Constraint Logic Programming

Helmut Simonis
COSYTEC SA
4, rue Jean Rostand
F-91893 Orsay Cedex
France
simonis@cosytec.fr

1. Abstract

In this tutorial we give an overview of constraint logic programming (CLP), a combination of two declarative programming paradigms, logic programming and constraint solving. We present an informal introduction to different constraint solving methods used in CLP systems, with the emphasis on finite domain constraints. In addition we look at different ways to express heuristics as part of CLP programs and discuss problem solving methodology. We also give an overview of different CLP systems currently available and present some of their application domains, with special emphasis on industrial applications for production planning, scheduling and resource allocation.

2. Introduction

In the last ten years, constraint logic programming has emerged as a very interesting sub field of logic programming. CLP aims at combining the declarative aspects of logic programming and constraint solving in an efficient problem solving environment. Constraints over different domains can be stated in a uniform framework and are solved with methods originating in various areas, from artificial intelligence (AI) to Operations Research (OR).
CLP is now used industrially for applications as diverse as digital circuit design, portfolio management and production scheduling systems and there are several industrial systems available for constraint programming.

2.1 Structure of document

The tutorial is structured in the following way. We start with a section on the motivation for using constraints, discussing typical problems with conventional program development for complex decision support systems. We then briefly review some of the techniques which are used inside CLP. It combines methods from logic programming and constraint solving, as well as advanced techniques from Operations Research and finite mathematics. We also give some pointers to the theoretical aspects of constraint logic programming.
In section 5, we introduce a classification of different constraint solving methods, which we will use later on. This classification distinguishes complete and incomplete solvers, based on syntactical or semantic methods. In the next section, we present an overview of different constraint solving methods which are currently applied in CLP systems. The computation domains handled by these solvers are quite diverse, including Boolean algebra, linear programming over rational numbers, finite domains or list and set handling. As it is the most interesting domain from a practical point of view, we present the finite domain constraint solver in more detail in section 7.

There are many different types of finite domain constraints besides the basic arithmetic ones. In recent years new types of very high level, global constraints have been introduced, which we discuss in section 7.2.2. Another important aspect of finite domain constraints is the possibility to control the solution process with an enumeration scheme. Heuristics and strategies can be expressed in different ways. We present some alternatives in section 7.3.
We then come to an overview of different constraint solving tools, where we also briefly discuss the history of the various constraint solvers.
In the last part, section 9, we give an overview of industrial applications which have been developed using CLP tools. Most of these applications are from the scheduling and planning areas, with some notable exceptions in other domains. At the end we also talk about some more in-depth material, which could be used for study after this tutorial.

3. Interesting Problems

In this section we discuss the motivation for the use of constraint logic programming. We explain which type of problem is best suited for the CLP approach and where these problems occur in practice. They share a set of characteristics, which make them very hard to tackle with conventional problem solving methods.
Combinatorial problems occur in many different application domains. We encounter them in Operations Research (for example scheduling and assignment), in hardware design (verification and testing, placement and layout), financial decision making (option trading or portfolio management) or even biology (DNA sequencing). In all these problems we have to choose among many possible alternatives to find solutions respecting a large number of constraints.
We may be asked to find an admissible, feasible solution to a problem or to find a good or even optimal solution according to some evaluation criteria.
From a computational point of view, we know that most of these problems are difficult. They belong to the class of NP-hard [GJ79] problems. This means that no efficient and general algorithms are known to solve them.
At the same time, the problems themselves are rapidly changing. For example, in a factory, new machines with new characteristics may be added, which might completely change the production scheduling problem. New products with new requirements will be added. If a scheduling system can not be adapted to these changes, it will rapidly become useless.
Another aspect is that the complexity of the problems is steadily increasing. This may be due to increased size or complexity of the problem, or may be caused by higher standards on quality or cost effectiveness. Many problems which have been handled manually for a long time now exceed the capacity of a human problem solver.
At the same time, humans are typically very good at solving these complex decision making problems. They can have a good understanding of the underlying problem and often know effective heuristics and strategies to solve them. In many situation they also know which constraints can be *relaxed* or ignored when a complete solution is not possible. For this reason, it is necessary to build systems which co-operate with the user via friendly graphical interfaces and which can incorporate different rules and strategies as part of the problem solving mechanism.
Developing such applications with conventional tools has a number of important drawbacks.

- The *development time* will be quite long. This will increase the cost, making bespoke application development very expensive.
- The programs are *hard to maintain*. Due to their size and complexity, a change in one place may well lead to a number of other changes which are required in other places.

- The programs are *difficult to adapt* and to extend. As new requirements arise during the life cycle of the program, changes become more and more difficult.

The use of constraint logic programming should lead to a number of improvements:

- The *development time is decreased*, as the constraint solving methods are reused.
- A declarative problem statement makes it *easier to change* and modify programs.
- As constraints can be added incrementally to a system, *new functionality can be added* without changing the overall architecture of the system.
- Strategies and *heuristics can be easily added* or modified. This allows to include problem specific information inside the problem solver.

4. Background Techniques

We will now review some of the background techniques which are used inside CLP. Due to space limitations we will restrict ourselves to the main concepts. Further material can be found in the references. In Figure 1, we show the basic influences on constraint logic programming

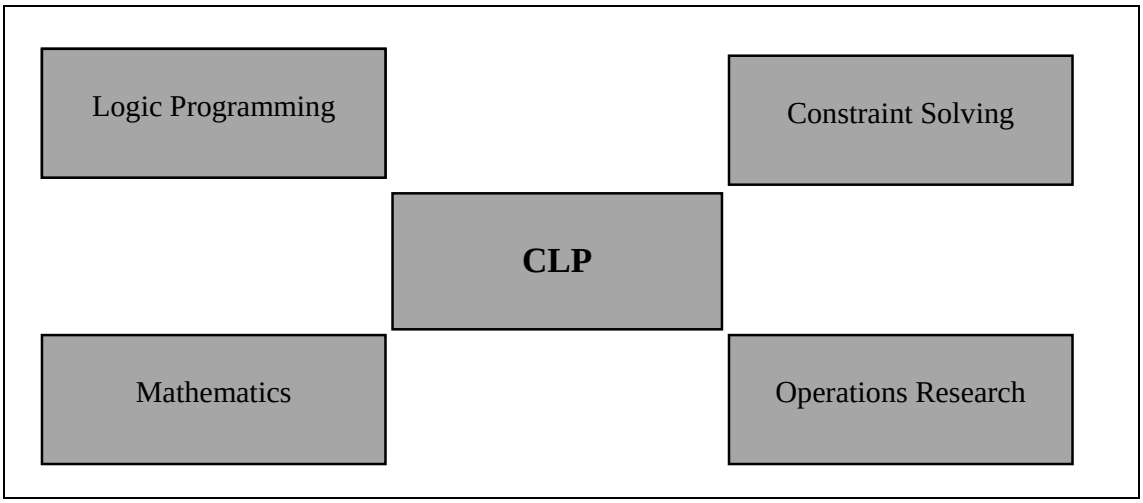


Figure 1: Techniques behind constraints

In the next paragraphs, we will discuss these different influences.

