ECL^{i}PS^e : A Platform for Constraint Logic Programming

Mark Wallace, Stefano Novello, Joachim Schimpf
Contact address: IC-Parc,
William Penney Laboratory, Imperial College, LONDON SW7 2AZ.
email: mgw@doc.ic.ac.uk

August 1997

Abstract

This paper introduces the Constraint Logic Programming (CLP) platform ECL^{i}PS^e . ECL^{i}PS^e is designed to be more than an implementation of CLP: it also supports mathematical programming and stochastic programming techniques. The crucial advantage of ECL^{i}PS^e is that it enables the programmer to use a combination of algorithms appropriate to the application at hand. This benefit results from the ECL^{i}PS^e facility to support fine-grained hybridisation.

ECL^{i}PS^e is designed for solving difficult "combinatorial" industrial problems in the areas of planning, scheduling and resource allocation. The platform offers a conceptual modelling language for specifying the problem clearly and simply, in a way that is neutral as to the algorithm which will be used to solve it. Based on the conceptual model it is easy to construct alternative design models, also expressed in ECL^{i}PS^e . A design model is a runnable program, whose execution in ECL^{i}PS^e employs a specific combination of algorithms. Thus the platform supports experimentation with different hybrid algorithms.

Technically the different classes of algorithms mentioned above have two aspects: constraint handling, and search. Various different constraint handling facilities are available as ECL^{i}PS^e libraries. These include finite domain propagation, interval propagation and linear constraint solving. In ECL^{i}PS^e the same constraint can be treated concurrently by several different handlers.

With regard to search behaviour, CLP and also mathematical programming typically impose new constraints at lower levels in the search tree. By contrast, stochastic techniques search for good solutions by locally repairing an original solution, and repeating the process again and again. ECL^{i}PS^e supports both kinds of search, and allows them to be combined into hybrid search techniques.

1 Introduction: The ECL^{i}PS^e Philosophy

The first generation of constraint programming languages focussed on a single technique: constraint propagation, as described in section 4 of [Wal97]. Whilst constraint propagation has proved itself on a variety of applications, it cannot alone suffice to efficiently produce solutions for typical practical industrial problems.

Over the years Operations Researchers have designed highly efficient algorithms for several classes of problems, such as set partitioning, matching, knapsack, and network flow problems, using techniques based on Mixed Integer Programming (MIP). More recently stochastic techniques, such as Simulated Annealing, have achieved striking results on optimisation problems such as the travelling salesman problem.[1]

ECL^{i}PS^e is designed to take advantage of all these results, by supporting industrial scale MIP functionality, and stochastic techniques, as well as constraint propagation and solving.

[1] The travelling salesman problem is to find the shortest route which starts at a certain point, visits a given set of destinations (customers), and returns to the starting point at the end.

More importantly, real industrial problems seldom fit into a specific class: the pure travelling salesman problem rarely comes up in real life because there are typically many salesmen available to cover the different customers, certain customers can only be visited at certain times of day, also roads are busier at certain times of day so the journey time may vary with the time of day, and anyway the poor salesmen need some time to rest - they can't usually complete their circuits before lunchtime! These "side constraints" may belong to another problem class - such as the class of set covering problems, or scheduling problems.

Industrial problems typically have constraints that belong to different problem classes - they are in a sense "hybrid". Accordingly it is not only necessary to offer a wide choice of algorithms for solving such problems, but also the facility to mix and match the algorithms, i.e. to build hybrid algorithms.

ECL^{i}PS^e is designed to support the fast development of specific hybrid algorithms tuned to the problem at hand. It is not assumed that the first algorithm implemented by the application developer is guaranteed to be the best one: rather ECL^{i}PS^e provides a platform supporting experimentation with different hybrid algorithms until an appropriate one is found which suits the particular features of the application.

In the next section we shall explore ECL^{i}PS^e as a problem modelling language. We distinguish two kinds of model: the *conceptual* model, which captures the problem specification, and the *design* model, which is tuned for efficient solving on a computer. ECL^{i}PS^e is designed to support both kinds of models, and the mapping between them.

In the following two sections we shall examine the ECL^{i}PS^e facilities for handling constraints. In [Wal97] we encountered different kinds of constraints - *primitive* constraints, *propagation* constraints and *constraint agents*. ECL^{i}PS^e supports various classes of built-in constraints, both *primitive* constraints and *propagation* constraints. ECL^{i}PS^e also allows complex constraints and constraint behaviours to be constructed from the built-in classes, thus supporting *constraint agents*.

After constraint handling we return to the second major aspect of problem solving: the search for solutions.

We will separate this discussion into two subsections. The first is concerned with *constructive* search, and the second with *repair-based* search. Constructive search explores the consequences of making choices for decision variables one-at-a-time. Each choice reduces the set of viable choices for the remaining decisions. By contrast repair-based search explores the consequences, not of making decisions, but of *changing* them. In this case the new choice is typically compared with the previous one, in the context of other suggested choices for the other decision variables. Initially it is not expected that the suggested choices are necessarily consistent with the constraints. The idea of changing the choices is to reduce the number of constraint violations until all the constraints are finally satisfied.

Finally there is a brief section on the ECL^{i}PS^e system, its external communication facilities, embeddability, documentation and how to obtain the system.

2 ECL^{i}PS^e as a Modelling Language

2.1 Overview of ECL^{i}PS^e as a Modelling Language

ECL^{i}PS^e is tailored for solving combinatorial problems. Such problems are characterised by a set of decisions which have to be made (where each decision has a set of alternative choices) and a set of constraints between the decisions (so a certain choice for one decision may preclude, or entail, certain choices for other decisions).

In ECL^{i}PS^e each decision is modelled by a variable, and each choice by a possible value for that variable. The constraints are modelled by relations between the variables. As an example consider the map colouring program, with four countries to colour.

This program was also used to illustrate constraint logic programming in [Wal97]. ECL^{i}PS^e is a constraint logic programming language, and it uses the same syntax as Prolog. Hopefully this syntax will already be familiar to many readers. At the same time, we also hope that any readers who have suffered from the limitations of Prolog will not conclude that ECL^{i}PS^e therefore suffers from the same limitations!

The problem involves four decisions, one for each country. These are modelled by the variables A, B, C and D. *Countries* is just a name for the list of four variables. Each

```
:- lib(apply_macros).

coloured(Countries) :-
    Countries = [A,B,C,D],
    applist(value,Countries),
    ne(A,B), ne(A,C), ne(A,D), ne(B,C), ne(B,D), ne(C,D).

value(red).
value(green).
value(blue).
```

Figure 1: A Generic Logic Program for Map Colouring

decision variable, in this problem, has the same set of choices, modelled as possible values for the variables (*red*, *green* and *blue*). There are six constraints, each of which is modelled by the same relation (*ne* meaning *not equal to*).

The first command `:- lib(apply_macros).` loads an ECL^{i}PS^e library. Much of the functionality of ECL^{i}PS^e is held in different libraries, some of which will be introduced in the next section. The library *apply_macros* holds the definition of the *applist* predicate, which applies a predicate to each element of a list. *applist(value,Countries)* is equivalent to *value(A), value(B), value(C), value(D)*.

2.2 Why Logic Programming

The requirements on ECL^{i}PS^e are of two kinds: to enable such problems to be modelled simply and naturally; and to enable the resulting problem model to be solved efficiently. The separation of modelling and solving is supported in ECL^{i}PS^e by distinguishing the conceptual model, expressed as a "pure" logical ECL^{i}PS^e program, from the design model, which is constructed from the conceptual model by adding control to the ECL^{i}PS^e program.

This combination of requirements is difficult to satisfy - perhaps impossible if a completely general modelling language is required, suitable for every kind of application. However the applications for which ECL^{i}PS^e is designed are decision support applications involving combinatorial problems.

Logic programming is peculiarly apt for modelling problems of this kind for two reasons.

- It is based on relations
- It supports logical variables

Since every combinatorial problem is naturally modelled as a set of variables and a set of constraints (i.e. relations) on those variables, the facilities of logic programming precisely match the requirements for modelling combinatorial problems.

Every predicate in a logic program defines a relation, either explicitly as a set of facts, or implicitly in terms of rules. We can recall the example from the [Wal97]. The predicate *meat* was defined by two facts:

```
meat(beef,5).
meat(pork,7).
```

whilst the predicate *main* (meaning "main course") was defined by two rules:

```
main(M,I) :- meat(M,I).
main(M,I) :- fish(M,I).
```

Variables in logic programming are logical variables. Thus it is entirely natural to initialise the problem variables (for example by writing $Countries = [A, B, C, D]$) and then to constrain them (for example by writing $ne(A, B)$ and so on).

We briefly compare ECL^{i}PS^e as a modelling language with formal specification languages, mathematical modelling languages, mainstream programming languages and object oriented languages.

2.2.1 Formal Specification Languages

Formal specification language are designed for formality, but not for execution. Consequently they include constructs, such as universal quantification, which are precisely defined but are not constructive. In other words there are constructs which cannot be mapped onto any (practical) algorithm.

Luckily the class of problems for which ECL^{i}PS^e is designed have a finite set of decision variables each of which admits only finitely many alternatives. Consequently it is only necessary to support a restricted form of logic[2] which is easier to understand and easier to implement. The nearest thing ECL^{i}PS^e offers to universal quantification is iteration over finite sets, as for example the goal *applist(value,Countries)* in figure 1.

The restricted logic of ECL^{i}PS^e has a benefit that the mapping from the conceptual model of the problem to the design model is an extension of the conceptual model rather than a rewriting. This means that when problem requirements change it is natural to capture this change in the conceptual model, and then carry them through to the design model. The result is that during application development the conceptual model and the design model remain in step. This avoids many of the pitfalls which await developers working on applications whose specifications are changing even during application development!

2.2.2 Mathematical Modelling Languages

There already exists a class of modelling languages designed for combinatorial problems. These are the mathematical modelling languages typically used as input to mixed integer programming (MIP) packages. We further discuss MIP, and how to use it through ECL^{i}PS^e , in section 3.4 below.

Although the syntax is different, mathematical modelling languages share many of the features of logic programming. They support logical variables, and constraints. They support numerical constraints which, though not supported in traditional logic programs, are supported by constraint logic programs as we shall see in the following section. They support named constraints, which is achieved in constraint logic programming by introducing a predicate name, eg `precede(T1,T2) :- T1 >= T2.`

There are two facilities in constraint logic programming which are not available in mathematical modelling languages. The main one is quite simple: in constraint logic programs it is possible to define a constraint which involves a disjunction. Mathematical programming cannot handle disjunction directly. The second difference is that logic programming allows new constraints to be defined in terms of existing ones, even recursively. In mathematical programming the model is essentially flat, which not only complicates the model but also reduces reuseability within an application and across applications.

