# Constraint Programming

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

September 1995

## 1 Introduction

Constraint programming is a paradigm that is tailored to hard search problems. To date the main application areas are those of planning, scheduling, timetabling, routing, placement, investment, configuration, design and insurance. Constraint programming incorporates techniques from mathematics, artificial intelligence and operations research, and it offers significant advantages in these areas since it supports fast program development, economic program maintenance, and efficient runtime performance. The direct representation of the problem, in terms of constraints, results in short, simple programs that can be easily adapted to changing requirements. The integration of these techniques into a coherent high-level language enables the programmer to concentrate on choosing the best combination for the problem at hand. Because programs are quick to develop and to modify, it is possible to experiment with ways of solving a problem until the best and fastest program has been found. Moreover more complex problems can be tackled without the programming task becoming unmanageable. A tutorial introduction to constraint logic programming can be found in (Frühwirth et.al, 1992).

Constraint logic programming (CLP) combines logic, which is used to specify a set of possibilities explored via a very simple inbuilt search method, with constraints, which are used to minimise the search by eliminating impossible alternatives in advance. The programmer can state the factors which must be taken into account in any solution - the constraints -, state the possibilities - the logic program -, and use the system to combine reasoning and search. The constraints are used to restrict and guide search.

The whole field of software research and development has one aim, viz. to optimise the task of specifying and writing and maintaining correct, functioning programs. Three important factors to be optimised are:

- correctness of programs

- clarity and brevity of programs
- efficiency of programs

Constraint programming is, perhaps, unique in making a direct contribution in all three areas. This is why it is such an exciting paradigm.

# 2 History

In 1963 Sutherland introduced the Sketchpad system, a constraint language for graphical interaction. Other early constraint programming languages were Fikes' Ref-Arf, Lauriere's Alice, Sussmann's CONSTRAINTS and Borning's ThingLab. These languages already offered the most important features of constraint programming: declarative problem modelling and efficient constraint enforcement; propagation of the effects of decisions; flexible and intelligent search for feasible solutions. Each of these three features has been the study of extensive research over a long period.

The current flowering of constraint programming owes itself to a generation of languages in which declarative modelling, constraint propagation and explicit search control are supported in a coherent architecture that makes them easy to understand, combine and apply.

## 2.1 Declarative Modelling and Efficient Enforcement

### 2.1.1 Algorithm = Logic + Control

Declarative programming has a long history yielding languages such as LISP, Prolog and other purer functional and logic programming languages, and of course it underpinned the introduction of relational databases and produced SQL which, for all its faults, is today's most commercially successful declarative programming language.

There has been a recognition that declarative programming has problems with performance and scaleability. One consequence has been a swing back to traditional procedural programming. However, constraint programming, whilst recognising that efficiency is an important issue, retains the underlying declarative approach. The idea is not to abandon declarative programming (that would amount to throwing away the baby with the bathwater), but to augment it with explicit facilities to control evaluation. Hence Kowalski's maxim that *Algorithm = Logic + Control*.

When constraints are used in an application, both the issues of modelling and performance are considered. An early use of constraints was in the modelling of electrical circuits. Such circuits involve a variety of constraints from simple equations (the current at any two points in a sequential circuit is the same), to linear equations (when a circuit divides, the current flowing in is the sum of the

currents flowing out), to quadratic equations (voltage equals current multiplied by resistance) and so on. A constraint solver that can handle all the constraints on a circuit would be prohibitively inefficient. Consequently Sussmann sought to model circuits using only a simple class of constraints. He showed that the lack of expressive power of simple constraints can be compensated for by using multiple orthogonal models of the circuit. The different constraints of the different models interact to produce more information than could be extracted from the models independently.

### 2.1.2 Constraints for Multi-Directional Programming

In many early constraint systems, constraints were little more than functions which were evaluated in a data-driven way. The logic programming paradigm, however, suggested that programs should be runnable "in both directions". In addition to evaluating a function $f(\overline{X})$ yielding the result $Y$, it must be possible to solve the equation $f(\overline{X}) = Y$ for a given value $Y$ but unknown arguments $\overline{X}$.

Naturally when a function is evaluated "backwards" - i.e. from its result producing its input - it is no longer a function! Attempts to integrate functional and logic programming motivated much research on equation solving systems, and in the end spawned constraint logic programming.

It was recognised that constraint solving lies at the heart of logic programming, in its built-in unification. Researchers began to replace (syntactic) unification with other equation solvers. An important example of this was Boolean unification: this is a solver for equations between Boolean expressions, whose possible values are only *true* or *false*. This development has now found a commercially successful application for design and verification of digital circuits. Moreover Boolean unification is also being applied to the design and verification of real-time control software.

### 2.1.3 Constraint Logic programming

Soon an even more radical step was taken when it was recognised that unification could be replaced by any constraint system and solver, provided certain conditions were satisfied. There was no need for a unification algorithm (which reduces an equation between expressions to an equivalent set of variable assignments). Indeed the constraints need not be equations at all.

The resulting scheme (Jaffar and Lassez, 1987) called the Constraint Logic Programming Scheme, and written CLP(X), was illustrated by choosing mathematical equations and inequations as the constraint system, and the Simplex algorithm as the solver. This instance of CLP(X) is called CLP($\mathcal{R}$), and is described in a later section.