4.1.1 Logic Programming

Logic programming is based on the idea that (a subset of) first order logic can be used for computing. The first logic programming language PROLOG was created by A. Colmerauer in 1972. Other important influences come from the work of J. A. Robinson on unification and of R. Kowalski on SL-resolution. Originally intended for natural language processing, PROLOG is now used as a general purpose language.
A basic idea in logic programming is the separation of *logic* and *control* [Kow79]. Programming in logic is concerned first with the correct statement of a problem. The program should state what is true about the problem, not how to solve it procedurally. Efficiency of the program should come second, and ideally should be handled automatically by the

language itself. This separation of concerns helps to develop correct programs more easily and makes it possible to change and extend existing programs without rewriting large parts.
Three other fundamental concepts of logic programming are also very important for constraint logic programming. They are

- *relational form* which allows to define predicates without fixing input and output arguments a priori
- *non determinism* which includes a tree search procedure (backtracking) as part of the computation mechanism
- *unification* which solves equations of (un-interpreted) symbolic terms

While logic programming offers a big advantage over conventional programming in terms of simplicity, it also encounters some difficulties for solving the type of complex problems we are interested in.

- The built-in search procedure is based on the *generate and test* paradigm, which becomes very inefficient for complex problems. Solutions are generated by non-deterministic choice, and then tested in a deterministic way.
- The basic data structures in Prolog are un-interpreted (Herbrand) terms. These symbolic terms are not specialised for the particular computation domains encountered in applications. There is a large conceptual gap between the problem to be solved and the expression of the problem in logic programming.
- Unification and resolution are general, but rather weak inference mechanisms. More advanced, specialised methods exist in other areas for problem solving. Equation solving methods from mathematics or consistency techniques from constraint solving are examples of such more powerful methods.

4.1.2 Constraint Handling

Constraint handling techniques are a well known method in the artificial intelligence area. Early traces can be found in work on computer graphics by Sutherland [Sut63] and Borning [Bor81]. They are also used by Sussman and Steele [SS80] for circuit analysis or by Davis [Dav84] for circuit diagnosis. Another early use of constraints is in computer vision by Waltz [Wal72]. More general constraint handling systems are for example REF-ARF [Fik70] and ALICE [Lau78]. The basic idea is the declarative expression of problems as constraints and specialised constraint solvers over some restricted domains (numbers, finite sets) to efficiently find solutions to the constraints. A common technique in these solvers is local propagation or demon based computation [SS80]. A problem is expressed by a network of constraints. As soon as some information becomes available at some point in this network, constraint demons are woken which try to propagate this information through the network. This may wake up other constraints and so on.
Consistency techniques form a important subclass of constraint handling methods. They have been introduced in [Mon74] [Mac77] [MF85] [HE80] to express problems over finite domains. Problems are modelled with two components, variables, which can range over finite sets of possible values and constraints, which are relations between the variables. Different techniques, for example arc- and path- consistency, are used to remove inconsistent values from the variable domains until solutions are found. An introduction to this field can be found in [Tsa93] [FM94]. While consistency techniques present a good theoretical framework for the analysis of different constraint solving methods, they are rarely used for problem solving

as standalone procedures. Constraint techniques must be embedded in a programming system which allows easy constraint manipulation, as well as the combination with heuristics and strategies.

4.1.3 Operations Research

A large number of techniques from Operations research are used in CLP systems. The constraint solvers for linear constraints over rational numbers are based on linear programming techniques and the Simplex algorithm. The new global constraints [AB93] [BC94] over finite domains as well use techniques from scheduling and placement to check consistency. Some properties of OR methods pose problems when these methods are used inside CLP:

- Most OR systems are build for specialised, particular problems. This way mathematical properties of the complete problem can be exploited to improve the solution.
- Normally, all constraints to be included are known when the system is set up. The solution methods do not have to be incremental.
- The control flow can be rearranged to minimise the effect of rounding errors in order to achieve numerical stability.

For that reason, the techniques used in CLP systems often employ custom variants of well known algorithms, which must satisfy the following conditions:

- They must be *incremental*, i.e. constraints can be added one by one without requiring the solution process to start again from scratch.
- The methods must be *general*. Adding new constraints should not invalidate the preconditions of a method. This means that algorithms for particular problems will be normally not useful in CLP systems.
- Since the control flow in a constraint solver is quite complex and completely data dependent, special care must be taken for achieving *numerical stability*. A typical case is the use of exact arithmetic for linear constraints [Col90] [DVS88a] or for correct, conservative interval arithmetic in some constraint solvers [OV90].

4.1.4 Mathematics

CLP also uses a number of techniques from particular mathematical areas. Some of the problem solvers for Boolean problems for example use classical results from Boolean algebra [NM89]. Other results from finite mathematics and graph theory are used for global constraints [BC94]. The remarks made for OR techniques also apply here. We must look for incremental and general methods which can be combined in different ways inside the constraint solvers.

4.2 Theoretical Issues

Similar to logic programming, we can define different semantics for constraint logic programming. A mechanism for defining this semantics is given in the CLP(X) [JL87] scheme. This scheme allows to obtain general results on correctness and completeness for a class of languages CLP(X), which are parametrized in the constraint solver *X* used in a particular instance. Different constraint languages can be defined by defining sets of basic constraints and their constraint solver. A comprehensive overview of theoretical issues is given in [JM94].