To illustrate the advantage of handling disjunction in the modelling language, we take a toy example and present two models: a mathematical programming model and a constraint logic programming model.

Consider the constraint that two tasks sharing a single resource cannot be done at the same time. The constraint involves six variables: the start times S_1, S_2, end times E_1, E_2 and resources R_1, R_2 of the two tasks. The specification of this constraint is as follows:

> *either* the two tasks use distinct resource (R_1 *ne* R_2) *or* task$_1$ ends before task$_2$ starts ($E_1 \leq S_2$) *or else* task$_2$ ends before task$_1$ starts ($E_2 \leq S_1$).

First we shall show how it can expressed as a mathematical model without disjunctions. For this purpose it must be encoded using numerical equations and inequalities, together with integer constraints.

The disjunctions can be captured by introducing three *0/1* variables, B_{r1}, B_{r2}, and B_t, and using some large constant, say 100000, larger than any possible values for any of the six variables. Now we can express the constraint in terms of numerical inequalities as follows:

$$R_1 + 100000 * B_{r1} + 100000 * B_{r2} \geq R_2 + 1$$
$$R_2 + 100000 * B_{r1} + 100000 * (1 - B_{r2}) \geq R_1 + 1$$
$$S_1 + 100000 * (1 - B_{r1}) + 100000 * B_t \geq E_2$$
$$S_2 + 100000 * (1 - B_{r1}) + 100000 * (1 - B_t) \geq E_1$$

[2] technically called Horn clauses

If $B_{r1} = 0$ then the two tasks use different resources. In this case, if also $B_{r2} = 0$ then $R_1 \geq R_2 + 1$, otherwise $B_{r2} = 1$ and $R_2 \geq R_1 + 1$. It is an exercise for the reader to prove that if $B_{r1} = 0$ then the tasks can overlap. Otherwise, if $B_{r1} = 1$, then $B_t = 0$ entails $S_1 \geq E_2$ and $B_t = 1$ entails $S_2 \geq E_1$.

In ECL^{i}PS^e this constraint can be modelled directly in logic, as illustrated in figure 2.

```
taskResource(S1,E1,R1,S2,E2,R2) :-
        ne(R1,R2).
taskResource(S1,E1,R1,S2,E2,R2) :-
        R1=R2, S1 >= E2.
taskResource(S1,E1,R1,S2,E2,R2) :-
        R1=R2, S2 >= E1.
```

Figure 2: Specifying a Resource Contention Constraint in ECL^{i}PS^e

We note that the ECL^{i}PS^e model is a *conceptual* model, whilst the mathematical model is a *design* model. The point here is that in ECL^{i}PS^e both models can be expressed, whilst mathematical modelling can only express a design model. Indeed we shall show in section 4.1 below a design model written in ECL^{i}PS^e that is very close to the conceptual model.

Another ECL^{i}PS^e design model, which is also close to the conceptual model, is handled in ECL^{i}PS^e by an automatic translator which builds the MIP model and passes it to the MIP solver of ECL^{i}PS^e . This translator is described in [RWH97] which is available from the IC-Parc home page (whose URL is given in section 6 below).

Whilst the above example shows that such complex constraints can be expressed in terms of numerical inequalities, as required for MIP, the encoding is awkward and difficult to debug. It becomes increasingly difficult as the constraints become more complex (eg the current example immediately becomes harder still if the resources have a finite capacity greater than one).

Notice, finally, that the mathematical model requires resources to be identified by numbers, whilst the constraint logic programming model imposes no such restriction as we shall show in section 4 below.

2.2.3 Mainstream Programming Languages

Naturally the implemented solution to an industrial problem must be delivered into the industrial computing environment. It is sometimes argued that this is only possible if the solution is implemented in a mainstream programming language such as `C`, `C++` or even `Java`. There are two arguments supporting this view, firstly that of embeddability (it is easier and more efficient to pass data and control between modules written in the same programming language), and secondly that of system support (mainstream language programmers are much easier to find and replace than specialist programmers).

Whilst this argument only supports a mainstream programming language being used for implementation, and not conceptual modelling, it has consequence for the modelling language as well on the assumption, which we discussed above, that the conceptual model should be close to the design model. Thus if the design model is encoded in a mainstream programming language, then either the conceptual model must be compromised becoming more like a design model, or the gap between the conceptual model and design model grows very wide.

Sadly the attempt to tackle combinatorial problems with mainstream programming languages has too often foundered because the implemented solution has proved not to solve the actual industrial requirement (often because requirements change during application development). The solution cannot then be modified to meet the actual, or new, requirements within a reasonable cost and timescale.

Given that the core combinatorial optimisation problem is best solved by a specialised programming platform (either mathematical or constraint-based), the problem of embedding has to be solved.

One approach is to embed constraint solving in a mainstream programming language. As we shall see in section 5 below, search and constraint handling are closely interdependent. Even if the search is encoded in a mainstream programming language, the programmer is required to understand in detail not only the data structures used by the constraint handlers, but their operational behaviour.

In practice packages providing an embedding of constraints in mainstream programming languages also encapsulate search within the package. The application developer is required to control the search. To avoid any mismatch between the host programming language and search control within the package, a popular approach is to implement the package as a library of the host programming language.

The result is that the separation of conceptual modelling and design modelling is given up, in favour of staying within the confines of the expressive capabilities of the host programming language. This approach not only requires specialist programmers to develop and support the application, but it also sacrifices the modelling advantages of mathematical and constraint logic programming.

In fact the problem of embedding has been overcome, though first generation constraint logic programming languages were deficient in this area. ECL^{i}PS^e is fully embeddable in `C` and `C++`, and indeed uses an external solver, written in `C` to handle linear constraints, since the runtime cost of such an interface is perfectly acceptable even for a tightly integrated component such as a constraint handler!

2.2.4 Object Oriented Languages

ECL^{i}PS^e supports object-orientation through two distinct features, *modules* and *structures*. Modules support *behavioural* object orientation, and structures support *structural* object orientation.

Because of the nature of combinatorial problems, the only requirement for behavioural object orientation is in the constraint handlers. The implementation of each constraints library is hidden inside a module, and access to the internal data structures is only through predicates exported from the module.

The remaining objects that can occur in an ECL^{i}PS^e model have attributes but no behaviour, and so they require only structural object orientation.

In our first example we modelled a map colouring problem using only variables and constraints. It can be argued, however, that for more complex applications, the conceptual model can benefit from a notion of object, into which variables can be built. For example in modelling a resource scheduling problem the notion of a *task* with certain attributes is useful. A task might have an *identifier*, a *start time*, and *end time* and a *duration*.

After declaring structures for *tasks* and *times*, as in figure 3, the programmer can access any of their attributes independently.

```
[eclipse 1]: lib(structures).
*       structures loaded

[eclipse 2]: define_struct( task(id, start, end, duration)),
             define_struct( time(hour, minute)).
*       yes.

[eclipse 3]:   T=task with [id:a,duration:10].
*       T = task(a, _, _, 10)
*       yes.

[eclipse 4]:  T1=task with [id:a3,start:S3,end:(time with hour:H3)],
              T2=task with [id:a4,start:S3,end:(time with hour:H4)],
              H3>H4.
*       T1 = task(a3, S3, time(H3, _), _)
*       T2 = task(a4, S3, time(H4, _), _)
*       yes.
```

Figure 3: Defining a Task Structure

Each ECL^{i}PS^e prompt (eg `[eclipse 1]:`) is followed by a user query (eg `lib(structures).`). In the rest of the article, "query N" always refers to the query which is preceded by the prompt `[eclipse N]:`.

The programmer enters `lib(structures).` to which the system responds `structures loaded`. I have added a star to the beginning of each line showing a system response.

Query 2 defines the attributes for objects in the classes *task* and *time*. Query 3 shows how the user can equate a variable with a structured object (i.e. the variable is instantiated to the structure). ECL^{i}PS^e automatically constructs unknown values (written `_`) for the unspecified attributes.

Query 4 illustrates something of the expressive power needed in a constraint programming language which supports objects. Not only do the objects $T1$ and $T2$ share an attribute value - this is a shared subobject - but they also have non-shared subobjects whose attributes are connected by a constraint. Such a constraint, between distinct objects, is typically not expressible within the traditional object-oriented framework.

2.3 The Conceptual Model and the Design Model

The main benefit of constraint logic programming over other platforms for solving combinatorial problems is in the closeness between the conceptual model and the design model. ECL^{i}PS^e takes full advantage of this by offering facilities to choose different annotations of the same conceptual model to achieve design models which, whilst syntactically similar, can have radically different behaviour.

2.3.1 Map Colouring

Let us start by mapping the conceptual model for the map colouring example illustrated in figure 1 into a design model which uses the finite domain constraint handler of ECL^{i}PS^e .

The design model is encoded as shown in figure 4.

```
:- lib(fd).

coloured(Countries) :-
    Countries=[A,B,C,D],
    Countries :: [red,green,blue],
    ne(A,B), ne(A,C), ne(A,D), ne(B,C), ne(B,D), ne(C,D),
    labeling(Countries).

ne(X,Y) :- X##Y.
```

Figure 4: A Finite Domain CLP Program for Map Colouring

The design model extends the conceptual model in four ways.

1. The ECL^{i}PS^e finite domain library is loaded (using *:- lib(fd)*).
2. An explicit finite domain is associated with each decision variable (using *Countries :: [red, green, blue]*).
3. The finite domain built-in disequality constraint is used to implement the *ne* constraint (using *ne(X,Y) :- X## Y*). `##` is a special syntax for disequality used by the finite domain constraint solver.
4. This program includes a search algorithm, invoked by the goal *labeling(Countries)*. As we shall see later, this predicate tries choosing, for each of the variables A, B, C and D in turn, a value from its domain. It succeeds when a combination of values has been found that satisfies the constraints.

Naturally this is a toy example, and it is not always so easy to turn a conceptual model, such as the ECL^{i}PS^e program in figure 1, into a design model, such as the program in figure 4. Nevertheless constraint logic programming, and in particular ECL^{i}PS^e , have made a lot of progress in achieving a close relationship between the conceptual model and the design model. The different components of the ECL^{i}PS^e system all support the separate development of a clear, correct conceptual model, and an efficient design model, and they also support the mapping between the two.

2.3.2 Having Enough Change in Your Pocket

Let us now take a more interesting problem, which has been set as a recent challenge within the MIP community. The problem is apparently rather simple: what is the minimum number of coins a purchaser needs in their pocket in order to be able to buy any one item costing less than one pound, and guarantee to be able to pay the exact amount?

The problem involves only six decision variables, one for the number of coins of each denomination held in the pocket (the denominations are 1,2,5,10,20,50).