It has inspired a whole research area, exploring the interface between logic and mathematical programming. One resulting constraint programming language is 2LP (Linear Programming and Logic Programming) which embeds

mixed integer programming in a constraint programming system. Another is Newton, which uses interval constraints to do solve hard mathematical problems involving polynomials.

Whilst constraint logic programming offers a powerful modelling language, new constraint propagation algorithms and a clean execution model, mathematical programming offers some sophisticated algorithms, highly optimised implementations and a wealth of industrial application know-how.

The next step beyond the standard constraint logic programming scheme was to include more than one constraint system and solver in a single system. Even CLP($\mathcal{R}$) was, in fact, such a combination including syntactic unification, Gaussian elimination and the Simplex. CLP systems nowadays include a variety of solvers which exchange information through shared variables. For numeric variables, in addition to the above solvers, there may also be a Groebner base rewriting system for handling polynomial equations, a very powerful CAD solver and a weaker but very useful constraint handler for reasoning on numeric intervals. The latter three system are typically useful for non-linear constraints, containing expressions in which variables are multiplied together.

## 2.2 Propagation

### 2.2.1 Early Applications

Constraint propagation was used in 1972 for scene labelling applications, and has produced a long line of local consistency algorithms, recently surveyed in (Tsang, 1993).

Constraint propagation offers a natural way for a system to spontaneously produce the consequences of a decision. (*Propagation* is defined in the dictionary as "dissemination, or diffusion of statements, beliefs, practices"). Propagation is the most important form of immediate feedback for a decision-maker.

Propagation works very effectively in interactive decision support tools. In many applications constraint programming is used in conjunction with other software tools, taking their results as input, performing propagation, and outputting the consequences. Typically feedback from the propagation tool is given in the form of a spreadsheet interface.

Many early applications of constraint programming were related to graphics: geometric layout, user interface toolkits, graphical simulations, and graphical editors. Constraint propagation has played a key role in all these applications, with the result that control over the propagation has been thoroughly investigated: leading to a current generation of very high-performance constraint-based graphics application.

### 2.2.2 Constraint Satisfaction Problems

On the other hand constraint propagation has been the core algorithm used in solving a large class of problems termed constraint satisfaction problems

(CSP's). Standard CSP's have a fixed finite number of problem variables, and each variable has a finite set of possible values it can take (called its domain). The constraints between the variables can always be expressed as a set of admissible combinations of values. These constraints can be directly represented as relations in a relational database, in which case each admissible combination is a tuple of the relation. CSP's have inspired a fascinating variety of research because despite their simplicity they can be used to express - in a very natural way - real, difficult, problems[1].

One line of research has focussed on constraint propagation, showing how to propagate more and more information (forward checking, arc-consistency, path-consistency, k-consistency, relational consistency and so on). For example arc-consistency is achieved by reducing the domains of the problem variables until the remaining values are all supported; a value is supported if every constraint on the variable includes a tuple in which the variable takes this value and the other variables all take supported values.

Even if none of the arc-consistent domains are empty, this does not imply the CSP has a solution! To find a solution it is still necessary to try out specific values for the problem variables. Only if all the variables can be assigned a specific value, such that they all support each other, has a solution been found. One algorithm for solving CSP's, which has proved useful in practice, is to select a value for each variable in turn, but, after making each selection, to re-establish arc-consistency. Thus search is interleaved with constraint propagation. In this way the domains of the remaining values are reduced further and further until either one becomes empty, in which case some previous choice must be changed, or else the remaining domains contain only one value, in which case the problem is solved.

Another line of research has investigated the global shape of the problem. This shape can be viewed as a graph, where each variable is a node and each constraint an edge (or hyper-edge) in the graph. Tree-structured problems are relatively easy to solve, but research has also revealed a variety of ways of dealing with more awkward structures, by breaking down a problem into easier subproblems, whose results can be combined into a solution of the original problem. Picturesque names have been invented for these techniques such as "perfect relaxation" and "hinge decomposition".

More recently researchers have begun to explore the structure of the individual constraints. If the constraints belong to certain classes, propagation can be much more efficient - or it can even be used to find globally consistent solutions in polynomial time. Indeed there are quite nice sufficient conditions to distinguish between NP-complete problem classes and problem classes solvable with known polynomial algorithms.

[1] The class of CSP problems is NP complete.

### 2.2.3 Constraint Propagation in Logic programming

The practical benefits of constraint propagation really began to emerge when it was embedded in a programming language (Van Hentenryck, 1989). Again it was logic programming that was first used as a host language, producing impressive results first on some difficult puzzles and then on industrial problems such as scheduling, warehouse location cutting stock and so on (Dincbas et.al., 1988). The embedding suggested new kinds of propagation, new ways of interleaving propagation and search and new ways of varying the propagation according to the particular features of the problem at hand.

These advantages were very clearly illustrated when, using lessons learned from the Operations Research community, constraint logic programming began to outperform specialised algorithms on a variety of benchmark problems. However the main advantage of constraint programming is not the good performance that can be obtained on benchmarks, but its flexibility in modelling and efficiently solving complex problems.

A constraint program for an application such as Vehicle Scheduling, or Bin Packing, not only admits the standard constraints typically associated with that class of problems, but it also admits other side-constraints which cause severe headaches for Operations Research approaches.

Currently several companies are offering constraint programming tools and constraint programming solutions for complex industrial applications. As host programming language, not only logic programming but also LISP and C++ are offered.