5. Constraint Solving Classification

We now describe a classification of constraint solving mechanisms which we will use later on to distinguish between constraint solvers. Constraint solvers can differ in various aspects. A first criteria is the use of different computation domains. The computation domain defines the objects over which constraints can be expressed. We will present possible domains and discuss some considerations in the choice of a computation domain. Another important aspect of constraint solvers is the amount of propagation that they provide. We have to explain the basic notions of complete or incomplete constraint solvers. A third classification criteria distinguishes the techniques which are used for constraint solving for different constraints, which may be based on syntactic (domain independent) or semantic (domain dependent) methods.

5.1 Choice of Computation Domains

Constraints can be expressed over a variety of different domains. As a general rule, three conditions [DSV87] should be satisfied for a CLP language over a computation domain:

- Constraints can be handled in a deterministic way without loosing completeness.
- Efficient constraint solving methods exist for the domain.
- The computation domain can be used for more than one application.

A number of computation domains have been identified which satisfy these conditions, and which are now used in one or several CLP systems. They are

Finite domains
Finite domain constraints [DSV87] [VH89] are expressed over variables which range over a finite set of possible values. Often, the domains are expressed as bounded subsets of the natural numbers. Constraints may be arithmetic or symbolic. Complex, global constraints have been added in the last years.

Linear arithmetic terms
Constraints over linear arithmetic terms [Col90] [DVS88a] [JMS90] can be handled efficiently by Gaussian elimination and the Simplex algorithm. Several systems allow the use of rationals, while other handle floating point numbers.

Boolean terms
Equality constraints over Boolean terms can be handled by Boolean unification [NM89] [BS87]. Since any complete solver has worst case exponential complexity, other, incomplete solvers are popular for certain applications.

Pseudo Boolean Constraints
Arithmetic constraints over 0/1 variables with integer coefficients are called Pseudo Boolean constraints [Boc93] [Bar94]. Different methods from the Boolean domain generalise to this case, but there are also other methods originating in OR.

Intervals
Equality and inequality constraints over intervals [OV90] [BMV94] can be handled with bound propagation.

Non-linear arithmetic

Non linear arithmetic constraints [ASS88] [Hon93] over real numbers can be handled by different constraint solvers. These constraint solvers use different results from computer algebra and mathematics.

Sets
Several different constraint solvers [Ger94] [LL91] over sets (or multi-sets) have been proposed. Constraints include quality, membership or set inclusion.

Lists
Equality constraints over finite lists or sequences [Col90] [LL91] can be handled in an efficient manner.

Some other computation domains have been proposed, but are not yet widely used for problem solving.

We will return to the different computation domains in section 6.

5.2 Complete/Incomplete

The amount of propagation performed by a constraint solver is an important classification criteria. Some constraint solvers are called *complete*. This means that they can decide the satisfiability of any set of constraints over the computation domain. This clearly is a very desirable property. At each step of the computation, we can decide whether the current constraint set still has a solution. Unfortunately, satisfiability of a constraint set may be a undecidable problem in some domains or may have exponential complexity. The later is the case for example for Boolean terms or finite domains. This restricts complete solvers to either relatively well structured domains like linear arithmetic terms or to special purpose solvers like the complete Boolean solvers.
Incomplete solvers can be defined for most application domains. They may of course differ in the amount of propagation which is performed on the constraints.

5.3 Syntactic/Semantic

Another difference between constraint solvers is the use of syntactic or semantic methods for solving.
Syntactic methods use constraint simplification and rewriting for constraint propagation. Most basic constraint solvers use syntactic methods like variable elimination or bound propagation. These methods are domain independent and can be used in similar form for different domains.
Semantic methods use methods particular to the computation domain. These methods often require insights to the mathematical or structural properties of the domain. The handling of global constraints in CHIP [AB93] [BC94] uses semantic methods obtained from Operations Research, finite mathematics and graph theory to deduce better results than those obtained by the syntactic methods of the primitive constraints.
In general, for one computation domain different combinations of complete/incomplete solvers using syntactic or semantic methods are possible. We will present some examples in the next sections.

6. Constraint Solving Methods

In this section we want to give an idea of how the different constraint solvers in CLP systems work. For examples, we mainly use the form of constraints in the CHIP system. Other CLP languages use slightly different syntax.
Figure 2 shows an overview of the methods which will be discussed. According to their importance to practical problems, the finite domain solver and to a lesser extend the rational and Boolean solvers will occupy more space in the presentation.

6.1 Complete Solvers

We start with a description of some complete solvers. A complete solver can decide the consistency of any set of constraints over its domain. In order to improve efficiency, these solvers work incrementally. When a new constraint is added, the constraint set is not resolved completely, but only updated to check consistency. For this purpose, the constraints are internally kept in a *solved form*, a representation of the constraints in a normal form.
All complete solvers which we discuss here use syntactical methods based on term manipulation and constraint simplification.