The conceptual model for this problem is shown in figure 5.

```
:- lib(apply_macros).

solve(PocketCoins,Min) :-
        PocketCoins=[P,Tw,Fv,Te,Twe,Ff],                    %1
        applist(range(0,99),[Min|PocketCoins]),             %2
        Min = P+Tw+Fv+Te+Twe+Ff,                            %3
        fromto(1,99,genc(PocketCoins)),                    %4
        minimize(Min).                                      %5

genc(PocketCoins,Total) :-
        Coins=[P1,Tw1,Fv1,Te1,Twe1,Ff1],                    %6
        applist(range(0,99),Coins),                         %7
        Total = P1+2*Tw1+5*Fv1+10*Te1+20*Twe1+50*Ff1,       %8
        maplist( ´<=´,Coins,PocketCoins).                   %9
```

Figure 5: Conceptual Model for the Coins Problem

The lines are numbered, using the syntax `%N`, as `%` is a comment symbol in ECL^{i}PS^e . We describe this program line by line.

1. The variable *PocketCoins* is just a shorthand for the list of six variables, *[P, Tw, Fv, Te, Twe, Ff]* which denote the number of coins of each denomination held in the pocket.
2. *[A,B,C]* is a list, but ECL^{i}PS^e allows lists to be written in an alternative syntax *[Head | Tail]*. Thus *[Min | PocketCoins]* is simply another way of writing the list of seven variables, *[Min, P, Tw, Fv, Te, Twe, Ff]*. The command *applist(range(0,99), [Min | PocketCoins])* associates a range (between 0 and 99) with each of the variables.
3. *Min* is the total number of coins in the pocket, as enforced by the equation $Min = P+Tw+Fv+Te+Twe+Ff$.
4. To ensure that these coins are enough to make up any total between 1 and 99, we now impose 99 further constraints, one for each total. *genc(PocketCoins,Total)* is called for each value of *Total* between 1 and 99.
5. *minimize(Min)* simply specifies that the best feasible solution to the problem is one which minimises the value of the variable *Min*.
6. *genc(PocketCoins,Total)* initialises another set of coins *[P1, Tw1, Fv1, Te1, Twe1, Ff1]* needed to make up the total *Total*.
7. This set of coins is also initialised to range between 0 and 99
8. Their total value is constrained to be equal to *Total*. This constraint is enforced by the equation $Total = P1 + 2*Tw1 + 5*Fv1 + 10*Te1 + 20*Twe1 + 50*Ff1$.

9. Finally the constraint that the required coins of each denomination must be less than, or equal to, the number of coins of that denomination in the pocket, is enforced by the constraints: *P1* `<=` *P*, *Tw1* `<=` *Tw*, *Fv1* `<=` *Fv*, *Te1* `<=` *Te*, *Twe1* `<=` *Twe*, *Ff1* `<=` *Ff*.

   These constraints are generated by the single command *maplist(* `<=`*, Coins, PocketCoins)*.

Let's start by trying mixed integer programming on this problem. To do this we add *integer* declarations for each of the integer variables, and change the constraints to use the syntax recognised by the (external) MIP solver accessed via the ECL^{i}PS^e library *eplex*. For equations we use the syntax `$=`, and for inequalities we use `$>=`. The design model is shown in figure 6.

```
:- lib(apply_macros).
:- lib(eplex).

solve(PocketCoins,Cost) :-
        PocketCoins=[P,Tw,Fv,Te,Twe,Ff],
        applist(range(0,99),[Min|PocketCoins]),
        Min $= P+Tw+Fv+Te+Twe+Ff,
        fromto(1,99,genc(PocketCoins)),
        optimize(min(Min),Cost).

genc(PocketCoins,Total) :-
        Coins=[P1,Tw1,Fv1,Te1,Twe1,Ff1],
        applist(range(0,99),Coins),
        Total $= P1+2*Tw1+5*Fv1+10*Te1+20*Twe1+50*Ff1,
        maplist( ´$=<´,Coins,PocketCoins).

range(Min,Max,Var) :-
        integers(Var),
        Var $>= Min,
        Var $=< Max.
```

Figure 6: Conceptual Model for the Coins Problem

This program passes all the `$=` and `$>=` constraints to the *CPLEX* mixed integer programming package [CPL93], and invokes the CPLEX branch and bound solver, to minimise the value of the variable *Min*. This minimum is placed in the variable *Cost*.

As such this model can only solve the problem of producing the exact change up to 59 pence (replacing 99 with 59 in the above program). For the full problem the system runs out of memory. There are standard MIP solutions to the problem which run overnight, but it is a tough challenge to reduce this time from hours to minutes!

In figure 7 we illustrate an ECL^{i}PS^e program for solving the "Coins" problem using the facilities of the ECL^{i}PS^e finite domain constraint solver implemented in the ECL^{i}PS^e library *fd*.

In this case the `#=` and `#>=` constraints and the optimisation predicate *minimize* are implemented in the ECL^{i}PS^e finite domain library. This program proves within a few seconds that the minimum number of coins a purchaser needs in their pocket to make up any total between 1 and 99 is eight coins! One solution is: $P = 1, Tw = 2, Fv = 1, Te = 1, Twe = 2, Ff = 1$.

We have shown how the same underlying model for the "Coins" problem can be passed to different solvers so as to use the best one. However in ECL^{i}PS^e it is not a choice of either/or: the same constraints can easily be passed to several solvers at the same time! For instance we can define `X $#= Y` to be both `X $= Y` and `X #= Y` and replace `=` in the above model with `$#=`, and we can treat `>=` similarly.

Whilst for this problem the finite domain solver alone solves the problem most efficiently, we have encountered practical examples where the combination of both solvers outperforms each on its own.

```
:- lib(apply_macros).
:- lib(fd).

solve(PocketCoins,Min) :-
        PocketCoins=[P,Tw,Fv,Te,Twe,Ff],
        applist(range(0,99),[Min|PocketCoins]),
        Min #= P+Tw+Fv+Te+Twe+Ff,
        fromto(1,99,genc(PocketCoins)),
        minimize(labeling(PocketCoins),Min).


genc(PocketCoins,Total) :-
        Coins=[P1,Tw1,Fv1,Te1,Twe1,Ff1],
        applist(range(0,99),Coins),
        Total #= P1+2*Tw1+5*Fv1+10*Te1+20*Twe1+50*Ff1,
        maplist( ´#<=´,Coins,PocketCoins).

range(Min,Max,Var) :-
        Var::Min..Max
```

Figure 7: *fd* Constraints for the Coins Problem

3 Solvers and Syntax

ECL^{i}PS^e offers several different libraries for handling symbolic and numeric constraints. They are the *fd* (finite domain) library, the *range* library, the *ria* (real number interval) library, and finally the *eplex* (MIP) library.

3.1 The *fd* (Finite Domain) Library

The finite domain library has been used and refined over a 10 year period. As a result it has a great many constraint handling facilities. It is best seen as three libraries.

The first is a library for handling symbolic finite domains, with values like *red, machine_1* etc. The built-in constraints on symbolic finite domain variables are equations and disequalities: these constraints can only hold between expressions which are either constants or variables. These constraints can also be used when the domains are numeric.

The second is a library for handling integer variables, and numerical constraints on those variables. The library propagates equations and inequalities between linear expressions. A linear numeric expression is one that can be written in the form $Term_1 + Term_2 + \ldots + Term_n$, where each term can, in turn, be written $Number$ or $Number * Variable$. The number can be positive or negative. An example is the expression $3 * X + (-4) * Y + 3$ (which we would normally write $3 * X - 4 * Y + 3$).

The third is a library supporting some built-in complex constraints. Two examples of such constraints are the `alldistinct` constraint, which constraints a set of variables to take values which are pairwise distinct, and the `atmost` constraint, which constrains at most N variables from a given set to take a certain value.

3.1.1 The *fd* Symbolic Finite Domain Facilities

In figures 1 and 4, above, we showed a map colouring problem and its solution. The domains associated with the countries were *red, green* and *blue*. These were declared as finite domains, with the usual syntax: X *:: [red, green, blue]*.

The problem could have been modelled using numbers to represent colours, so there is no extra power in allowing symbolic finite domains as well as numeric ones. However when developing ECL^{i}PS^e programs for real problems, it is a very great help to use meaningful names so as to distinguish different types of finite domain variables. In particular it is crucial during debugging!

Figure 8 illustrates the basic constraints on finite domain variables, and predicates for accessing and searching these domains.

The second query associates a symbolic finite domain with the variable X. In response ECL^{i}PS^e prints out the variable name and its newly assigned domain. The fact that the variable has an associated domain does not require any changes in other parts of the program, where X may be treated as an ordinary variable.

Query 3 shows that symbolic domains can include values of different types.

Query 4 shows the use of the *dom* predicate to retrieve the domain associated with a variable.

Queries 5 and 6 illustrate the equality and disequality constraints, and their effects on the domains of the variables involved. Finite domain constraints use a special syntax to make explicit which constraint library is to handle the constraint, for example it uses `#=` instead of `=`.

Queries 7, 8 and 9 illustrate search. Strictly one would not expect search predicates to belong to a constraint library, but in fact search and constraint propagation are closely connected.

Query 7 shows the *indomain* predicate instantiating a domain variable X to a value in its domain. ECL^{i}PS^e asks if more answers are required, and when the user does indeed ask for more, another value from the domain of X is chosen, and X is instantiated to that value instead. When the user asks for more again, X is instantiated to the third and last value in its domain, and this time ECL^{i}PS^e doesn't offer the user any further choices, but simply outputs *yes*.

Query 8 illustrates the built-in finite domain *labeling* predicate. This predicate simply invokes *indomain* on each variable in turn in its argument. In this case it calls *indomain* first on X, then Y and then Z. However the variables are constrained to take different values by three disequality constraints, and only those labelings that satisfy the constraints are admitted. Consequently this query has six different answers, though the user stops asking for more after the second answer.

Query 9 illustrates a heuristic based on the *fail first* principle. In choosing the next decision to make, when solving a problem, it is often best to make the choice with the fewest alternatives first. The predicate *deleteff* selects a variable from a set of variables which has the fewest alternatives: i.e. the smallest finite domain. In the example there are three variables, X, Y and Z representing three decisions, *deleteff* picks out Y because it has the smallest domain, and then *indomain* selects a value for Y. The third argument of *deleteff* is an output argument: *Rest* returns the remaining variables after the selected one has been removed. These are the decisions yet to be made.

3.1.2 The *fd* Integer Arithmetic Facilities

For numeric finite domains the *fd* library admits equations, inequalities and disequalities over numeric expressions.

Additionally the *fd* library includes some built-in optimisation predicates. These are all illustrated in figure 9.

Query 2 illustrates how a numeric finite domain can be initialised just by giving lower and upper bounds, instead of the whole list of members. In fact, internally, finite domains are stored as lists of intervals (for example *[1..5, 8..10, 15]*).