## 2.3 Search

The topic of search has been at the heart of AI since GPS. Some fundamental search algorithms were generate and test, branch and bound, the A* algorithm, iterative deepening, and tree search guided by the global problem structure, or by information elicited during search, or by intelligent backtracking.

The contribution of constraint programming is to allow the end user to control the search, combining generic techniques and problem-specific heuristics. A nice illustration of this is the $n$-queens problems: how to place $n$ queens on an $n$X$n$ chess board, so that no queens can take each other. For $n = 8$, depth-first generate-and-test with backtracking is quite sufficient, finding a solution in a few seconds. However, when $n = 16$, it is necessary to interleave constraint propagation with the search so as to obtain a solution quickly. However when $n = 32$ it is necessary to add more intelligence to the search algorithm. A very general technique is to choose as the next variable to label the one with the smallest domain - i.e. the smallest number of possible values. This is called *first-fail*. It is particularly effective in conjunction with constraint propagation, which reduces the size of the domains of the unlabelled variables (as described for arc-consistency above), For $n$ queens, the first-fail technique works very well,

yielding a solution within a second. Unfortunately even first-fail doesn't scale up beyond $n = 70$. However there is a problem-specific heuristic which starts by placing queens in the middle of the board, and then moving outwards. With the combination of depth-first search, interleaved with constraint propagation, using the first-fail ordering for choosing which queen to place next, and placing queens in the centre of the board first, the 70-queens problem is solved within a second, and the algorithm scales up easily to 200 queens[2].

# 3 Programming with a Constraint Store

## 3.1 Primitive Constraints

The traditional model of a computer store admits only two possible states for a variable: assigned or unassigned. Constraint programming uses a generalisation of this model. A so-called *constraint store* can hold partial information about a variable, expressed as constraints on the variable. In this model, an unassigned variable is an unconstrained variable. An assigned variable is maximally constrained: no further non-redundant constraints can be imposed on the variable, without introducing an inconsistency.

Primitive constraints are the constraints that can be held in the constraint store. The simplest constraint store is the ordinary single-assignment store used in functional programming. In our terms it is a constraint store in which all constraints have the form $Variable = Value$.

The first generalisation is the introduction of the logical variable. This allows information of the form $Variable = Term$ to be stored, where $Term$ is any term in first-order logic. For example it is possible to store $X = f(Y)$. The same representation can be used to store partial information about $X$. Thus if nothing is known about the argument of $f$ we can store $X = f(\_)$. This is the model used in logic programming, and in particular by Prolog.

The storage model used by logic programming has a weakness, however. This is best illustrated by a simple example. The equation $X - 3 = Y + 5$ is rejected because logic programming does not associate any meaning with $-$ or $+$ in such an equation.

The extension of logic programming to store equations involving mathematical functions was an important breakthrough. Equations involving mathematical functions are passed to the constraint store, and checked by a specialised solver. In fact not only (linear) equations but also inequations can be checked for consistency by standard mathematical techniques. It is necessary, each time a new equation or inequation is encountered, to check it against the complete set of equations encoutered so far.

[2]Solutions for $n$ queens can be generated by a deterministic algorithm, however this problem is useful for providing a simple and illuminating example of techniques which also pay off where there are no such alternative algorithms!

Linear equations and inequations are examples of primitive constraints. Thus we have an example of a *constraint store*. Further constraint stores can be built for different classes of primitive constraints, by designing constraint solvers specifically for those classes of constraints.

We use the term storage model, rather than data model, because the facility to store constraints is independent of the choice of data model - object-oriented, temporal etc. On the other hand the term storage model as used here does not refer to any physical representation of the stored information.

**Definition** *A constraint store is a storage model which admits primitive constraints of a specific class. Each new primitive constraint that is added to the store is automatically checked for consistency with the currently stored constraints.*

This definition of a constraint store specifies an equivalent operation to writing to a traditional store. However no equivalent to the read statement is specified. There are two important facilites useful for extracting information from a constraint store.

Firstly it is useful to retrieve all those constraints that involve a given variable, or set of variables. For example if the constraint store held three constraints $X \geq Z, Y \geq X, W \geq Z$ the constraints involving $X$ would be $Y \geq X$ and $X \geq Z$. However retrieving only constraints explicitly involving a variable may not give a full picture of the entailed constraints on the variable. For example the store $X \geq Y, Y \geq Z$ entails $X \geq Z$. The mechanism necessary to return all the constraints and entailed constraints on a given variable or set of variables is termed *projection*. A very useful property of a class of primitive constraints is the property that the projection of a set of primitive constraints on a given variable, or set of variables, is also expressible as a set of primitive constraints. If the primitive constraints have this property it is possible, for example, to drop a variable from the constraint store when it is no longer relevant. This is the equivalent to reclaiming the store associated with a variable in a traditional programming language when the variable passes out of scope.

Secondly it is useful to retrieve particular kinds of entailed constraints from a constraint store. For example it is very useful to know when a constraint store entails that a particular variable has a fixed value. For example the constraint store $X \geq Y, Y \geq 3, 3 \geq X$, entails that both $X$ and $Y$ have the value 3.