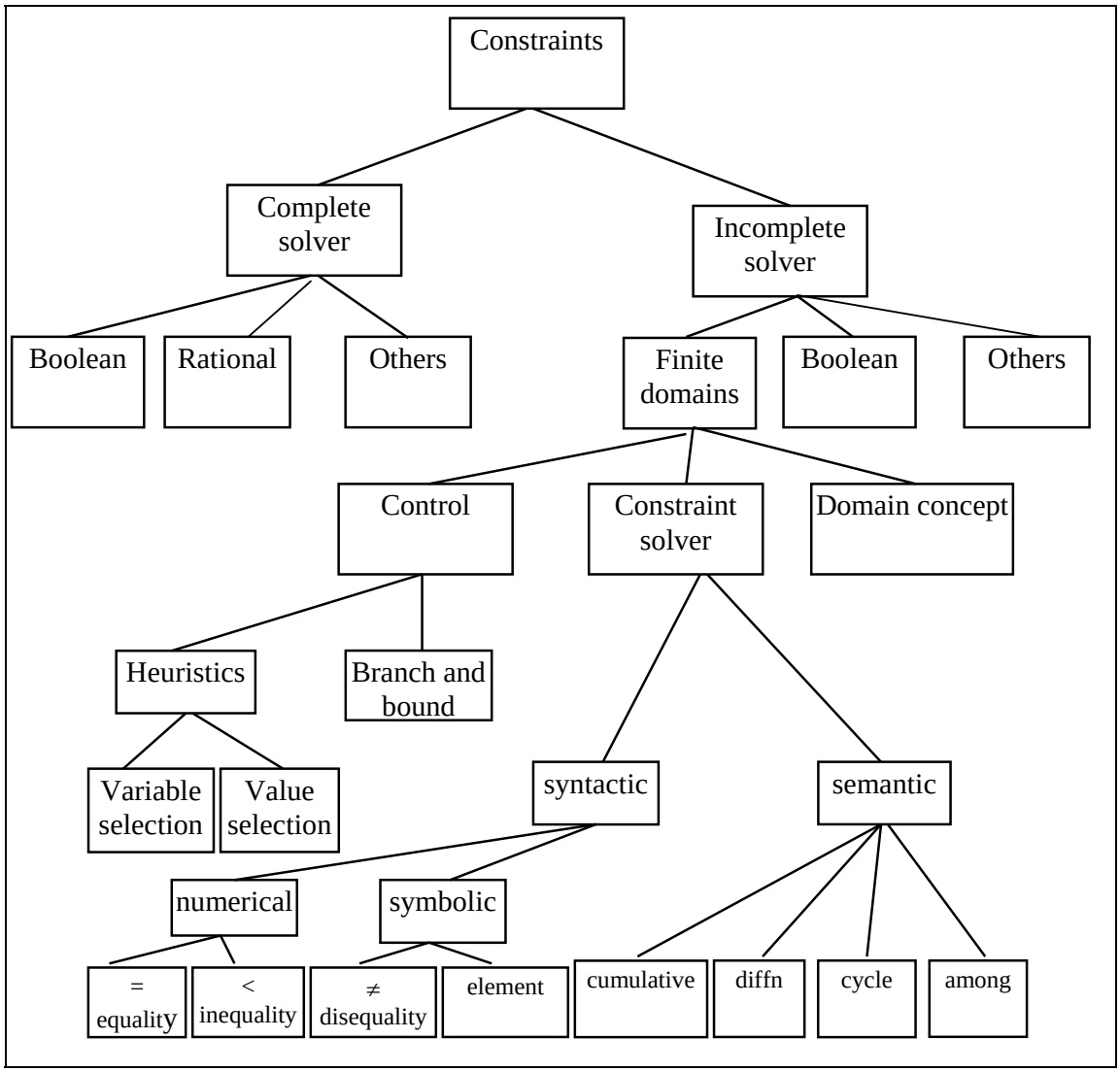


Figure 2: Constraint solving overview

6.1.1 Rational

The rational constraint solver [Col90][DSV88a] provides a complete solver over a continuous domain. Rational numbers instead of real numbers are used since they offer exact representation and arithmetic and since the rationals are closed under linear equalities and inequalities. If floating point numbers are used instead [JMS90], the correctness of the constraint system becomes compromised due to rounding and representation errors. A possible alternative is the use of intervals in a complete solver. This seems to be computationally expensive [CL94].

6.1.1.1 Domain

Within the rational computation domain we can express constraints on linear terms over rational numbers. We can handle equality and inequality constraints as well as disequality constraints. Minimisation or maximisation of a term is possible with some meta-logical constructs. In CHIP, the primitives *rmin X* and *rmax X* are used.

6.1.1.2 Methods

The constraint solver for rational terms is based on *Gaussian elimination* and the *Simplex algorithm*. Constraints are transformed in a standard form based on a symbolic representation of arithmetic terms. As the number of constraints and variables, and therefore the number of lines and columns in the Simplex tableau are not known a priori we can not use a matrix representation. Inequalities are converted into equality constraints by introducing non-negative *slack* variables. These additional variables can also occur in the solution. Different alternatives for the solved forms exist [VG90].
The handling of *disequality* constraints requires some extra work. Basically they are delayed until enough information is available. In some systems, non-linear constraints can be handled in a similar way. Their execution is delayed until the constraint becomes linear [JMS90].
The Simplex method has a worst case exponential complexity, but this is rarely observed in practice. Alternatively, other linear programming methods like the *interior point methods* with polynomial complexity could be adapted as a CLP solver.

6.1.1.3 Example

This example is taken from [Col90]. The problem is to compute the yearly instalments in order to reimburse a capital borrowed from a bank. We suppose that:

- the payments in instalments are fixed,
- the interest rate is equal to 10% .

```
payments ([], C):-
                C ^= 0.
payments([V | ListOfPayments], C):-
                RemainToPay ^= (1 + 10/100)*C - V,
                payments(ListOfPayments, RemainToPay).
```

The ^= denotes an equality constraint over rational terms. The first argument of the predicate *payments* is the list of yearly reimbursements which are necessary to reimburse C, the capital of the load.

The first clause of the program expresses that it is not necessary to reimburse a capital which is null. The second clause says that the list of the n+1 successive payments is the payment V followed by the list of n successive payments which will allow to reimburse the capital C increased by 10% of the interest but decreased by the payment V already done.

As an example we want to compute the values of the three payments V1, V2 and V3 such that:

- V2 is twice the amount of V1,
- V3 is three times the amount of V1,
- the capital to reimburse is 1000.

Example 1:

```
?- payments([V1, 2*V1, 3*V1], 1000).

V1 = 207.64430
```

The program above can be used in different ways. As the constraints express a relation which is valid between the variables, we can for example reverse the query. If we want to compute how much we can borrow assuming that we can pay back the loan with five payments of 100, the query will be:

Example 2:

```
?- payments([100, 100, 100, 100, 100], C).

C = 379.07867.
```

This flexibility is a very important aspect of CLP programming. The same program can be used in different ways without re-programming. The constraint solver handles the details of how the solution is obtained without user intervention.

6.1.2 Boolean

The Boolean computation domain differs from the rational domain mainly in the fact that deciding satisfiability of Boolean constraint is NP-hard [GJ79]. This means in practice that the basic constraint solving method can be very expensive.

6.1.2.1 Domain

The Boolean computation domain consists of terms build from the truth values 0 and 1, from Boolean variables and operators for the functions *and*, *or*, *xor*, *not* etc. We only need to

handle equality constraints, as other constraints (disequality, implication) can be easily expressed as equalities.