Query 3 shows how the user can find out the lower bound of a variable's numeric finite domain. There is a similar predicate for retrieving the upper bound.

Queries 4, 5 and 6 illustrate some features of finite domain constraint propagation.

Query 4 shows the pruning achieved by a simple numerical finite domain constraint. Notice that both the domains of X and Y are pruned - constraints work in all directions!

Query 5 illustrates that a finite domain constraint remains active even after it has achieved some pruning. This query is the same as query 3, with an extra constraint imposed subsequently. The $X\# > Y + 1$ constraint is still active, and prunes the domain of X still further from *[3..10]* to *[8..10]*.

Query 6 shows that, in the interest of computational efficiency, the mathematical constraints only narrow the bounds of the finite domains. In this example the domain of X could theoretically be reduced to *[4,6,8,10]*, but this would require much more computation - especially if the finite domains were quite large!

```
[eclipse 1]: lib(fd).
*       fd loaded

[eclipse 2]:  X::[a,b,c].
*       X = X{[a, b, c]}
*       yes.

[eclipse 3]: X::[a, 3.1, 7].
*       X = X{[3.0999999, 7, a]}
*       yes.

[eclipse 4]: X::[a,b,c], dom(X,List).
*       X = X{[a, b, c]}
*       List = [a, b, c]
*       yes.

[eclipse 5]: X::[a,b,c], Y::[b,c,d], X#=Y.
*       X = X{[b, c]}
*       Y = X{[b, c]}
*       yes.

[eclipse 6]: X::[a,b,c], X##b.
*       X = X{[a, c]}
*       yes.


[eclipse 7]: X::[a,b,c], indomain(X).
*       X = a     More? (;)
*       X = b     More? (;)
*       X = c
*       yes.

[eclipse 8]: [X,Y,Z]::[a,b,c], X##Y, Y##Z, X##Z, labeling([X,Y,Z]).
*       X = a
*       Y = b
*       Z = c     More? (;)

*       X = a
*       Y = c
*       Z = b     More? (;)
*       yes.

[eclipse 9]:  [X,Z]::[a,b,c], Y::[a,c],
              deleteff(Var,[X,Y,Z],Rest), indomain(Var).

*       X = X{[a, b, c]}
*       Y = a
*       Z = Z{[a, b, c]}
*       Rest = [X{[a, b, c]}, Z{[a, b, c]}]
*       Var = a     More? (;)
*       yes.
```

Figure 8: Using Symbolic Finite Domains

```
[eclipse 1]: lib(fd).
*       fd loaded

[eclipse 2]: X::1..10.
*       X = X{[1..10]}
*       yes.

[eclipse 3]: X::1..10, mindomain(X,Min).
*       X = X{[1..10]}
*       Min = 1
*       yes.

[eclipse 4]: [X,Y]::1..10, X#>Y+1.
*       X = X{[3..10]}
*       Y = Y{[1..8]}
*       yes.

[eclipse 5]: [X,Y]::1..10, X#>Y+1, Y#=6.
*       X = X{[8..10]}
*       Y = 6
*       yes.

[eclipse 6]: [X,Y,Z]::1..10, X #= 2*(Y+Z).
*       X = X{[4..10]}
*       Y = Y{[1..4]}
*       Z = Z{[1..4]}
*       yes.

[eclipse 7]: X::1..10, mindomain(X,Min).
*       X = X{[1..10]}
*       Min = 1
*       yes.

[eclipse 8]: [X,Y,Z]::1..10, X #= 2*(Y+Z), Y##Z,
        minimize(labeling([X,Y,Z]),X).
*       Found a solution with cost 6
*       Y = 2
*       Z = 1
*       X = 6
*       yes.
```

Figure 9: Numeric Finite Domains

Query 7 is an example of the use of the built-in *minimize* predicate. This predicate returns an admissible labeling of the variables X, Y and Z which yields the smallest value for X. In general any search procedure can be substituted for *labeling([X,Y,Z])* as the first argument to *minimize*. For example we could have used *minimize((indomain(X), indomain(Y), indomain(Z)), X)*.

3.1.3 The *fd* Complex Constraints

There are two motivations for supporting complex constraints. One is to simplify problem modelling. It is shorter, and more natural, to use a single *alldistinct* constraint on N variables than to use $n*(n-1)/2$ (pairwise) disequalities!

The second motivation is to achieve specialised constraint propagation behaviour. The `alldistinct` constraint on N variables, has the same semantics as $n*(n-1)/2$ (pairwise) disequalities, but it can also achieve better propagation than would be possible with the disequalities. For example if any M of the variables have the same domain, and its size is less than M, then the `alldistinct` constraint can immediately fail. However if two variables X and Y have the same domain, with $M > 1$ elements, the constraint X `##` Y can achieve no propagation at all. Thus the pairwise disequalities are unable to achieve the same propagation as the *alldistinct* constraint.

The constraint $atmost(Number, List, Val)$ constrains atmost $Number$ of the variables in the list *List* to take the value Val. This is a difficult constraint to express using logic. One way is to constrain each sublist of length $Number+1$ to contain a variable with value different from Val, but the resulting number of constraints can be very large!

A more natural way is to constrain all the variables to take a value different from Val, and to allow the constraint to be violated up to N times. The *fd* library supports such a facility with the constraint `#=`*(T1, T2, B)*. This constraint makes $B = 1$ if $T1 = T2$ and $B = 0$ otherwise. It is possible to express *atmost* by imposing the constraint $\# = (Var_i, Val, B_i)$ for each variable Var_i in the list and then adding the constraint $B_1 + \ldots + B_m \# <= N$. The built-in *atmost* constraint is essentially implemented like this.

The other *fd* constraints (`#<`, `#>`, etc.) can be extended with an extra *0/1* variable in the same way.

The *fd* library includes a great variety of facilities, which are best explored by obtaining the ECL^{i}PS^e extensions manual [Be97] and looking at the programming examples in the section on the *fd* library there.

3.2 The *range* Library

The range library does very little itself, but it provides a common basis for the interval and the MIP libraries. By contrast with the finite domain library, the *range* library admits ranges whose lower and upper bound are either real numbers or integers. The library enables the programmer to associate a range with one or more variables, as illustrated in figure 10.

```
[eclipse 1]: lib(range).
*       range loaded

[eclipse 2]: X::0.0..9.5, lwb(X,4.5).
*       X = X{4.5 .. 9.5}
*       yes.
[eclipse 3]: X::4.5..9.5, X=6.0.
*       X = 6.0
*       yes.
[eclipse 4]: X::4.5..9.5, X=1.0.
*       no (more) solution.
[eclipse 5]: X::0.0..9.5, lwb(X,4.5), integers([X]).
*       X = X{5 .. 9}
*       yes.
```

Figure 10: Example Queries Using the *range* Library

In query 2, the programmer enters `X::0.0..9.5, lwb(X,4.5).`, and the system responds by printing out the resulting range. When the variable is instantiated, the range is checked for compatibility, as shown by queries 3 and 4.

Finally, what might be treated as type information in other programming paradigms, can be treated as a constraint in the constraint programming paradigm. Thus we can add a constraint that something is an integer in the middle of a program, as shown by query 5.

3.3 The *ria* (Real Interval Arithmetic) Library

The *ria* library supports numeric constraints which may involve several variables. Throughout program execution, *ria* continually narrows the ranges associated with the variables as far as possible based on these constraints. In other words *ria* supports propagation of intervals, using the range library to record the current ranges, and to detect inconsistencies.

The constraints handled by *ria* are equations and inequalities between numerical expressions. The expressions can be quite complex, they can include polynomials and trigonometrical functions. This functionality is quite similar to that offered by *fd*, except that *fd* can only propagate linear constraints. On the other hand, the finite domain library uses integer arithmetic instead of real number arithmetic, so it is in general more efficient than *ria*.

We shall confine ourselves here to a single example showing *ria* at work.

Suppose we wish to build a garden house, whose corners must lie on a given circle. The house should be a regular polygon, but may have any number of sides. It should be as large as possible within these limitations. (Note that the more sides the larger the area covered, until it covers practically the whole of the circle.) However each extra side incurs a fixed cost. The problem is to decide how many sides the garden house should have?

If it had six sides, the house would look as illustrated in figure 11.

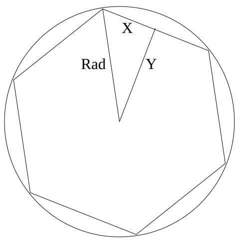

Figure 11: The Garden House

The area of the house is *2*6*A* where A is the area of the triangle in the illustration. The area of an N-sided house can be modelled in ECL^{i}PS^e as shown in figure 12. N is the number of sides, and $Area$ is the area of the house. The variable *Rad* denotes the radius of the circle, and X and Y are the lengths of the sides of the triangle, as illustrated in figure 11.

ria requires its constraints to be written with a specific syntax (eg `X *>= Y` instead of just `X >= Y`). This distinguishes *ria* constraints from linear and finite domain constraints, which each have their own special syntax.

To work out the payoff between the area and the number of sides, we define the cost of the house to be $W1 * Area - W2 * N$, where $W1$ and $W2$ are weighting factors that we can chose to reflect the relative costs and benefits of the area against the number of sides. In the model shown in figure 12, *tcost* returns the cost-benefit of an N-sided house in case the radius of the circle is 10, and the weights are $W1 = 1$ and $W2 = 10$.

We can place this program in a file called *house.pl*, and then use ECL^{i}PS^e to find out some costs by first "consulting" the file, as illustrated figure 13, query 1.

Queries 2-6, in figure 13, would seem to indicate that the seven sided house is best for the given cost weightings.[3]

However it is also interesting to see whether the interval reasoning system itself can achieve useful propagation without even knowing the number of sides of the house. We show this in query 7.

[3] The intervals returned from *ria* are much narrower than this, but for this paper I have reduced the output to three significant figures.

```
:- lib(ria).

area(N,Rad,Area) :-
        X*>=0, Y*>=0, N*>=3, integers(N),
        Rad *>=0, Area*>=0,
        Area *=< pi*sqr(Rad),
        cos(pi/N) *= Y/Rad,
        sqr(Y)+sqr(X) *= sqr(Rad),
        Area *= N*X*Y.

cost(N,Rad,W1,W2,Cost) :-
        W1*>=1, W2*>=1, Cost *>=0,
        area(N,Rad,Area),
        Cost *= W1*Area-W2*N.

tcost(N,Cost) :-
        cost(N,10,1,10,Cost).
```

Figure 12: The Area, and Cost-Benefit, of a Garden House

```
[eclipse 1]: [house].
*       range  loaded
*       house.pl   compiled
*       yes.

[eclipse 2]: tcost(3,C).
*       C = C{99.90 .. 99.91}
*       yes.
[eclipse 3]: tcost(4,C).
*       C = C{159.9 .. 160.0}
*       yes
[eclipse 4]: tcost(6,C).
*       C = C{199.8 .. 199.9}
*       yes.
[eclipse 5]: tcost(7,C).
*       C = C{203.6 .. 203.7}
*       yes.
[eclipse 6]: tcost(8, C).
*       C = C{202.8 .. 202.9}
*       yes.

[eclipse 7]: tcost(N,C).
*       N = N{3 .. 31}
*       C = C{0.0 .. 284.2}
*       yes.
[eclipse 8]: tcost(N,C), squash([C],1e-2,lin).
*       N = N{3 .. 31}
*       C = C{0.0 .. 224.2}
```

Figure 13: Finding the Optimal Shape for a Garden House

An upper bound on the number of sides is extracted due to the constraint that the cost-benefit must be positive, but the propagation on the cost-benefit is rather weak. In cases like this, propagation can be augmented by a technique known as squashing, as illustrated in query 8.