## 3.2 CLP($\mathcal{R}$)

The constraint logic programming scheme, written CLP(X), is a generic extension of logic programming to compute over any given constraint store. Logic programming over a constraint store has all the advantages of traditional logic programming, plus many further advantages for high-level modelling and efficient evaluation. If the constraint store holds primitive constraints from the class $X$, logic programming over this constraint store is termed $CLP(X)$. In this section we shall use a particular class of primitive constraints, linear equations

and inequations, termed $\mathcal{R}$, to illustrate the scheme. We shall use an example from (Colmerauer, 90) to illustrate how it works.

> Given the definition of a meal as consisting of an appetiser, a main meal and a dessert, and given a database of foods and their calorific values, we wish to construct light meals i.e. meals whose total calorific value does not exceed 10.

A $CLP(\mathcal{R})$ program for solving this problem is shown in figure 3.2.

```
lightmeal(A,M,D) :-
   I>0, J>0, K>0,
   I+J+K <= 10,
   appetiser(A,I),
   main(M,J),
   dessert(D,K).

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

```
appetiser(radishes,1).
appetiser(pasta,6).

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

fish(sole,2).
fish(tuna,4).

dessert(fruit,2).
dessert(icecream,6).
```

Figure 1: The Lightmeal Program in $CLP(\mathcal{R})$

A $CLP(\mathcal{R})$ program is syntactically a collection of *clauses* which are either *rules* or *facts*. Rules are as in logic programming, with the addition that they can include constraints, such as $I + J + K \leq 10$, in their bodies.

The intermediate results of the execution of this program will be descibed as computation states. Each such state comprises two components, the constraint store, and the remaining goals to be solved. We shall represent such a state as `Store @ Goals`. $CLP(\mathcal{R})$ programs are executed by reducing the goals using the program clauses. Consider the query `lightmeal(X,Y,Z).` which asks for any way of putting together a light meal. The initial state has an empty constraint store and one goal: `@ lightmeal(X,Y,Z).`

This goal is reduced using the clause whose head matches the goal. The goal is then replaced by the body of the clause, adding any constraints to the

constraint store[3]:

```
X=A,Y=M,Z=D, I+J+K =< 10, I>=0, J>=0, K>=0 @
appetiser(A,I), main(M,J), dessert(D,K)
```

The execution continues, choosing a matching clause for each goal and using it to reduce the goal. Variables which neither appear in the original goal, nor any of the currently remaining goals are projected out, as described above. A successful derivation is a sequence of such steps that reduces all the goals without ever meeting an inconsistency on adding constraints to the store. An example is:

```
X=radishes, Y=M, Z=D, 1+J+K=<10, J>=0, K>=0 @
main(M,J), dessert(D,K)

X=radishes, Y=M, Z=D, 1+J+K=<10, J>=0, K>=0 @
meat(M,J), dessert(D,K)

X=radishes, Y=beef, Z=D, 1+5+K=<10, K>=0 @
dessert(D,K)

X=radishes, Y=beef, Z=fruit  @
```

Note that at the last step the constraint $1 + 5 + 2 =< 10$ is added to the store, but it is immediately projected out.

Next we give an example of a failed derivation. The initial goal is the same as before, but this time `pasta` is chosen as an appetiser instead of `radishes`:

```
X=A,Y=M,Z=D, I+J+K =< 10, I>=0, J>=0, K>=0 @
appetiser(A,I), main(M,J), dessert(D,K)

X=pasta, Y=M, Z=D, 6+J+K=<10, J>=0, K>=0 @
main(M,J), dessert(D,K)