6.1.2.2 Methods

There are a number of different methods to solve Boolean constraints in a complete solver.
Two variants of *Boolean unification* have been introduced. These methods solve sets of equality constraints over Boolean terms. Boole's method [BS87] is based on variable elimination, Loewenheim's method [NM89] is based on the generalisation of a particular solution to the constraint set. Another approach is a specialised version of the *Groebner bases* method used in [ASS88]. Yet another approach uses *saturation* to check consistency [Col90].
An important aspect of Boolean constraint solving is the need of a compact data representation. Ordered binary decision diagrams (OBDD) offer such a compact representation for many common Boolean functions [Sim92].

6.1.2.3 Example

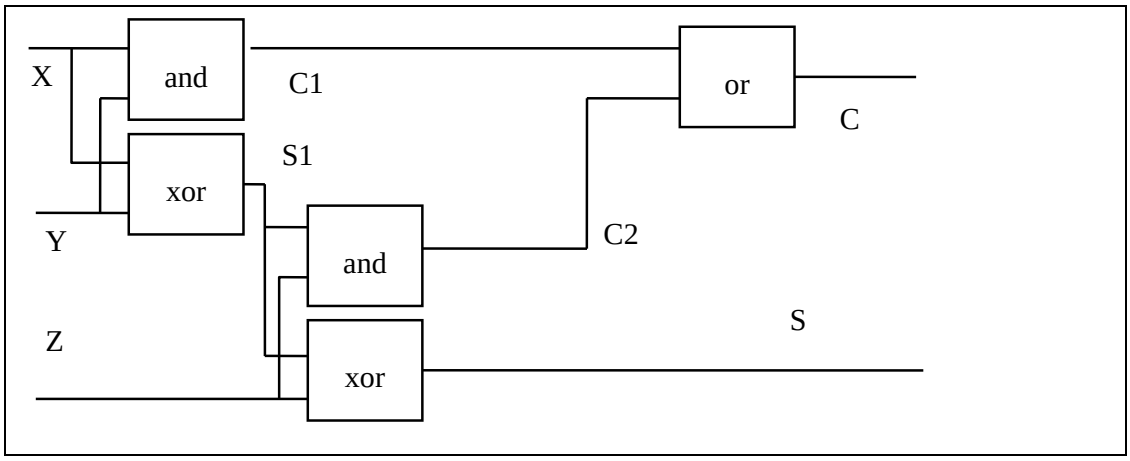


Figure 3: Full adder circuit example

The classical example of Boolean constraint solving is a full adder circuit (see [BS87]). The circuit computes the addition of three binary inputs *X*, *Y*, and *Z* as a sum *S* and a carry *C*.
A representation of the circuit as a constraint problem is shown below:

```
fa(X, Y, Z, S, C):-
                S1 &= X xor Y,
                C1 &= X and Y,
                S &= S1 xor Z,
                C2 &= S1 and Z,
                C &= C1 or C2.
```

The &= symbol denotes an equality constraint over Boolean terms. We obtain the following results from the query

```
?-fa(a,b,c,S,C).
        S1 = a xor b
        C1 = a and b
        S = a xor b xor c
        C2 = (a and c) xor (b and c)
        C = (a and b) xor (a and c) xor (b and c)
```

The results are expressed in terms of the functions *and* and *xor* and can be symbolically compared with a specification of the expected behaviour. Any differences show combinations of inputs where the expected and the actual behaviour differ.

6.1.3 Pseudo Boolean

The methods of the Boolean constraint solvers can be generalised for the Pseudo Boolean case [Boc93]. In this domain, we allow arithmetic constraints over 0/1 variables with integer coefficients. Symbolic solutions obtained by a complete solver are very difficult to use in this case, incomplete methods from Operations Research seem to be more promising from a practical point of view.

6.2 Incomplete Solvers

In this section we discuss some incomplete solvers for different computation domains. The most important one is the finite domain solver, which we will present in more detail in the next section. Other incomplete solvers can be defined for the Boolean domain or for Pseudo Boolean constraints. The bound propagation of finite domain constraints can be extended to intervals over real numbers.

6.2.1 Finite Domains

6.2.1.1 Domain

Similar to consistency methods, the basic elements of finite domain solvers are variables and constraints. The variables range over a finite set of possible values which is known a priori. For practical purposes, the finite set is defined as a finite subset of the natural numbers. The constraints express relations over these variables. Many different types of constraints can be defined, we will discuss below arithmetic, symbolic and global constraints.
As the constraint solver is not complete, a statement of the domains and the constraints will normally not be enough to solve a problem. The constraint solver must be combined with an *enumeration procedure*, which assigns the variables to admissible values. The choice of this enumeration procedure can have a huge effect on the performance of a particular program.

6.2.1.2 Classification

Based on the initial work on CHIP [DSV87] [DVS88a] [VH89], a number of different finite domain solvers have been designed. They differ in aspects of the

- constraint granularity
- richness of constraint sets
- propagation results
- user definable constraints/control

The *constraint granularity* defines which level of primitives are provided. Some systems are based on very low level constraint primitives [VSD92] [VSD92a], from which all other constraints can be defined. This provides a uniform framework and allows to refine constraint definitions based on those primitives. On the other hand, it makes application programming and debugging more complex and optimisation less likely. Other systems, like CHIP, provide constraints which correspond directly to constraints in the application domain. The constraint set is richer, and more easily understood without a deep understanding of the operational behaviour. As a disadvantage we can see the need to decide which alternative constraints to use and basically a restriction to the pre-defined constraint set.
Different constraint systems also differ in the *amount of propagation* that is provided by the constraints. Especially for more complex constraints, different degrees of propagation can be implemented. We will discuss this distinction below in the section on semantic or syntactic methods. As problems become more and more complex, the need for powerful constraint propagation increases, even if this may mean a higher cost for simple problems. Ideally, the application programmer can influence the amount of propagation.
Most systems provide some way of defining constraints by programming. More important is the possibility to control the enumeration scheme by easily *defining heuristics* and strategies inside the system.

