. **Attr** is the attribute of a set domain variable, the del_inst list can be obtained by:

```
arg(del_inst of set, Attr, Dinst)
```

or by using a unification:

```
Attr = set{del_inst: Dinst}
```

3.7.2 Set Domain access

The domains are represented as abstract data types, and the users are not supposed to access them directly. So we provide a number of predicates to allow operations on set domains.

set_range(?Svar,?Glb,?Lub)

If *Svar* is a set domain variable, it returns the lower and upper bounds of its domain. Otherwise it fails.

glb(?Svar,?Glb)

If *Svar* is a set domain variable, it returns the lower bound of its domain. Otherwise it fails.

lub(?Svar, ?Lub)

If *Svar* is a set domain variable, it returns the upper bound of its domain. Otherwise it fails.

el_weight(++E, ?We)

If E is element of a weighted domain, it returns the weight associated to E. Otherwise it fails.

max_weight(?Svar,?E)

If *Svar* is a set variable, it returns the element of its domain which belongs to the set resulting from the difference of the upper bound and the lower bound and which has the greatest weight. If *Svar* is a ground set, it returns the element with the biggest weight. Otherwise it fails.

Two specific predicates make a link between a ground set and a list.
set2list(++S, ?L)

If S is a ground set, it returns the corresponding list. If L is also ground it checks if it is the corresponding list. If not, or if S is not ground, it fails.

list2set(++L, ?S)

If L is a ground list, it returns the corresponding set. If S is also ground it checks if it is the corresponding set. If not, or if L is not ground, it fails.

3.7.3 Set variable modification

A specific predicate operate on the set domain *variables*. When a set domain is reduced, some suspension lists have to be scheduled and woken depending on the bound modified.
NOTE: Their are 4 suspension lists in the **conjunto.pl** library, which are woken precisely when the event associated with each list occurs. For example, if the lower bound of a set variable is modified, two suspension lists will be woken: the one associated to a **glb** modification and the one associated to **any** modification. This allows user-defined constraints to be handled efficiently.
modify_bound(Ind, ?S, ++Newbound)

Ind is a flag which should take the value **lub** or **glb**, otherwise it fails ! If S is a ground set, it succeeds if we have *Newbound* equal to S. If S is a set variable, its new lower or upper bound will be updated. For monotonicity reasons, domains can only get reduced. So a new upper bound has to be contained in the old one and a new lower bound has to contain the old one. Otherwise it fails.

3.8 Example of defining a new constraint

The following example demonstrates how to create a new set constraint. To show that set inclusion is not restricted to ground herbrand terms we can take the following constraint which defines lattice inclusion over lattice domains:

```
S_1 incl S
```

Assuming that S and S_1 are specific set variables of the form

```
S `:: {} ..{{a,b,c},{d,e,f}}, ..., S_1 `:: {} ..{{c},{d,f},{g,f}}
```

we would like to define such a predicate that will be woken as soon as one or both set variables' domains are updated in such a way that would require updating the other variable's domain by propagating the constraint. This constraint definition also shows that if one wants to iterate over a ground set (set of known elements) the transformation to a list is convenient. In fact iterations do not suit sets and benefit much more from a list structure. We define the predicate `incl(S,S1)` which corresponds to this constraint:

```
:- use_module(library(conjunto)).
incl(S,S1) :-
          set(S),set(S1),
          !,
          check_incl(S, S1).
incl(S, S1) :-
          set(S),
          set_range(S1, Glb1, Lub1),
          !,
          check_incl(S, Lub1),
          S + Glb1 `= S1NewGlb,
          modify_bound(glb, S1, S1NewGlb).
incl(S, S1) :-
          set(S1),
          set_range(S, Glb, Lub),
          !,
          check_incl(Glb, S1),
          large_inter(S1, Lub, SNewLub),
          modify_bound(lub, S, SNewLub).
incl(S,S1) :-
          set_range(S, Glb, Lub),
          set_range(S1, Glb1, Lub1),
          check_incl(Glb, Lub1),
          Glb \/ Glb1 `= S1NewGlb,
          large_inter(Lub, Lub1, SNewLub),
```

```
          modify_bound(glb, S1, S1NewGlb),
          modify_bound(lub, S, SNewLub),
          ( (set(S) ; set(S1)) ->
               true
         ;
               make_suspension(incl(S, S1),2, Susp),
               insert_suspension([S,S1], Susp, del_any of set, set)
          ),
          wake.

large_inter(Lub, Lub1, NewLub) :-
          set2list(Lub, Llub),
          set2list(Lub1, Llub1),
          largeinter(Llub, Llub1, LNewLub),
          list2set(LNewLub, NewLub).

largeinter([], _, []).
largeinter([S | List_set], Lub1, Snew) :-
          largeinter(List_set, Lub1, Snew1),
          ( contained(S, Lub1) ->
                Snew = [S | Snew1]
          ;
                Snew = Snew1
          ).

check_incl({}, _S) :-!.
check_incl(Glb, Lub1) :-
          set2list(Glb, Lsets),
          set2list(Lub1, Lsets1),
          all_union(Lsets, Union),
          all_union(Lsets1, Union1),
          Union `< Union1,!,
          checkincl(Lsets,Lsets1).
checkincl([], _Lsets1).
checkincl([S | Lsets],Lsets1):-
          contained(S, Lsets1),
          checkincl(Lsets,Lsets1).

contained(_S, []) :- fail,!.
contained(S, [Ss | Lsets1]) :-
          (S `< Ss ->
                true
          ;
                contained(S, Lsets1)
```

```
).
```