We now give two short examples showing limitations of interval reasoning in general. This will motivate the introduction of a linear constraint solver in ECL^{i}PS^e , described in section 3.4.

The two limitations are that interval reasoning cannot, even in some quite simple examples, detect inconsistency among the constraints; and in cases where the constraints have only one solution, interval reasoning often fails to reflect this in the results of propagation.

This is illustrated by a couple of simple examples in figure 14. In this case the system

```
[eclipse 1]: lib(ria).
*       ria loaded

[eclipse 2]: [X,Y,Z]:: -100.0..100.0,
             X+Y *=<1, Z+X*=<1, Y+Z*=<1, X+Y+Z*>=2.
*       X = X{-100.1 .. 100.1}
*       Y = Y{-100.1 .. 100.1}
*       Z = Z{-100.1 .. 100.1}
*       yes

[eclipse 3]: [X,Y]:: -100.0 .. 100.0, X+Y *= 2, X-Y *= 0.
*       X = X{-98.1 .. 100.1}
*       Y = Y{-98.1 .. 100.1}
*       yes
```

Figure 14: Poor Interval Propagation Behaviour

failed to detect the inconsistency in query 2, and did not deduce that only one value was possible for X and Y in query 3. The answer is not incorrect, as *ria* only guarantees that any possible answers must lie in the intervals returned - it does not guarantee the existence of an answer in that interval. Nevertheless it would be useful to have available a more powerful solver to recognise cases such as these.

3.4 The *eplex* (External CPLEX Solver Interface) Library

Equations and inequalities between linear numeric expressions, as defined in section 3.1 above, are a subset of the constraints which can be handled by *ria*. However this class can be handled very powerfully, so much so that any inconsistency between the constraints is guaranteed to be detected. Techniques for solving linear constraints have been at the heart of operations research for half a century, and highly efficient linear solvers have been developed.

One of the most widely distributed, scaleable and efficient packages incorporating a linear constraint solver is the CPLEX MIP package [CPL93]. CPLEX offers several algorithms for solving linear constraints including the Simplex and Dual Simplex algorithms. These algorithms are supported by sophisticated data structures, and the package can handle problems involving ten of thousands of linear constraints over ten of thousands of variables.

In the discussion so far, we have not yet mentioned an important aspect of most industrial combinatorial problems. Not only is it required to make decisions that satisfy the constraints, but they must also be chosen to optimise some measure. In the travelling salesman problem for example, the decisions of what order to visit the cities are based on optimising the total distance travelled by the salesman.

One feature of available packages for solving linear and mixed integer problems, is support for optimisation. Figure 15 is a design model for a transportation problem, which uses the *eplex* library to pass the constraints to the CPLEX package.

Note that, where *fd* uses a # and *ria* uses a *, *eplex* uses a $.

The answer returned by ECL^{i}PS^e is

```
:- lib(eplex).

main(Cost, Vars) :-

        % Transportation problem:  clients A,B,C,D, plants 1,2,3.
        % Variables represent amount delivered from plant to client.

        Vars = [A1, B1, C1, D1, A2, B2, C2, D2, A3, B3, C3, D3],
        Vars :: 0.0..10000.0,                   % variables

        A1 + A2 + A3 $= 200,            % client demand constraints
        B1 + B2 + B3 $= 400,
        C1 + C2 + C3 $= 300,
        D1 + D2 + D3 $= 100,

        A1 + B1 + C1 + D1 $=< 500,      % plant capacity constraints
        A2 + B2 + C2 + D2 $=< 300,
        A3 + B3 + C3 + D3 $=< 400,

        optimize(min(                   % solve minimizing
            10*A1 + 7*A2 + 11*A3 +      % transportation costs
             8*B1 + 5*B2 + 10*B3 +
             5*C1 + 5*C2 +  8*C3 +
             9*D1 + 3*D2 +  7*D3), Cost).
```

Figure 15: A Design Model for a Transportation Problem

C = 6600.0 V = [0.0, 200.0, 300.0, 0.0, 0.0, 200.0, 0.0, 100.0, 200.0, 0.0, 0.0, 0.0]

In fact MIP packages typically not only offer optimisation as a facility, they insist on an optimisation function in the design model. Therefore in illustrating the two examples where *ria* performed badly, using instead the *eplex* library, we shall insert a dummy optimisation function! The use of *eplex* on these examples is shown in figure 16.

Query 2 is the same set of constraints whose inconsistency is not detected by *ria*. *eplex*, however, recognises their inconsistency.

In order to establish that there is only one possible value for X we have had to use two queries, 3 and 4, first finding the minimum value for X and then the maximum. Although the same value for Y was returned in both solutions, the *eplex* library has still not proved that 1 is the only possible value for Y.

For problems involving only real number (or *continuous*) variables, linear constraint solving alone suffices to solve the problem. However industrial problems typically include a mixture of real number and integer variables. For example in problems involving discrete resources the decision as to which resource to use for a task cannot be modelled as a continuous variable. Traditionally operational researchers will use a binary (or *0/1*) variable or an integer variable. Most resources are discrete, for example machines for jobs, vehicles for deliveries, rooms for meetings, berths for ships, people for projects, and so on. Another fundamental use of discrete variables is in modelling the decision as to which order to do things in - for example visiting cities in the travelling salesman problem, or performing tasks on the same machine.

From the point of view of the programmer, adding the constraint that a variable is integer-valued is straightforward. However the effect of such a constraint on the performance of the solver can be disastrous, because mixed integer problems are much harder to solve than linear problems!

The *eplex* library uses standard range-variables provided by the range-library, which facilitates interfacing to other solvers. The interface to CPLEX enables state information to be retrieved, such as constraint slacks, basis information, and reduced costs. Also many para-

```
[eclipse 1]: lib(eplex).
*       eplex loaded

[eclipse 2]: X+Y $=< 1, Z+X $=< 1, Y+Z $=< 1, X+Y+Z $>= 2,
             Opt $= 0, optimize(min(Opt),Cost).
*       no (more) solution.

[eclipse 3]: X+Y $= 2, X-Y $= 0, optimize(min(X),Cost).
*       X = 1.0
*       Y = 1.0
*       Cost = 0.0
*       yes.

[eclipse 4]: X+Y $= 2, X-Y $= 0, optimize(max(X),Cost).
*       X = 1.0
*       Y = 1.0
*       Cost = 0.0
*       yes.
```

Figure 16: Linear Constraint Solving

```
[eclipse 1]: lib(eplex).
*       eplex loaded

[eclipse 2]: X+Y $>= 3, X-Y $= 0, optimize(min(X), C).
*       Y = 1.5
*       X = 1.5
*       C = 1.5
*       yes.

[eclipse 3]: integers([X]), X+Y $>= 3, X-Y $= 0, optimize(min(X), C).
*       Y = 2.0
*       X = 2
*       C = 2.0
*       yes.
```

Figure 17: Mixed Integer Programming

meters can be queried and modified. A quite generic solver demon is provided which makes it easy to use CPLEX within a data-driven CLP setting.

The notion of solver handles encourages experiments with multiple solvers. A pair of predicates make it possible to read and write problem files in MPS or LP format.

MIP packages such as CPLEX and XPRESS , which is also currently being integrated into the *eplex* package, are surprisingly effective even for some problems involving many discrete variables. Their underlying linear solvers reflect a carefully chosen balance between flexibility and scaleability. They offer less flexibility than the linear solvers which are usually built into constraint programming systems, such as $CLP(R)$, but much better scaleability.

It has proved possible, within ECL^{i}PS^e , to achieve much of the flexibility of $CLP(R)$ within the restrictions imposed by MIP solvers [RWH97].

4 Complex Constraints

Whilst constraint programming languages offer a broad selection of built-in constraints, each new industrial application typically requires a number of application-specific constraints which aren't among the built-in constraints.

Let us take, as an ongoing example, the constraint that two tasks sharing a single resource cannot be done at the same time. This constraint was introduced in section 2.2.2 above.

The constraint involves six variables: the start times S_1, S_2, end times E_1, E_2 and resources R_1, R_2 of the two tasks. The specification of this constraint is as follows:

> *either* the two tasks use distinct resource ($R_1 \quad ne \quad R_2$) *or* task$_1$ ends before task$_2$ starts ($E_1 \leq S_2$) *or else* task$_2$ ends before task$_1$ starts ($E_2 \leq S_1$).

We shall compare three different ways of handling this constraint.

First we recall how it can be encoded using numerical equations and inequalities, together with integer constraints. This is the encoding necessary to allow it to be solved using MIP algorithms, as available through the *eplex* library. However the MIP package is not necessarily the best algorithm for handling such a constraint.

Indeed experience with practical applications suggests that the more *0/1* variables it is necessary to introduce to handle each constraint, the less efficient MIP becomes. The inefficiency comes partly because the MIP constraints are handled globally, and the cost of handling extra constraints and boolean variables increases very fast with their number.[4] It also comes because, until the boolean variables take a value very close to 0 they have very little effect on the search.[5]

By contrast we shall show how it can be handled using two further libraries of ECL^{i}PS^e - the *propia* library and the *chr* library. These libraries allow the constraint to be modelled much more simply and handled more efficiently.

4.1 The *propia* (Generalised Propagation) Library

A major issue in defining complex constraints is how to handle disjunction. The resource constraint of our running example can be quite easily expressed using a disjunction of finite domain constraints. Indeed ECL^{i}PS^e allows us to express disjunction as alternative clauses defining a predicate, so the constraint can be expressed as a single ECL^{i}PS^e predicate thus:

```
fdTaskResource(S1,E1,R1,S2,E2,R2) :-
        R1 ## R2.
fdTaskResource(S1,E1,R1,S2,E2,R2) :-
        R1#=R2, S1 #>= E2.
fdTaskResource(S1,E1,R1,S2,E2,R2) :-
        R1#=R2, S2 #>= E1.
```

The purpose of the *propia* library is to take exactly such disjunctive definitions and turn them into constraints!