X=pasta, Y=M, Z=D, 6+J+K=<10, J>=0, K>=0 @
meat(M,J), dessert(D,K)
```

At the next step whichever clause is used to reduce the goal `meat(M,J)`, an inconsistent constraint is encountered. For example choosing beef requires the constraint $J = 5$ to be added to the store, but this is inconsistent with the two constraints $6 + J + K \leq 10$ and $K \geq 0$.

When the attempt to add a constraint to the store fails, due to inconsistency, a $CLP(X)$ program abandons the current branch of the search tree and

[3]Strictly all variables in the clause are renamed, but we omit this detail for simplicity.

tries another choice. In sequential implementations this is usually achieved by backtracking to some previous choice of clause to match with a goal. However there are or-parallel implementations where several branches are explored at the same time, and a failing branch is simply given up, allowing the newly available processor to be assigned to another branch.

# 4 Constraint Propagation

## 4.1 Propagating Changes

A key innovation behind constraint programming is constraint propagation. Propagation is a generalisation of data-driven computation. Consider the constraint $x = y + 1$, where $x$ and $y$ are variables. In a constraint program, any assignment to the variable $y$ (eg $y = 5$) causes an assignment to $x$ ($x = 6$). Moreover the very same constraint also works in the other direction: any assignment to $x$ (eg $x = 3$) causes an assignment to $y$ ($y = 2$).

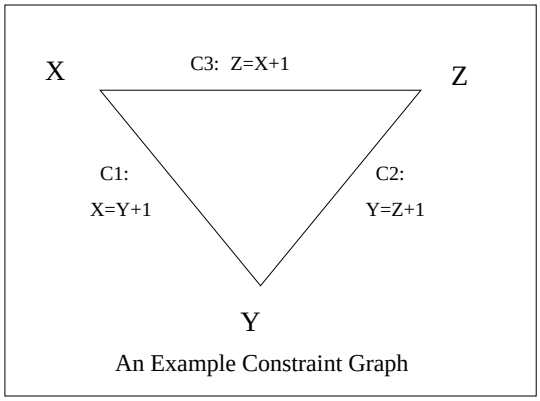
In a graphical application, constraint propagation can be used to maintain constraints between graphical objects when they are moved. For example if one object is constrained to appear on top of another object, and the end-user then moves one of the objects sideways, the other object will move with it as a result of constraint propagation.

In general each object may be involved in many constraints. Consequently the assignment of a new position to a given object as a result of propagation, may propagate further new assignments to other objects, which may cause further propagation in their turn. If each constraint between two object is represented as an edge in a graph, the propagation spreads through the connected components of the graph.

When a particular object is assigned a new position, and the change is propagated from the object to other objects, there is a causal direction. In this case we can assign a direction with each edge of the (connected component of) the graph. As long as the graph is free of cycles, the propagation behaviour is guaranteed to terminate, and produce the same final state irrespective of the order in which constraints are propagated. Efficient algorithms, such as the *DeltaBlue* algorithm, have been developed for propagation of graphical constraints. They work by firstly generating the directed graph whenever an object is moved, and then compiling this directed graph into highly efficient event-driven code.

However if the graph contains cycles both these issues arise. Consider, as a simple example, the three constraints C1, C2 and C3 specified thus - C1: $x = y + 1$, C2: $y = z + 1$ and C3: $z = x + 1$.

Assigning $y = 3$ may start a non-terminating sequence of propagations cycling through the constraints C1 (which yields $x = 4$), then C3 (which yields $z = 5$), then C2 (which yields $y = 6$) and then C1 again and so on. Alternatively the same assignment $y = 3$ could propagate through C2 yielding $z = 2$, thence



An Example Constraint Graph

$x = 1$ and $y = 0$ via C3 and C1. In this case the propagation also goes on for ever, but this time the values of the variables decrease on each cycle! Thirdly the same assignment $y = 3$ could yield $z = 5$ via C1 and C3, and $z = 2$ via C2!

In this example the original constraints are, logically, inconsistent. However similar behaviour can occur when the constraints are consistent. In the following example there are constraints on three variables. C4: $x + y = z$, C5: $x - y = z$. Suppose the initial (consistent) assignments are $x = 2, y = 0, z = 2$. Now a new assignment is made to $z$: $z = 3$. If constraint propagation on C4 yields $y = 1$, then propagation on C5 yields $x = 4$. Now propagation on C4 and C5 can continue for ever, alternately updating $y$ and $x$.

Propagation algorithms have been developed which can deal with cycles, but if the class of admissible constraints is too general, it is not possible to guarantee that propagation is confluent and terminating.

## 4.2 Active Constraints

### 4.2.1 Propagating New Information

The changes propagated by label propagation, as discussed in the previous section, are variable assignments as held in a traditional program store. However in this section we explore the application of constraint propagation to constraint stores, which maintain partial information about the program variables expressed as primitive constraints. Using a constraint store, it is possible to develop a quite different model of computation in which the store is never destructively changed by propagation, but only augmented. One great advantage of this combination is that confluence properties are easy to establish, and consequently there is little need for the programmer or applications designer to know in what order the propagation steps take place.

### 4.2.2 Constraint Agents

We have encountered two very different kinds of constraints. Primitive constraints are held in a constraint store, and tested for consistency by a constraint solver. On the other hand propagation constraints actively propagate new information, and they operate independently of each other. Propagation constraints are more commonly called *constraint agents*. The behaviour of a constraint agent is to propagate information to the underlying store. In case the underlying store is a constraint store, the information propagated is expressed as primitive constraints.

Constraint agents are processes that involve a fixed set of variables. During their lifetime they alternate between suspended and waking states. They are woken from suspension when an extra primitive constraint on one or more of their variables is recorded. Sometimes, after checking certain conditions, the woken agent simply suspends again. Otherwise the agent exhibits some active behaviour which may result in new agents being spawned, new primitive constraints being added to the store, or an inconsistency being detected (which is equivalent to an inconsistent constraint being added to the store). Subsequently the agent either suspends again, or exits, according to its specification.

### 4.2.3 Some Built-in Constraint Agents

The simplest constraint agent is one which adds a primitive constraint to the constraint store and then exits. The most fundamental example is assigning a value to a variable, eg. `X=3`. This agent adds $X = 3$ to the constraint store and exits.

The next two examples are disequality constraints, which will be illustrated in the next section. The first disequality constraint is invoked by the syntax `X~=Y`. This agent does not do anything until both $X$ and $Y$ have a fixed value. Only when the primitive constraints in the store entail $X = val_x$ and $Y = val_y$ for some unique values $val_x$ and $val_y$, does the agent wake up. Then its behaviour is to check that $val_x$ is different from $val_y$. In case they are the same, an inconsistency has been detected.

If the constraint store holds finite domain constraints, then the more powerful constraint agent invoked by the syntax `X ## Y` can be used. This agent wakes up as soon as either $X$ or $Y$ has a fixed value. It then removes this value from the finite domain of the other variable and exits.

## 4.3 Map Colouring

### 4.3.1 The Map Colouring Program

As a toy example let us write a program to colour a map so that no two neighbouring countries have the same colour. In constraint logic programs, variables start with a capital letter (eg `A`), and constants with a small letter (eg `red`).

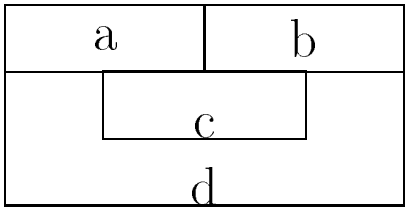



Figure 2: A Simple Map to Colour

A generic logic program, in Prolog syntax, that tries find possible ways of colouring this map with only three colours (red, green and blue) is in figure 3.