The execution of this constraint is dynamic, *i.e.*, the predicate `incl/2` is called and woken following the following steps:

- We check if the two set variables are ground `set`. If so we just check deterministically if the first one is included (lattice inclusion) in the second one `check_incl`. This predicate checks that any element of a ground set (which is a set itself in this case) is a subset of at least one element of the second set. If not it fails.

- We check if the first set is ground and the second is a set domain variable. If so, `check_incl` is called over the first ground set and the upper bound of the second set. If it succeeds then the lower bound of the set variable might not be consistent any more, we compute the new lower bound (*i.e.*, adding elements from the ground set in it (by using the union predicate) and we modify the bound `modify_bound`. This predicate also wakes all concerned suspension lists and instantiates the set variable if its domain is reduced to a single set (upper bound = lower bound).

- We check if the second set is ground and the first one is a set variable. If so, `check_incl` is called over the lower bound of the first set and the second ground set. If it succeeds then the upper bound of the set variable might not be consistent any more. The new upper bound is computed by intersecting the first set with the upper bound of the set variable in the lattice acceptation `large_inter` and is updated `modify_bound`.

- we check if both set variables are domain variables. If so the lower bound of the first set should be included in the lattice sense in the upper bound of the second one `check/incl`. If it succeeds, then if the lower bound the second set is no more consistent we compute the new one by making the union with first sec lower bound. In the same way, the upper bound of the first set might not be consistent any more. If so, we compute the new one by intersecting (in the lattice acceptation) the both upper bounds to compute the new upper bound of the first set `large_inter`. The upper bound of the first set variable is updated as well as the lower bound of the second set `modify_bound`.

- After checking all these updates, we test if the constraint implies an instanciation of one of the two sets. If this is not the case, we have to suspend the predicate so that it is woken as soon as any bound of either set domain is changed. The predicate `make_suspension/3` can be used for any ECL^{i}PS^e module based on a meta-term structure. It creates a suspension, and then the predicate `insert_suspension/4`,

puts this suspension into the appropriate lists (woken when any bound is updated) of both set variables.

- the last action `wake` triggers the execution of all goals that are waiting for the updates we have made. These goals should be woken after inserting the new suspension, otherwise the new updates coming from these woken goals won't be propagated on this constraint !

3.9 Set Domain output

The library **conjunto.pl** contains output macros which print a set variable as well as a ground set respectively as an interval of sets or a set. The **setdom** attribute of a set domain variable (metaterm) is printed in the simplified form of just the *glb..lub* interval, e.g.

```
[eclipse 2]: S `:: {}..{a,v,c}, svar_attribute(S,A), A = set{setdom : D}.

S = S{{} .. {a, c, v}}
A = {} .. {a, c, v}
D = [{}, {a, c, v}]
yes.
```

3.10 Debugger

The ECL^{i}PS^e debugger which supports debugging and tracing of finite domain programs in various ways, can just be used the same way for set domain programs. No specific set domain debugger has been implemented for this release.

Chapter 4

Porting to Standalone Eplex

Since ECL^{i}PS^e version 5.7, standalone eplex have become the standard eplex, loaded with `lib(eplex)`. The previous `lib(eplex)`, which loads eplex with the range bounds keeper and the IC variant have now been phased out, so users of these old variants must now move to using standalone eplex. There are some differences at the source level between standalone and the older non-standalone eplex. This chapter outlines these differences to help users to port their existing code to standalone eplex.

4.1 Differences between Standalone Eplex and Older Non-Standalone Eplex

The main difference between the standalone eplex and the non-standalone eplex is that the standalone version does not use an ECL^{i}PS^e 'bounds keeper' like lib(ic) or lib(range) to provide the ranges for the problem variables. Instead, ranges for variables are treated like another type of eplex constraint, i.e., they are posted to an eplex instance, and are stored with the external solver state.

In the non-standalone eplex, the bounds of *all* problem variables are transferred from the bounds keeper to the external solver each time the solver is invoked, regardless of if the bounds for the variables have changed or not since the last invocation. This can become very expensive if a problem has many variables. With the standalone eplex, this overhead is avoided as the external solver bounds for variables are only updated if they are explicitly changed. A possible inconvenience is that for hybrid programming, where eplex is being used with another ECL^{i}PS^e solver, any bound updates due to inferences made by the ECL^{i}PS^e solver are not automatically transferred to the external solver. This can be an advantage in that it leaves the programmer the freedom of when and how these bound changes should be transferred to the external solver.