6.2.1.3 Methods

Two basic methods for finite domain constraints should be explained at this point. The first one is called *forward checking* and is applied for disequality constraints, the second one is called *lookahead* and is applied to inequality constraints [HE80] [DSV87] [VH89].
Consider a disequality constraint between two variables which range over possible values from one to five. As long as no additional information is available, we can not solve the constraint. If one of the variables is bound to a value, say three, we can deduce more information. Since the other variable must have a different value, we can solve the constraint by removing the value three from the domain of the remaining variable. The general principle of *forward checking* is to wait until the constraint contains only one free variable. At this point, all incompatible values for this variable can be removed from the domain by the constraint procedure. The remaining values in the domain of the variable satisfy the constraint, which is therefore solved and does not have to be reconsidered.
For inequality constraints, the stronger *lookahead* procedure can be applied even if no variables have been assigned yet. If we consider an inequality $X < Y$ where X has a domain from 5 to 15 and Y a domain from 0 to 10, we note that some values for X and Y are not admissible. X can never take values from 10 to 15, since X is smaller than Y and the largest value for Y is 10. In the same way, Y can not take values from 0 to 5, since it is larger than X and the smallest value of X is 5. We can therefore immediately remove those values from the domains of X and Y.
But this does not solve the constraint. We might still choose values for X and Y so that the constraint is violated. We have to check the constraint again whenever the domains of X or Y change until one the variables becomes instantiated. For inequalities, we do not need to look at values inside the domain, we can calculate with the minima and maxima of the domains of the variables. This *bound propagation* or *partial lookahead* is a useful specialisation of the more general lookahead procedure.

6.2.2 Boolean

The incomplete solvers for Boolean constraints are based on the idea of local propagation. This idea was already exploited in constraint systems like CONSTRAINTS [SS80] or ThingLab [Bor81]. *Local propagation* (also called *value propagation*) amounts to deducing values for some variables in a constraint given the values of other variables. For instance, an and-constraint over the Boolean domain may be defined by rules similar to

- If one input is 0, then the output is 0
- If the output is 1, then all inputs have value 1

To implement a program achieving this form of propagation, it is necessary to introduce a form of *data driven computation* where goals are suspended when not enough information is available and where they are reactivated when new information becomes available.
This may be done with basic constraint primitives or the constraints can be built-in. It is also possible to simulate Boolean constraints with a number of other constraint solvers, this is shown in more detail in [SD93].
In CHIP, the basic Boolean constraints are built-in in form of demons. The demon for the and operation, for example performs the following rules

```
and(X,Y,Z):-
                X = 0 -> Z = 0,
                X = 1 -> Z= Y,
                Y = 0 -> Z = 0,
                Y = 1 -> Z = X,
                Z = 1 -> (X = 1, Y = 1),
                X = Y -> X = Z.
```

Each line of the clause defines a rule with the condition on the left side and an implication on the right side. The input argument of the and constraint are *X* and *Y*, the output argument is *Z*. If the constraint is called with the query

```
?- and(0, Y, Z)
```

the demon will wake and apply the first rule. As a result the output *Z* will be set to 0 and the constraint is solved. The value of *Y* is not required.
On the other hand a query of the form

```
?- and(X, Y, 0)
```

will *flounder*. The constraint will be delayed, but not woken, as none of the rules applies and no additional values are set.

6.2.3 Pseudo Boolean

An interesting approach is taken in [Bar94] for an incomplete solver over pseudo Boolean constraints. Cutting plane methods are well known in Operations Research for solving integer programming problems. Special cuts exists for the 0/1 case. These methods can be modified for an incomplete constraint solver.

6.2.4 Intervals

One of the basic ideas in the finite domain solver is the propagation of minimum and maximum domain bounds in arithmetic equality and inequality constraints. This method can be generalised to bound propagation over real numbers [OV90]. Variables range over intervals represented by two floating point numbers. In difference to the case of finite domains, all operations must apply safe rounding modes in order to preserve correctness of the results. Interval methods allow to easily combine different domains like integer subsets, 0/1 variables or continuous domains. Unfortunately, bound propagation alone is a relatively weak propagation method. Intervals also do not handle disequality constraints, which severely restricts their usefulness for finite domain problems.

6.2.5 Others

There are other constraint solving methods for domains like sets, multi-sets, sequences or lists. Research on these topics is still ongoing [Ger94][Col90][LL91] and is not discussed here.

7. Finite domain solver

We will now return to the finite domain solver and discuss it in more detail.

7.1 Domain concept

7.1.1 Domain Definition

Domain variables are defined in CHIP with the *::* primitive. Several types of domains are provided. Some examples are

```
X :: 1..10
Y :: 100:200
[A, B, C]  :: 0..100
```

The first statement defines a single domain variable *X* with domain from 1 to 10. The second example defines a domain variable *Y* as an interval from 100 to 200. Interval variables can not be used in constraints which remove values from the middle of a domain, like disequality constraints. But as they do not have to represent the domain inside their data structure, they use less memory than explicit domains. The third example defines a list of three variables *A*, *B* and *C* which all have the same initial domain from 0 to 100.

7.1.2 Indomain

Another basic primitive in CHIP is the *indomain* predicate. This non-deterministic predicate can be used to enumerate all values in the domain of a variable. It has one argument, a domain variable, and binds the variable to the smallest values in the domain. On backtracking, it chooses the next value, until all values have been tested. The indomain

primitive can be combined with the built-in backtracking search of Prolog to define enumeration schemes for sets of variables. The most simple form is

```
labeling([]).
labeling([H|T]):-
        indomain(H),
        labeling(T).
```

This predicate will search for a ground assignment of a list of domain variables, and will, on backtracking, enumerate all such assignments.

7.2 Constraints

We now look at the different constraints provided for finite domains. We distinguish three classes of constraints, arithmetic and symbolic constraints based on syntactic methods, and global constraints based on semantic methods.

7.2.1 Syntactic Methods

We have already discussed in section 6.2.1.3 two basic principles which are used to solve constraints with syntactic methods, forward checking and partial lookahead.

7.2.1.1 Numerical Constraints

Numerical constraints express equality and inequalities between domain variables. The propagation method of these constraints is partial lookahead. Special versions of the constraint can be defined for the two variable case. The more general form of the constraint handles linear terms over domain variables.
It is possible to extend the constraints to non-linear constraints over finite domains. Experiments have shown that the propagation with partial lookahead is rather slow and that the constraints do not occur too often in practical problems.