```
coloured(A,B,C,D) :-
    ne(A,B), ne(A,C), ne(A,D), ne(B,C), ne(B,D), ne(C,D),
    chosen(A), chosen(B), chosen(C), chosen(D).

chosen(red).
chosen(green).
chosen(blue).
```

Figure 3: A Generic Logic Program for Map Colouring

In this program the (as yet undefined) predicate *ne* constrains its arguments to take different values. We will show how different definitions of *ne* cause the program to behave in different ways.

The predicate *chosen* can be satisfied in three ways. At runtime the system tries each alternative in turn. If a failure occurs later in the computation, then the alternatives are tried in a last-in first-out basis.

The first definition of *ne* uses the original disequality of Prolog:
`ne(X,Y) :- X\=Y.`
If invoked when either of its arguments are uninstantiated variables, `X\=Y` simply fails. To avoid this incorrect behaviour it is possible to place the constraints after all the choices. In this case the program correctly detects that there is no possible colouring after checking all 81 alternatives.

A more efficient Prolog program can be written by interleaving choices and constraints, but this requires the programmer to think in terms of the operational behaviour of the program on this particular map. The same effect can be achieved much more cleanly by using the above program with a new definition:
`ne(X,Y) :- X~=Y.`
This disequality predicate *delays* until both arguments have a fixed value. It then immediately wakes up and fails if both values are the same. If the values are different it succeeds. This program detects that our map cannot be coloured

with three colours after trying only 33 alternatives.

Another disequality constraint is available in CLP, which assumes its arguments have a finite set of possible values. We can use it by defining
`ne(X,Y) :- X##Y.`
This disequality delays until one of its arguments has a fixed value. This value is then removed from the set of possible values for the other. To obtain the advantage of `X##Y` it is necessary to declare the set of possible values for each variable, by writing `[A,B,C,D]::[red,green,blue]`.

```
coloured(A,B,C,D) :-
    [A,B,C,D]::[red,green,blue],
    ne(A,B), ne(A,C), ne(A,D), ne(B,C), ne(B,D), ne(C,D),
    chosen(A), chosen(B), chosen(C), chosen(D).

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

chosen(red).   chosen(green).  chosen(blue).
```

Figure 4: A Finite Domain CLP Program for Map Colouring

This program detects that the map cannot be coloured after trying only 6 alternatives.

Although this example is so trivial that it is quite simple to solve it in Prolog, the CLP program scales up to complex maps and graphs in a way that is impossible using a programming language without constraints.

## 4.4 Building Constraint Agents

### 4.4.1 Guards

Constraint agents can be built by directly defining their waking behaviour using the notion of a "guard". As an example we take a resource constraint on two tasks, $t1$ with duration $d1$ and $t2$ with duration $d2$ forcing them not to overlap. The variable $ST_1$ denotes the start time of $t1$ and $ST_2$ denotes the start time of $t2$. Suppose we wish to define the agent constraint $agent(ST_1, ST_2)$ thus: if the domain constraints on the start time of $t1$ and $t2$ prevent $t1$ from starting after $t2$ has finished, constrain it to finish before $t2$ has started.

This behaviour can be expressed as follows:

```
agent(ST_1,ST_2) <==>              % agent name and parameters
        ST_1 #< ST_2+d2  |         % guard
            ST_1+d1 #<= ST2        % body
```

The guard will keep the agent suspended until the domains of $ST_1$ and $ST_2$ are reduced to the point that the inequation $ST_1 \leq ST_2 + d2$ holds for every possible value of $ST_1$ and $ST_2$. When this is true, it wakes up and executes the body, invoking a new agent `ST_1+d1 #<= ST2`. Of course this guard may never become true, in case task $t2$ runs before task $t1$. To cope with this alternative we add another guard and body, yielding the final definition:

```
agent(ST_1,ST_2) <==>              % agent name and parameters
        ST_1 #< ST_2+d2  |         % guard1
            ST_1+d1 #<= ST2  ;     % body1
        ST_2 #< ST_1+d2  |         % guard2
            ST_2+d2 #<= ST1        % body2
```

This agent wakes up as soon as either of the guards are satisfied, and executes the relevant body. As soon as one guard is satisfied, the other guard and body are discarded.

#### 4.4.2 Agents Defined by Specific Codes

Constraint programming systems implement their built-in agents using specific codes which wake up, for example, whenever the upper or lower bound of the domain constraint on a given variable is altered.

Using specific codes it is also possible to build complex constraints and constraint behaviours to obtain good performance on large complex problems. Indeed this is the approach that has been very successfully applied on job-shop scheduling benchmarks and incorporated into commercial constraint programming tools such as CHIP and ILOG SCHEDULE.

# 5 Implementation and Applications

## 5.1 Constraints Embedded in A Host Programming Language

### 5.1.1 Control

A constraint programming language is the result of embedding constraints into a host programming language. The host program sends new constraints to the constraint handler, under program control. The information that can be returned to the host program depends on how tightly the constraints are embedded in the host language. Most basically the constraint handler can report consistency or inconsistency. Given a closer embedding it can also return variable bindings. Most closely it might allow the host programming language to be extended with guards and other annotations, so as to allow host program statements to be suspended and woken up like other constraint agents.

We can diagram the behaviour of a constraint program as follows:

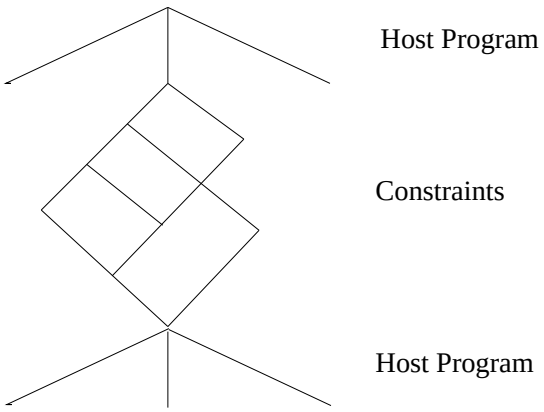



Figure 5: Control in Constraint Programming

The diagram shows three successive phases occurring during program execution.

In the first phase the host program is executing under explicit program control. The host program performs such tasks as input/output, event handling, and search. It may execute for some time before finally sending a constraint to the handler. The diagram illustrates the host program performing search over three branches. For the purposes of the diagram, it does not matter how these branches are explored (sequentially, or in parallel) and how they are expressed (by recursion over a set of alternatives, or by non-deterministic choice and backtracking). The succeeding phases of the execution are only shown for the second branch.

In the second phase the constraint handler is executing, and its control is constraint-driven. The constraint handler only becomes active when it receives a new constraint from the host program. The behaviour of the constraint handler (represented in the diagram as a network of thick lines) is defined by a set of atomic behaviours (each of which is represented by a single arc in the network). An atomic behaviour is the posting of a new constraint to the store, or a single propagation step performed by a constraint agent, or the invocation of the body in a guarded constraint. When no more constraint propagation is possible, the constraint handler returns to the host program with success, and the host program resumes control, as illustrated in the third phase of the above diagram.

### 5.1.2 Concurrency

The challenge for embedding constraint handling in a host programming language is to deal with constraint agents acting concurrently. Assuming the pro-

grammer has little or no control over the waking and resuspending of constraint agents, the constraint programming framework must ensure that the final result of constraint propagation, before the host program resumes control, is independent of the order in which the agents wake up and post new basic constraints to the constraint store. The concurrent constraints framework (Saraswat, 1993) enables this condition to be met for large classes of practically useful constraint agents.

### 5.1.3 Logic Programming as a Host Language

Constraints fit hand in glove with declarative host programming languages. Three of the most influential constraint programming languages were embed them in Prolog, CLP(R) and CHIP. Whilst all three system are still developing further, there are many new constraint programming systems emerging including ECLiPSe, Oz, 2LP, and Newton.

From a theoretical point of view the extension of logic programming to *Constraint Logic Programming* (CLP) has been very fruitful. A good survey is (Jaffar and Maher, 1994). For example ALPS - a form of logic programming with guards - was an extremely influential language, becoming the forerunner of the *Concurrent Constraints* paradigm (Saraswat, 1993). Concurrent constraint programming has in turn provided a very clean model of concurrent and multi-agent computing. Constraints can also be modelled in terms of information systems, which allows us to reason about the behaviour of constraint programs at an abstract level.

### 5.1.4 Libraries for Embedded Constraint Programming

The constraint programming technology has matured to the point where it is possible to isolate some essential features and offer them as libraries or embedded cleanly in general purpose host programming languages.

For example isolating constraints as libraries has made possible the development of sophisticated constraint-based scheduling systems, see (Zweben and Fox, 1994). More generally there are commercially available libraries supporting constraint handling such as the CHIP and ILOG C++ constraint libraries.

## 5.2 Applications of Constraint Programming

Based on a few constraint programming languages which support the storage of basic constraints and the waking and resuspension of constraint agents, the technology has achieved a number of remarkable successes on benchmarks and, more importantly, real industrial applications. A recent survey of practical applications of constraint programming (Wallace, 1996) estimated the annual revenue from constraint technology at around 100 million dollars per annum.

One early application, developed in 1990, was to container port planning in Hong Kong. The application was built by ICL, using finite domain constraints. Another early user was Siemens, who have applied Boolean constraints to problems of circuit design and integration. Both Siemens and Xerox are now applying constraints to real time control problems.

Constraints are used for graphical interface design and implementation at Object Technology International. Constraint-based scheduling has made a big impact in the USA, with applications in heavy industry, NASA and the Army. The application developers are typically small companies such as Recom, Red Pepper and the Kestrel Institute.

One company, ILOG, has sold constraint technology both in the USA and Europe. ILOG also have applications in south east Asia. Its French rival, Cosytec, is perhaps the only company to make all its business from constraint technology and applications. (Cras, 1993) gives a survey of industrial constraint solving tools.