This is illustrated in figure 18. The syntax *Goal infers most* turns any ECL^{i}PS^e goal into

[4] Using the Simplex or Dual Simplex algorithms this cost goes up, in the worst case, exponentially with the number of constraints and variables.

[5] Technically they are rarely facet-inducing cuts.

```
:- lib(propia).

propiaTR(S1,R1,S2,R2) :-
        [S1,S2]::0..100, [R1,R2]::[r1,r2,r3],
        E1 = S1+50,  E2 = S2+70,
        fdTaskResource(S1,E1,R1,S2,E2,R2) infers most.

fdTaskResource(S1,E1,R1,S2,E2,R2) :-
        R1 ## R2.
fdTaskResource(S1,E1,R1,S2,E2,R2) :-
        R1#=R2, S1 #>= E2.
fdTaskResource(S1,E1,R1,S2,E2,R2) :-
        R1#=R2, S2 #>= E1.
```

Figure 18: Specifying a Resource Contention Constraint in ECL^{i}PS^e

a constraint. It is supported by the *propia* library.

The behaviour of this constraint is to find which values for each variable are consistent with the constraint. The constraint has the propagation behaviour described in [Wal97]: it repeatedly attempts to reduce the domains of its variables further every time any other constraints reduce any of these domains. Figure 19 shows some examples of this behaviour. In query 2, the constraint deduces that, since the tasks cannot overlap, task_1 cannot start between 51 and 69, and task_2 cannot start between 31 and 49. In query 3, since the tasks are bound to overlap, the constraint deduces that task_2 must use either resource r_2 or r_3.

```
[eclipse 1]: [fdTaskResource].
*     propia     loaded
*     fdTaskResource.pl compiled.
*     yes.

[eclipse 2]: propiaTR(S1, R1, S2, R2), R1#=r1, R2#=r1.
*     S1 = S1{[0..50, 70..100]}
*     R1 = r1,
*     S2 = S2{[0..30, 50..100]},
*     R2 = r1
*     yes.

[eclipse 3]: propiaTR(S1,R1,S2,R2), R1=r1, S2#>=35, S2#<=45.
*     S1 = S1{[0..100]}
*     R1 = r1,
*     S2 = S2{[35..45]},
*     R2 = R2{[r2, r3]}
*     yes.
```

Figure 19: The Behaviour of *Goal infers most*

Other behaviour can be achieved by writing *Goal infers consistent* or *Goal infers ground* instead. These behaviours, together with other facilities of the *propia* library are described in the ECL^{i}PS^e extensions manual [Be97].

4.2 The *chr* (Constraint Handling Rules) Library

The ECL^{i}PS^e programmer has little control over the behaviour of complex predicates using the *propia* library. For example in the fdTaskResource query 2, illustrated in figure 19, the constraint detects "holes" inside the domains of the variables $S1$ and $S2$. However experience in solving scheduling problems suggests that the computational effort expended in detecting such holes is rarely compensated by any reduction the amount of search necessary to find solutions. Whilst this propagation is too powerful, the other alternatives available in the *propia* library are too weak.

The most useful behaviour for the constraint is to do nothing until one of the following conditions hold:

- If the tasks are guaranteed to overlap, constrain them to use distinct resources
- If the tasks must use the same resource, and one of the tasks cannot precede the other, constrain that task not to start until the other task has ended.

Notice that this is, unfortunately, not the behaviour achieved by the MIP encoding, either!

This behaviour can be expressed in ECL^{i}PS^e using the Constraint Handling Rules *chr* library. The required ECL^{i}PS^e encoding remains quite logical, but it needs a new concept, that of a *guard*. A rule with a guard is not executed until its guard is entailed, until then it does nothing. The data-driven implementation of guarded rules uses the same mechanisms as the data-driven implementation of constraints discussed in the following section.

The syntax for guarded rules is rather different from the syntax for ECL^{i}PS^e clauses encountered so far. This syntax is illustrated by the encoding of the tsakResources constraint in figure 20. In this example the constraint handling rules use finite domain constraints in their definitions.

```
chrTR(S1,R1,S2,R2) :-
        [S1,S2]::0..100, [R1,R2]::[r1,r2,r3],
        E1 = S1+50,  E2 = S2+70,
        chrTaskResource(S1,E1,R1,S2,E2,R2).

constraints chrTaskResource/6.

chrTaskResource(S1,E1,R1,S2,E2,R2) <==>
        R1 #= R2, E1 #> S2   |   E2 #<= S1.
chrTaskResource(S1,E1,R1,S2,E2,R2) <==>
        R1 #= R2, E2 #> S1   |   E1 #<= S2.
chrTaskResource(S1,E1,R1,S2,E2,R2) <==>
        E1 #> S2, E2 #> S1   |   R1 ## R2.
```

Figure 20: Constraint Handling Rules for the Task Resources Constraint

Logically each of these three rules states the same constraint: either $R1 \neq R2$ or $S2 \geq E1$ or $S1 \geq E2$. However each rule uses a different "if...then" statement. For example the first rule says that if $R1 = R2$ and $E1 > S2$ then $S1 \geq E2$.

In order to use constraint handling rules, it is necessary to translate them into the underlying ECL^{i}PS^e language using an automatic translator. The constraints must be written to a file called *file.chr* - in our example we shall use *chrTaskResource.chr*. To illustrate the loading and use of constraint handling rules, we give some example queries in figure 21.

Query 3 yields less propagation than *propiaTR* because this implementation does not punch holes in the variables' domains.

Query 4 does, however, produce new information, because not only do both tasks use the same resource, but also the constraint $S1 \leq 65$ means that task$_1$ must precede task$_2$. The constraint deduces that the latest start time for $S1$ is actually 50, and the earliest start time for $S2$ is also (by coincidence) 50.

Query 5 uses the fact that the tasks must overlap to remove r_1 from the domain of $R2$.

The *chr* library offers many more facilities, including multi-headed rules, and augmentation rules. These facilities can be explored in detail by studying the relevant chapter in [Be97], and trying out the example constraint handling rule programs which are distributed with ECL^{i}PS^e .

```
[eclipse 1]: lib(chr), lib(fd).
*     chr loaded
*     fd loaded

[eclipse 2]: chr(chrTaskResource).
*       chrTaskResource.chr compiled.
*       yes.

[eclipse 3]: chrTR(S1, R1, S2, R2), R1#=r1, R2#=r1.
*       S1 = S1{[0..100]}
*       S2 = S2{[0..100]}
*       R1 = r1
*       R2 = r1
*       yes.

[eclipse 4]: chrTR(S1, R1, S2, R2), R1=r1, R2=r1, S1#<=65.
*       S2 = S2{[50..100]}
*       R1 = r1
*       R2 = r1
*       S1 = S1{[0..50]}
*       yes.

[eclipse 5]: chrTR(S1,R1,S2,R2), R1=r1, S2#>=35, S2#<=45.
*       S1 = S1{[0..100]}
*       R2 = R2{[r2, r3]}
*       R1 = r1
*       S2 = S2{[35..45]}
*       yes.
```

Figure 21: The Behaviour of *chrTaskResource*

4.3 Explicit Data Driven Control

The *propia* and *chr* libraries are implemented using a set of underlying facilities in ECL^{i}PS^e which support data-driven computation. The main feature supporting data-driven computation is the *suspension*. This is illustrated in figure 22.

```
[eclipse 1]:    lib(fd).
*       fd loaded

[eclipse 2]:    suspend(writeln("Wake up!"),1,X->inst),
                writeln("Do this first"),
                X=1.
*       Do this first
*       Wake up!
*       X = 1
*       yes.

[eclipse 3]:    suspend(writeln("Wake up!"),1,X->inst),
                current_suspension(S),
                suspension_to_goal(S,Goal,M),
                kill_suspension(S),
                call(Goal,M).
*       Wake up!
*       ...
*       yes.

[eclipse 4]:    X::1..10,
                suspend(writeln("Wake up!"),1,X->min),
                X#>3,
*       Wake up!
*       X = X{[4..10]}
*       yes.
```

Figure 22: Handling Suspensions

A suspension is a goal that waits to be executed until a certain event occurs. Each suspension is associated with a set of variables, and as soon as a relevant event occurs to any one of the variables in the set, the suspension "wakes up" and the goal is activated. One such event is instantiation: all the suspensions on a variable wake up when the variable is instantiated.

In figure 22 the first query loads the *fd* library, which will be used in the last example. It is preferable to load all the libraries that may be needed at the start of the session.

Query 2 suspends a goal *writeln("Wake up!")* on one variable X. The goal will be executed as soon as X becomes instantiated ($X \rightarrow inst$). When woken the goal will be scheduled with a certain priority. The priority is given as the second argument of *suspend*. In this case the priority is 1, which is the highest priority. The remainder of query 2 performs another write statement and then instantiates X. The output from ECL^{i}PS^e shows that the suspended goal is not executed, until X is instantiated, after the system has already output *Do this first*.

Query 3 shows various facilities for explicitly handling a suspension. The current suspensions can be accessed. (It is also possible to access the just the suspensions on a particular variable.) A suspension can be converted to a goal.[6] A suspension can be "killed", so it is no longer accessible or wakeable. The suspension has no connection to the goal, however, which can still be executed.

To save space the output of variable values is omitted here.

Finally query illustrates another kind of event that can wake up a suspended goal. In this case the goal is suspended until the lower bound of the finite domain associated with X is

[6]The variable M denotes the module in which *writeln* is defined.

tightened ($X \to min$).

There are other events which can wake suspended goals associated with other constraint handlers, but the most general event is that the variable becomes more constrained in *any* way (expressed as $X \to constrained$). Goals suspended in this way will wake when any new constraint on X is added (an *fd* constraint, a *ria* constraint, or an *eplex* constraint.

Finally it is also possible to retrieve goals suspended on a given variable, or those associated with a given event on a given variable.

Based on this simple idea it is possible to define a constraint behaviour explicitly. As a simple example let us make a constraint that two variable differ by more than some input number N. We will call the constraint *ndiff(N,X,Y)*, where N is the difference, and X and Y the two variables. Its behaviour will be to tighten the finite domains of the variables. In figure

```
:- lib(fd).
:- suspend.

ndiff(N,X,Y) :-
    mindomain(X,XMin),
    maxdomain(Y,YMax),
    YMax<XMin+N, !,
    X#>=Y+N.

ndiff(N,X,Y) :-
    mindomain(Y,YMin),
    maxdomain(X,XMax),
    XMax<YMin+N, !,
    Y#>=X+N.

ndiff(N,X,Y) :-
    suspend(ndiff(N,X,Y),3,[X,Y] -> any).
```

Figure 23: Using Suspensions to Implement Constraints

23 we implement a behaviour for our *ndiff* constraint. Since we use underlying *fd* constraints, we load the *fd* library.[7]

The first clause for *ndiff* checks if the lower bound for X is so close to the upper bound for Y, that X cannot be less than Y (if it was, then to satisfy the *ndiff* constraint we would need to have $Y >= X + N$). In this case it imposes the constraint that $X\# >= Y + N$.