7.2.1.1.1 Linear Terms

If we extend the arithmetic constraints to linear terms, then the most general form of the equality constraint becomes

$$a_1X_1 + a_2X_2 + \ldots + a_nX_n + c_1 = b_1Y_1 + b_2Y_2 + \ldots + b_mY_m + d_1$$

where *X1,..Xn, Y1,..Ym* are domain variables and *ai, bi, ci, di* are natural numbers. The reasoning on this constraint is again performed by partial lookahead. We obtain bounds for the left and right hand side of the equation

$$a_1X_1 + a_2X_2 + \ldots + a_nX_n + c_1 \in \left[\min_l, \max_l\right]$$

and

$$b_1Y_1 + b_2Y_2 + \ldots + b_mY_m + d_1 \in \left[\min_r, \max_r\right]$$

If the constraint is satisfied the bounds can be strengthened to

$\min = \max(\min_l, \min_r)$

$\max = \min(\max_l, \max_r)$

From this we can deduce new bounds for the individual variables.
We see this on a small example. The equality constraint over finite domains is expressed with the symbol #=. From the constraint

```
[X, Y, Z]:: 1..10,
2*X + 3*Y + 3  #= Z
```

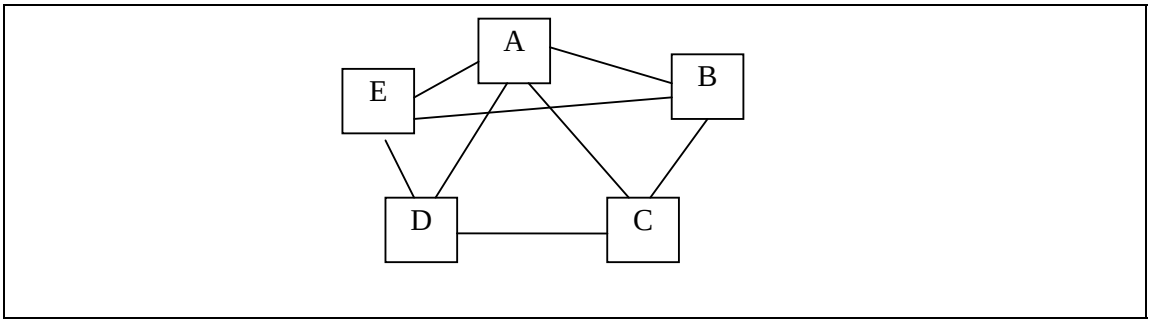
we deduce the bounds 8..53 for the left hand side and 1..10 for the right hand side. If the constraint is satisfied, then the value must be in the range 8..10. This restricts the domain on *Z* directly. Since *X* has a minimum value of 1, the term *3*Y* must be in the range 3..5. The only compatible value for *Y* is 1, and this leads to the possible values for *X* of 1 and 2. The constraint reasoning stops at this point. Note that the value 9 for *Z* can never be achieved, but that the bound propagation does not remove inconsistent values from the middle of the domain.

7.2.1.2 Symbolic Constraints

Symbolic constraints are useful to express non-arithmetic conditions between domain variables. The most basic form are disequality and the binary element constraints. Other symbolic constraints work on sets of domain variables. In difference to the global constraints defined below they use syntactic constraint propagation mechanisms like forward checking or partial lookahead to reduce the search space.

7.2.1.2.1 disequality

We will now consider a small example to show how the disequality constraints are used. Consider a simple graph colouring problem. The graph in Figure 4 should be coloured with 3 colours, i.e. we have to assign colours to the nodes of the graph in such a way that nodes which are linked never have the same colour. We describe each node as a domain variable



with domain 1 to 3, and each link in the graph as a disequality constraint

Figure 4: Graph colouring example

```
graph(A,B,C,D,E):-
        [A,B,C,D,E]  :: 1..3,
        A ≠ B, A  ≠ C, A ≠ D, A ≠ E, B ≠ C, B ≠ E, C ≠ D, D ≠ E,
        labeling([A,B,C,D,E]).
```

When the program is executed, the disequality constraints are set up. They delay until the labeling procedure is called and assigns the first variable *A* to the value 1. At this moment, all constraints using the variable *A* are woken and the value 1 is removed from the domain of the variables *B, C, D* and *E*. In the next labeling step, the variable *B* is instantiated to the value 2. This wakes the unsolved constraints using the variable *B*. These constraints remove the value 2 from the variables *C* and *E*. Since the domain of these variables now only contains one value, they are automatically assigned to the value 3. This wakes the two remaining constraints which delete the value 3 from the domain of *D*, which is automatically assigned to the value 2. At this point all variables are assigned and all constraints are satisfied. The labeling routine continues on ground values and the system returns the answer 1, 2, 3, 2, 3 as a first solution. The enumeration scheme and the constraint solver are interacting without explicit user control. The program still shows a very simple structure even if the actual calculation of an answer is completely data-driven and quite complex.

7.2.1.2.2 element

The element constraint expresses a functional dependency between two variables. It has the form

```
element(X, L, Y)
```

with domain variables *X* and *Y* and a list of integers *L* and states that *Y* is equal to the *Xth* element of *L*. The constraint reasoning on the constraint works in both directions. If the domain of *X* is restricted, then only those values in the domain of *Y* are retained which correspond to the possible entries in the list *L*. If alternatively the domain of *Y* is constrained, the domain of *X* is restricted to the corresponding indices.
The element constraint is particularly useful to express the cost of making choices. If in an assignment problem a task can be assigned to five possible machines named 1 to 5, with costs of 10, 20 ,5, 10 and 5 respectively, then we can link the decision variable *X* and the cost variable *Y* via an element constraint

```
element(X, [10, 20, 5, 10, 5], Y)
```

The variable *Y* now stands for the cost of the assignment of *X*. If for example the domain of the cost variable is restricted by *Y < 15*, then value 2 will be removed from the domain of *X*. In the same way, a constraint *X < 3*, will remove the value 5 from the domain of *Y*.
For many problems, using the element constraint allows to derive a more compact and natural problem representation and helps to avoid large sets of 0/1 decision variables.