The main user visible differences with the non-standalone eplex are:

- Bounds constraints intended for an eplex instance should be posted to that instance, e.g.

```
[eclipse 3]: eplex_instance(instance).
...

[eclipse 4]: instance: eplex_solver_setup(min(X)),
        instance: (X:: 0.0..10.0), instance: eplex_solve(C).

X = X{0.0 .. 10.0 @ 0.0}
C = 0.0
Yes (0.00s cpu)
```

  The ::/2 ($::/2) constraints are treated like other eplex constraints, that is, the bounds for the variables are specific to their eplex instance. Other eplex instances (and indeed any other bounds-keeping solver) can have different and even incompatible bounds set for the same variable. Also, if the variable(s) do not already occur in the eplex instance, they will be added. Both of these are different from the non-standalone eplex, where bound constraints were treated separately from the eplex constraints.

  Like other eplex constraints, inconsistency within the same eplex instance will lead to failure, i.e. if the upper bound of a variable becomes smaller than its lower bound, this will result in failure, either immediately or when the solver is invoked.

  One potential problem is that with the non-standalone eplex, the bound keeper's `::/2` was re-exported through the eplex module (but not through the eplex instances). One was able to write `eplex: (X :: 1.0..2.0)` and affect the bounds of the variable for *all instances*, even though this was not posting a constraint to any eplex instance. With the standalone eplex, the same code, `eplex: (X :: 1.0..2.0)` has different semantics and *is* a constraint for the eplex instance `eplex` only.

  A variable never becomes ground as a result of an eplex instance bound constraint, even when the upper and lower bounds are identical.

  Posting eplex arithmetic constraints involving one variable is the same as posting a bounds constraint. Unlike the non-standalone eplex, the variable will be added to the eplex instance even if it does not occur in any other constraints.

  No propagation of the bounds is performed at the ECLiPSe level: the bounds are simply passed on to the external solver. In general, the

external solver also does not do any bounds propagation that may be implied by the other constraints in the eplex instance.

Note that the generic **get_var_bounds/3** and **set_var_bounds/3** applies to *all* the eplex instances/solver states. If set_var_bounds/3 is called, then failure will occur if the bounds are inconsistent between the eplex instances.

- **integers/1** only indicates that a variable should be treated as an integer by the external solver in the eplex instance, but does not impose the integer type on the variable. In addition, the type of the solution returned for a variable is determined only by if it was in an `integers/1` declaration for the eplex instance. (In non-standalone eplex, the type is determined by the type given the variable by the bounds keeper)

- If a bounds keeper like lib(ic) is loaded, then any bounds constraints posted to this solver are *not* automatically visible to the eplex instances. Instead, the bounds can be transferred explicitly by the user (e.g. by calling the eplex instance bounds constraints when the bounds in the solver changes). To allow for more compatibility with the other versions of eplex, the `sync_bounds(yes)` option can be specified during solver setup (using **eplex_solver_setup/4**). This will 'synchronise' the bounds of all problem variables when the external solver is invoked, by calling `get_var_bounds/3` for all problem variables. Note that it is the generic get bounds handler that is called.

- When a demon solver is invoked, the update to the objective variable is via an update to its bounds. In the standalone eplex, this is done by calling the generic `set_var_bounds/3`. However, if there are no bounds on this variable, the update will be lost. A warning is given during the setup of the demon if the objective variable has no bounds.

  One possible solution is to add the objective variable to the problem (e.g. by giving it bounds for the eplex instance). However, this can induce extra 'self-waking' that needlessly invokes the solver (e.g. if the bounds trigger option is used). Another solution is to add bounds to the variable via some other bounds keeper, e.g. `lib(ic)`. Note that it is always possible to retrieve the objective value via the `objective` option of **eplex_get/2**.

- When a constraint is posted to an eplex instance after solver setup, that constraint is immediately added to the external solver, rather than only 'collected' by the external solver when it is invoked.

- The solver setup predicates have been simplified in that the suspension priority is no longer specified via an argument, so these predicates have one less argument: **eplex_solver_setup/4**, **lp_demon_setup/5**

Instead, the priority can be specified as an option, if required. The older predicates with the priority argument are still available for compatibility purposes.

- **eplex_get/2** and **lp_get/3** now has an extra option: `standalone` which returns the value `yes` for standalone eplex and `no` otherwise.

- The order in which variables are passed to the external solver has changed. Also, with standalone eplex there may be more variables in the problem. This should not be visible to the user, except when examining a problem written out by the external solver. This makes it difficult to compare problems generated using standalone eplex and non-standalone eplex. Using the `use_var_names(yes)` options in setup should make this somewhat easier as the variables would have the same names.

Index

Bibliography