Areas where constraint programming has proven very successful include scheduling, rostering and transportation. Constraints are used for production scheduling in the chemical industry, oil refinery scheduling, factory scheduling in the aviation industry, mine planning and scheduling, steel plant scheduling, log cutting and transportation, vehicle packaging and loading, food transportation scheduling, nuclear fuel transportation planning and scheduling, platform scheduling, airport gate allocation and stand planning, aircraft rescheduling, crew rostering and scheduling, nurse scheduling, personnel rostering, shift planning, maintenance planning, timetabling, and even financial planning and investment management.

There is a regular conference on the Practical Applications of Constraint Technology, presented on `http://www.demon.co.uk/ar/PACT97/index.html`

# 6 Current Developments

## 6.1 Constraints in the Computing Environment

Naturally there is a great deal of useful research exploring ways of using constraints in an object oriented programming environment, in databases, and on the internet. The field of constraint databases, in particular, has thrown up a growing community of researchers who are exploring the theoretical and practical possibilities of storing constraints in databases, imposing constraints on databases, and retrieving constraints from databases. This work is starting to be noticed in the field of geographical information systems. There is a growing need for databases to handle space, in two and three dimensions, and time. Examples are environmental monitoring and protection, air traffic control, and reasoning about motion in three dimensions. The constraint database technology appears to address these requirements.

## 6.2 Mixed Initiative Programming

One of the great bugbears of constraint programming is how to deal with over-constrained problems. As Jean-Francois Puget put it "What solution do you return when there are no solutions?". The traditional approach in mathematical programming is to associate a penalty with each violated constraint, and seek the solution which minimises the total penalties.

A related approach is to decide between the different constraints which ones are more important than which others. The constraint program then only enforces a constraint if this does not cause a more important constraint to be violated.

The drawback is that it is not easy for the user to estimate the importance of a constraint, and the solution produced by the software may well not be the best solution in the opinion of the end users of the system. Moreover this approach is a black box, and the end users receive no feedback about "nearby" solutions, which might prove better on the ground.

Accordingly one current area of research is how to help the end user solve overconstrained problems, and multi-criteria optimisation problems which have different, and possibly conflicting, optimisation criteria. The challenge is to allow the end-user to explore the solution space interactively, eliciting information about solutions, and potential solutions, which enables the user to choose the very best solution for his or her purposes. This is called mixed initiative programming.

## 6.3 Interval Reasoning

Intelligent software systems have often been highly specialised for symbolic computation, but weaker on numeric computation. This is one reason why the combination of symbolic constraint solving and mathematical programming are proving to be so interesting both in theory and practice.

One recalcitrant problem for numeric computation is the problem of numeric instability. Under certain, unlikely, circumstances, tiny rounding errors in the basic mathematical routines can unexpectedly cause serious errors in the final result. The difficulty is that these errors are hard to predict, and no practical way has been found to predict them.

A way to contain this instability is to reason on numeric intervals, instead of numbers, ensuring that at each calculation the interval is rounded out so as to ensure the actual solution lies inside it. Unfortunately these intervals tend to grow too wide to be useful. Recently, however, using intervals as primitive constraints in a constraint programming framework some excellent results have been obtained for well-known mathematical benchmarks. These results compete with the best mathematical programming approaches, in particular when the input intervals are quite wide.

Intervals appear in a multitude of different contexts as a way of approximating values. In particular they are used in database indexing, in constraint propagation, and for specifying uncertainty.

The author predicts that interval constraints will play a crucial role in spatial and temporal databases and in the handling of uncertainty.

### 6.4 Stochastic Techniques

Organisations are increasingly able to capture an up-to-date picture of their global resources, and they are seeking to optimise their use of these resources. However for large organisations this optimisation problem is unmanageable: no algorithm could ever find the guaranteed best solution for the whole organisation.

Stochastic techniques are a way of exploring very large solution spaces and finding good solutions even when it is only possible to visit a (vanishingly!) small proportion of the solutions. Well-known techniques include simulated annealing, genetic algorithms and tabu search. A drawback is that for structured problems, where constraints impose complex dependencies between different parts of the solution, stochastic techniques are not able to enforce these constraints.

Recently researchers have begun to explore ways of embedding constraint propagation in stochastic algorithms, thus ensuring that the solutions visited by the algorithm satisfy the problem constraints. To date such hybrid algorithms have been rather loosely coupled. For example the stochastic technique only works on a small subset of the problem variables, producing skeleton solutions. These are then fleshed out using constraint handling techniques, and the cost of the resulting full solution is calculated, and fed back to the stochastic algorithm which generates another skeleton solution.

Tightly integrated algorithms combining techniques from mathematical programming, constraint programming and stochastic algorithms are now the vision of a growing research community. These algorithms may still not be the "golden bullet" that cuts through all forms of complexity, but they would certainly represent an important step in the right direction!

## References


A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.

J.-Y. Cras. A Review of Industrial Constraint Solving Tools ISBN 1-898804-001, AI Intelligence, 1993.

M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving large scheduling problems in logic programming. In *EURO-TIMS Joint International Conference on Operations Research and Management Science*, Paris, July 1988.

T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint logic programming: An informal introduction. Technical Report ECRC-93-5, ECRC, 1993.

J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL'87: Proceedings 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, January 1987. ACM.

J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

V. A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. MIT Press, Cambridge, MA, 1989.

M. G. Wallace. Practical applications of constraint programming. *Constraints*, 1(1), 1996.

M. Zweben and M.S. Fox, editors. *Intelligent Scheduling.* Morgan Kaufmann, 1994.