The second clause does the symmetrical test on the lower bound of Y and the upper bound of X.

If neither of these conditions are satisfied, then *ndiff* doesn't do anything. It just suspends itself until the finite domains of X or Y are tightened ($[X, Y] - > any$).

This very same mechanism of suspended goals is used to implement all the built-in constraints of ECL^{i}PS^e . For example the constraint $X\# > Y$ is implemented using a goal which is suspended on two events: a change in the maximum of the domain of X, and a change in the minimum of the domain of Y. Typically all the finite domain built-in constraints are suspended on events which occur to the finite domains of their variables.

Before concluding this subsection, we should observe that the different constraint libraries of ECL^{i}PS^e are supported by a very flexible facility. The information about each kind of constraint on a variable is held in a data structure which is attached to the variable called an *attribute*. When the *fd* library is loaded, each variable in ECL^{i}PS^e has a finite domain attribute. If the variable has no finite domain, this attribute contains nothing, and the behaviour of the variable is just as if it had no attribute. On the other hand if the variable does have a finite domain, then the attribute stores the finite domain, as well as pointers to all the suspended goals which are waiting for an event to occur to the finite domain.

[7] The *fd* library automatically loads the *suspend* library, so it is not actually necessary to load *suspend* explicitly.

Naturally *ria* constraints and *eplex* constraints are stored in other attributes, and they have their own suspended goals attached to them.

Any ECL^{i}PS^e user can define and implement a completely new constraint handling library in three steps.

1. A new attribute storing information about the new class of constraints, must be defined.
2. Events specific to this class of constraints must be specified.
3. New constraint behaviours must be implemented in terms of goals which suspend themselves on these events.

The ECL^{i}PS^e extensions manual [Be97] gives an example of defining such a new constraint library.

5 Search

5.1 Constructive Search

5.1.1 Branch and Bound

In the preceding sections we have encountered two optimisation procedures, the finite domain procedure *minimize* and the MIP procedure *optimize*. Both optimisation procedures implement an algorithm called *branch and bound*, which posts a new constraint, each time it finds a solution, that the cost of future solutions must be better than the cost of the current best solution. Eventually the new constraint will be unsatisfiable, and the algorithm will have proved that it has found the optimum.

5.1.2 Depth-First Search and Backtracking

We have also encountered the finite domain search procedure *labeling*, which successively instantiates a list of finite domain variables to values in their domains. In ECL^{i}PS^e the default search method is depth-first search and backtracking on failure. Of the complete search methods available, this is in practice the best because search algorithms with a breadth-first component quickly grow to occupy too much memory. We will discuss some incomplete search methods below.

5.1.3 *Guesses* - Constraints Imposed During Search

Search is, of course, much more general than just labelling. Certainly, for combinatorial problems, it involves making guesses that may later turn out to have been bad. However a guess need not involve guessing a value for a variable, as is done in labelling. For example if a variable X has range *[0..100]*, instead of guessing a precise value for X, it may be useful to perform a binary chop, first guessing that $X \geq 50$, and then, if the guess turns out to be bad, guessing that $X < 50$. A guess in the most general sense is the posting of a new (non-redundant) constraint which narrows the search space. However there is no guarantee that such a guess does not rule out solutions to the problem, therefore the system must also explore the remainder of the search space on backtracking. Typically this is done by imposing the negation of the constraint. However the negation of an inequality $\geq$ is a strict inequality $<$, which can't be handled by linear programming. However in case X is an integer variable, and N an integer, the negation of $X \geq N$ is $X \leq N - 1$ which can be handled.

5.1.4 MIP Search

Finite domain propagation only narrows domains, and does not guarantee to detect all inconsistencies. Thus there is no guarantee that a partial labelling (which assigns consistent values to some of the variables) can always be extended to a complete consistent labelling. However the linear constraint solver available through *eplex* does indeed guarantee to detect all inconsistencies between the linear constraints. On the other hand a linear solver does not take into account the constraint that certain variables can only take integer values, thus it can return proposed solutions in which non-integer values are proposed for integer variables. The linear solver can efficiently find an optimal solution to the problem in which integrality constraints on the variables are ignored. Such an optimum is termed an optimum of the "continuous relaxation" of the problem, or just the "relaxed optimum" for short.

This suggests a different search mechanism, in which a new constraint is added to exclude the non-integer value in the relaxed optimum returned by the linear constraint solver. If the value for integer variable X was 0.5 in the relaxed optimum, for example, a new constraint $X \geq 1$ might be added. Since this excludes other feasible solutions such as $X = 0$, this new constraint is only a guess, and if it turns out to be a bad guess then the alternative constraint $X \leq 0$ is posted instead.

This is the search method used in MIP when *optimize* is called in the *eplex* library.

5.1.5 Search Heuristics based on Hybrid Solvers *fdplex*

MIP search can be duplicated in ECL^{i}PS^e by passing the linear constraints to CPLEX and using the proposed solutions to decide which new constraint to impose (i.e. guess) next. Whilst there is little point in precisely duplicating the MIP search control with ECL^{i}PS^e , it allows the ECL^{i}PS^e programmer to define new search techniques using information from both the *fd* library and from *eplex*.

For example the size of the finite domain, as recorded in the finite domain library, can be used when choosing the next variable on which to guess a constraint. Then the value of this variable in the relaxed optimum returned from *eplex* can be used when choosing to which value to label it first.

This search technique is supported by the ECL^{i}PS^e library *fdplex*, and is illustrated in figure 24. The *fdplex* version of *indomain* selects the value closest to the value at the relaxed

```
:- lib(fdplex).

mylabeling([]).
mylabeling(Vars) :-
        deleteff(Var,Vars,Rest),
        indomain(Var),
        mylabeling(Rest).


solve(X,Y,Z,W) :-
        [X,Y]::1..5,
        [Z,W]::1..100,
        10*Z+7*W+4*X+Y #= 49,
        Cost #= Z-2*W+X-2*Y,
        minimize(mylabeling([X,Y,Z,W]),Cost).
```

Figure 24: Search with the *fdplex* Library

optimum returned by *eplex*. Indeed it is instructive to watch the search taking place using the ECL^{i}PS^e tracing facilities, so we shall load the program of figure 24 into a file called *fdplexsearch.pl*. Now we shall run it as shown in figure 25. In query 1 the *fdplex* is loaded, and it automatically loads the other libraries which are needed. Query 2 sets spypoints on two predicates. Now each time either of these predicates are called, and when they exit, the debugger stops and allows the programmer to study the state of the program execution. Query 3 calls the program defined in figure 24. Before labelling starts the domains of the variables have already been reduced by finite domain propagation. The reduced domains are automatically communicated to the *range* library, and passed into the linear solver. The linear solver (CPLEX) has already been invoked by *eplex* and has returned the values of the variables X, Y, Z, W at the relaxed optimum.

Now *deleteff* selects the variable with the smallest domain, which is Z. The *fdplex indomain* predicate labels Z to the integer value nearest to its value at the relaxed optimum. This wakes the *fd* constraint handler which tightens the domain of X, and it wakes the linear solver which returns a new relaxed optimum with new suggested values for the other variables.

```
[eclipse 1]: [fdplexsearch].
*       fd loaded
*       range loaded
*       eplex loaded
*       fdplex loaded
*       yes.

[eclipse 2]: spy(mylabeling), spy(indomain).
*       spypoint added to mylabeling / 1.
*       spypoint added to indomain / 1.
*       yes.


[eclipse 9]: solve(X, Y, Z, W).
*       CALL   mylabeling([X{eplex:1.0, range : 1..5, fd:[1..5]},
*                          Y{eplex:5.0, range : 1..5, fd:[1..5]},
*                          Z{eplex:1.2, range : 1..3, fd:[1..3]},
*                          W{eplex:4.0, range : 1..4, fd:[1..4]}]) (dbg)?- leap
*       CALL   indomain(Z{eplex:1.2, range : 1..3, fd:[1..3]}) (dbg)?- leap
*       EXIT   indomain(1) (dbg)?- leap
*       CALL   mylabeling([Y{eplex:5.0, range : 1..5, fd:[1..5]},
*                          X{eplex:2.0, range : 1..5, fd:[2..5]},
*                          W{eplex:3.7, range : 1..4, fd:[2..4]}]) (dbg)?- leap
*       CALL   indomain(W{eplex:3.7, range : 1..4, fd:[2..4]}) (dbg)?- leap
*       EXIT   indomain(4) (dbg)?- leap
*       CALL   mylabeling([3, 2]) (dbg)?- no debug

*       Found a solution with cost -11
*       X = 2
*       Y = 3
*       Z = 1
*       W = 4
*       yes.
```

Figure 25: Tracing *fdplex* Search

This time the variable with the smallest domain is W, and this is the one selected for instantiation. Once this has been instantiated to the integer value closest to its suggested value, *fd* propagation immediately instantiates the remaining values.

At the next spy point the user enters *no debug* and tracing is switched off. The optimal solution is indeed the one found first, which testifies to the usefulness of the combined heuristic used in the search.

5.1.6 Incomplete Constructive Search

For real industrial applications, the search space is usually too large for complete search to be possible. The branch and bound search yields better solutions with longer and longer delays until, in many cases, it fails to yield any new solutions but continues searching fruitlessly!

In cases where complete search is impractical, the heuristics guiding the search become very important. If bad heuristics are chosen the search may methodically explore some unpromising corner of the search space yielding very poor solutions which fail to drive the branch and bound search into more fruitful areas. Good heuristics depend on good constraint handling: the information returned from the constraint handlers is crucial in enabling the heuristics to focus search on promising regions. Moreover once some good choices have been made, propagation can achieve even better results supporting even better heuristics for future choices. This positive feedback produces a virtuous spiral.

Received wisdom suggests that local search techniques, based on solution repair, achieve faster convergence on good solutions than constructive search. However on several industrial applications our experience has shown the contrary. Good heuristics tailored to the application at hand has proved more effective in yielding high quality solutions than techniques based on solution repair.

5.1.7 Intelligent Backtracking and *nogood* Learning

ECL^{i}PS^e offers facilities for programmers to define specific constructive search algorithms. Intelligent backtracking has been implemented in ECL^{i}PS^e . However it is not offered as a library, because in practice any reduction in the amount of search due to intelligent backtracking is dominated by the cost of accessing and updating the necessary data structures.

The information about which constraints are involved, when a failure occurs during search, is useful for recording combinations of variable values which are mutually inconsistent. Such conflict sets can be used to impose extra constraints called *nogoods* which are learned during search.

nogood learning has also been implemeted in ECL^{i}PS^e and is proving useful on some benchmark examples, but as yet no library supporting *nogoods* is available. A paper describing this work [RR96] is available from the IC-Parc home page (whose URL is given in section 6 below).

5.2 Solution Repair

At the end of the previous section we suggested that even for incomplete search, constructive search with good heuristics can outperform solution repair. However there are many important examples, such as job-shop scheduling and travelling salesman problems, where repair performs better than constructive search. Moreover repair is very important in handling *dynamic* problems, which change after an initial solution has been found. The problem may be changed because the user is unsatisfied with the solution for reasons which are not captured in the implementation, and adds new constraints to exclude this solution. Otherwise the change may be due to external circumstances such as unplanned delays, rush orders, cancellations, and so on.

ECL^{i}PS^e uses the concept of the *tentative* value to support solution repair. This is the same concept that is used to return proposed values for variables from the linear solver, as discussed in the preceding section. In the case of repair, however, the tentative value comes not from a constraint handler, but from the original solution to the original problem.

When the problem changes, some of the tentative values may no longer satisfy some of the new constraints. Indeed the simplest change is to constrain a variable to take a new value. In this case the tentative value violates the new constraint. In case there is no violation, of course, the tentative values comprise a feasible solution to the new problem and there is no need to repair the solution at all!

The purpose of the ECL^{i}PS^e repair library is to support the process of detecting a variable whose tentative value is in conflict with a constraint, and in detecting further violations that result from choosing a value for a variable that differs from its tentative value.

5.2.1 "Constructive" Repair

There are several very different repair algorithms that arise from different choices of how to change the value of a variable from its tentative value. The algorithm most similar to constructive search simply instantiates the variable to the chosen new value. In this case the tentative values do no more than support a specific heuristic within a constructive search algorithm. Notice that the heuristic can do more than simply choosing the tentative value as the first guess for each variable during labelling. It can also take into account for each value for a variable the number of other tentative values with which it conflicts according to the constraints. Thus when a variable is labelled to a new value, the value is chosen so as to minimise disruption to the original solution.

The ECL^{i}PS^e *repair* library defines primitives for setting a tentative value for a variable (*tent_set*) and for looking it up (*tent_get*). It also supports a special annotation which changes the behaviour of a constraint from propagation to simply checking against the tentative values of their uninstantiated variables. The annotation is written *Constraint r*, where *Constraint* can be any built-in or user-defined constraint. Whenever the check fails, the constraint is recorded as a *conflict constraint*, and full propagation on the constraint is switched on. The set of conflict constraints can be accessed via the predicate *conflict_constraints*. This can be used in the search procedure to decide which variable to label next.

A built-in search predicate called *repair* is provided which selects a variable whose tentative value violates a repair constraint, labels it and succeeds when all the remaining variables have consistent tentative values.

We illustrate this repair algorithm (with an example from the IC-Parc ECL^{i}PS^e library manual [SNE97]) in figure 26. The solutions found are $[1,3,1]$ and $[1,3,2]$, which means that

```
solve(X,Y,Z) :-
      [X,Y,Z]::1..3,                     % the problem variables
      Y ## X r,                          % state the constraints
      Y ## Z r,
      Y #= 3 r,
      [X,Y,Z] tent_set [1,2,3],              % set existing solution
      repair,                            % invoke repair labeling
      [X,Y,Z] tent_get [NewX,NewY,NewZ].  % get repaired solution
```

Figure 26: The "Constructive" Repair Algorithm

only Z has been repaired. Initially only the constraint $Y\#=3$ is inconsistent with the solution so variable Y is repaired to take the value 3. This now affects the constraint $Y\#\#Z$, and Z must be repaired to either 1 or 2.

The constraint $Y\#\#X$ is not affected by the update. In particular, X keeps the value of the existing solution, and is not even being labeled by *repair/0*.

Constructive repair is also known as *informed backtracking* and has been used successfully on a variety of benchmarks [MJPL92].

5.2.2 Weak Commitment

Instead of instantiating a variable in order to repair it, an alternative method is simply to change its tentative value. This approach requires no backtracking, since every conflict can be fixed by just changing tentative values. The disadvantage is that cycles can easily occur in which two variables repeatedly switch their tentative values.

A very successful algorithm based on repairing tentative values is called *Weak Commitment* [Yok94]. On starting all the variables have tentative values. Variables in conflict are repaired

- by instantiating them - until either there are no more conflicts and the algorithm terminates, or the remaining conflicts cannot be repaired. The latter situation occurs when some variable in conflict cannot be instantiated to any value that is consistent with the variables instantiated so far.

When such a dead-end is encountered, the weak commitment algorithm simply uninstantiates all the variables, setting their tentative values to the values they had when they were instantiated. Then the algorithm restarts, fixing conflicts as before.

5.2.3 Local Improvement

Constructive repair and weak commitment are two algorithms designed to find feasible solutions to a problem. In case the problem additionally requires some cost to be minimised, the repair must be adapted to return better and better solutions.

For unconstrained problems, local improvement can be achieved by just changing the value of some variable, having chosen the variable and value such that the cost of the new solution is better than the cost of the previous solution. This idea underlies the various hill-climbing algorithms as well as stochastic techniques such as Simulated Annealing and Tabu search.

For problems with constraints, changing the value of a variable will not necessarily yield a feasible solution. The ECL^{i}PS^e repair library can be used, however, to find a feasible solution which incorporates the change.

A simulated annealing program has been written in ECL^{i}PS^e which ensures that moves respect the problem constraints. The program has been compared with a pure simulated annealing approach which simply associates a cost with violated constraints and otherwise treats the problem as unconstrained. Experiments showed that the "constrained simulated annealing" program outperformed the pure one.

For an industrial application the repair library has been used together with the *eplex* linear constraint library. In the algorithm used for this application, the relaxed optimum is checked against the repair constraints, and at each step a violated constraint is strengthened in such a way that the next solution returned from *eplex* must satisfy it. The algorithm outperforms standard MIP search because the problem is a dynamic constraint problem: there is an original solution and the requirement is to modify that solution to satisfy some new constraints.

Details of these algorithms are beyond the scope of this article, but hopefully this brief survey has offered a glimpse of the power of repair-based search in combination with the different solvers of ECL^{i}PS^e .

6 The ECL^{i}PS^e System

ECL^{i}PS^e is jointly owned by ICL and IC-Parc, which is an ICL-supported research centre at Imperial College. The system can be obtained by ftp from IC-Parc by emailing

`eclipse-request@doc.ic.ac.uk`

ECL^{i}PS^e runs under the Unix operating system (specifically SunOS 4 on Sun-4 hardware, Solaris on Sparc machines and Linux on PC's), and will be available under Windows-NT (version 4.0) by the end of 1997.

ECL^{i}PS^e is embeddable in `C` and `C++` programs. It is available in the form of a linkable library, and a number of facilities are available to pass data between the different environments, to make the integration as close as possible. Naturally facilities are also provided to allow ECL^{i}PS^e to invoke `C` and `C++`.

A tightly integrated graphical system is very useful for program development, and ECL^{i}PS^e offers such an integration to the Tcl/Tk toolkit, which is public domain software available under Unix and Windows. Typically ECL^{i}PS^e is invoked from Tcl which is driven directly by user interactions. An example graphical environment for ECL^{i}PS^e developers is the graphical constraint environment *Grace*, available as an ECL^{i}PS^e library. Grace is implemented using ECL^{i}PS^e and Tcl.

The manuals and other documentation include a manual covering the non-constraint facilities of ECL^{i}PS^e [Ae97], manuals covering the facilities supporting constraints [Be97, SNE97], and information covering the graphical user interface library, and embeddability in `C` and `C++`.

Background references can be found in the list of publications reachable from the IC-Parc home page at http://www-icparc.doc.ic.ac.uk/

7 Conclusion

The ECL^{i}PS^e platform has been under development for over ten years. During that time constraint programming has established itself not only as an important research area, but also in live industrial applications. The market for constraint technology is growing dramatically, to the point that the major vendor of MIP technology (CPLEX) has been recently taken over by a constraint technology vendor (ILOG).

Over the last five years ECL^{i}PS^e has moved on from its early roots in logic programming and constraint propagation, to a focus on hybrid algorithms. A tight integration between MIP and CLP has been developed and hybrid algorithms based on this combination have proved their efficiency on industrial applications. However hybrid search algorithms, in particular utilising solution repair, have also been a focus of research and development.

Based on growing experience with hybrid algorithms, we have been able to separate the features of the different algorithms both from each other, and from the underlying problem model. Consequently we have reached the point where ECL^{i}PS^e can be used to express a clear, precise and neutral conceptual model of an application, and this model can then be extended and annotated at the implementation stage. The result of implementation is a design model which implements fine-grained hybrid algorithms tailored to the application at hand.

This work has been based on experience on a variety of industrial applications. IC-Parc has developed applications for several of its industrial partners, and each application has contributed to the final architecture of the ECL^{i}PS^e platform. Ongoing applications, with partners such as British Airways, Wincanton Transport and Bouygues, continually give rise to new hybrid techniques, and these results will feed back into ECL^{i}PS^e , as the algorithms are encapsulated and added as new libraries.

Nevertheless the real benefit of ECL^{i}PS^e comes not from the algorithms that are already encapsulated as libraries, but from the ease with which new hybrid algorithms can be developed and validated, and delivered into the industrial computing environment.

References

[Ae97] Abderrahamane Aggoun and et.al. *ECLiPSe user manual.* IC-Parc, 1997.

[Be97] Pascal Brisset and et.al. *ECLiPSe Extensions manual.* IC-Parc, 1997.

[CPL93] CPLEX. Using the cplex callable library and cplex mixed integer library. Technical Report Version 2.1, CPLEX Optimisation Inc., 1993.

[MJPL92] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58, 1992.

[RR96] Tom Richards and Barry Richards. Nogood learning for constraint satisfaction. Technical report, IC-Parc, 1996. In Proceedings CP 96 Workshop on Constraint Programming Application.

[RWH97] Robert Rodosek, Mark Wallace, and Mozafian Hajian. A new approach to integrating mixed integer programming with constraint logic programming. Technical report, IC-Parc, 1997. To appear in Annals of Operations Research.

[SNE97] Joachim Schimpf, Stefano Novello, and Hani El Sakkout. *IC-Parc ECLiPSe Library Manual.* IC-Parc, 1997.

[Wal97] Mark Wallace. Constraint programming. Chapter 17 of The Handbook of Applied Expert Systems, CRC Press, 1997.

[Yok94] M. Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *Proc. 12th National Conference on Artificial Intelligence*, pages 313–318, 1994.

Contents