7.2.1.2.3 alldifferent

The alldifferent constraint states that all values in a list of domain variables must be different. It is equivalent to defining pair-wise disequality constraints between the variables. The constraint propagation uses forward checking as a syntactic method. We can use other, global constraints (see below) to express the same declarative condition and obtain better propagation.
Other constraints of a similar form, like *atmost*, *atleast* and *circuit* have been defined. They all use syntactic methods, which for most problems are inferior to propagation obtained by global constraints.

7.2.2 Semantic Methods

The constraints in the previous sections were all based on syntactic, domain independent propagation methods. We now present some finite domain constraints based on semantic methods. These constraints use domain specific knowledge to derive better propagation results. Constraint of this type are called *global* constraints [AB93][BC94] and combine several important properties:

- They model a complex condition on sets of variables.
- The condition can be used in multiple contexts.
- The constraint reasoning detects inconsistency in many situations and reduces the search space significantly.
- They can be applied to large problem instances.

In the following we will discuss four such constraints found in CHIP:

- The **cumulative** constraint is used to express cumulative resource limits over a period.
- The **diffn** constraint expresses non-overlapping constraints on n-dimensional rectangles.
- The **cycle** constraint finds cycles in directed graphs.
- The **among** constraints enforces constraints on sequences of numbers.

The constraints can be used and combined for many different problems and significantly extend the range of problems which can be handled with CLP.

7.2.2.1 cumulative

A typical use of the cumulative constraint is found in resource restricted scheduling. We have to schedule a number of tasks of different duration where the tasks require certain amounts of manpower during their operation. The overall amount of manpower available during the scheduling period is fixed and the total requirements at each time point by all tasks should not exceed the available limit. Other constraints, like precedence, may be required in the problem and can be expressed with inequality constraints for example.

Formulation

This manpower constraint can be modelled with a simple form of the cumulative constraint. For each of the *n* tasks, we define domain variables *Si* for the start time, *Di* for the duration and *Ri* for the resource use of the task, and state the constraint

```
cumulative([S1, S2,..., Sn], [D1, D2,..., Dn], [R1, R2,..., Rn], Limit)
```

where the domain variable *Limit* ranges between 0 and the available resource limit. The cumulative constraint has the mathematical definition that

$$\forall i \in \left[\min_{1\le j\le n}(S_j), \max_{1\le j\le n}(S_j + D_j)\right]: \quad \sum_{k:S_k \le i < S_k + D_k} R_k \le Limit$$

This means that at each timepoint between the start of the first task and the end of the last task the cumulative resource use by all tasks running at this time point is smaller than the available resource limit. We can visualise the meaning of the constraint with a little diagram. Figure 5 shows the profile generated by a set of tasks. In x-direction we display time, in y-direction the resource use. A task is shown as a rectangle with duration Di, height Ri and the left border at time Si. The cumulative profile of all tasks must stay below the overall resource limit.

We have shown here only the most basic form of the cumulative constraint. It has a number of additional parameters to express conditions on the end dates of the tasks, on the surface of the rectangles, on the overall end date or on intermediate resource limits. These additional parameters make the constraint more flexible and allow to express more complex conditions.

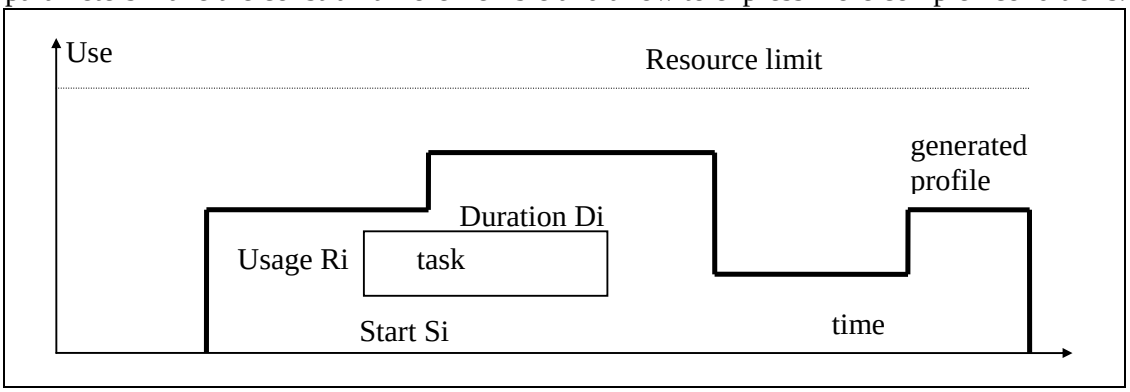


Figure 5: Cumulative resource profiles

Usage

An important consideration when defining the constraint was to make it as general as possible, so that the same constraint can be used for multiple purposes. In case of the cumulative constraint we can use it to express a large variety different conditions.

The handling of *disjunctive resources* can be treated as a special case of the cumulative constraint. A disjunctive constraint means that a resource can work on only one task at a time. Each task uses one resource unit, and the overall resource limit is one. These constraints are very common in scheduling problems, for example in the job-shop scheduling case and impose strong conditions on the sequence of tasks [AB93].

The cumulative constraint can also be used to express *bin packing* conditions. In bin packing problems, we have to place items of different sizes in bins of certain capacity. Often, the problem is to use a minimum number of bins. These constraints occur in many assignment and scheduling problems, where the bins correspond to resources with limited overall capacity [AB93].

Another use of the cumulative constraint is found in *producer/consumer* [SC94] constraints for consumable resources. These constraints occur in scheduling problems where certain operations produce materials, like intermediate products, which are consumed by other operations. The constraint states that at each time point more of these products must have been produced than consumed, but also that the amount of stock at any time point may be limited. A simple modelling of these constraints using the cumulative constraint is given in [SC94].

The cumulative constraint can also be used for a variety of placement and packing problems [AB93], where it imposes important necessary conditions on the existence of solutions.

Methods

The handling of the cumulative constraint combines a large number of different techniques. Around 20 different methods are used to check consistency or to define lower bounds of the resource use. To give an idea of how these methods work, we give two very simple examples.

One method compares the *available space* inside the resource constraint with the amount required by the different tasks. As an obvious bound, the following equation must be satisfied:

$$Limit \cdot (\max_j(S_j + D_j) - \min_j(S_j) \geq \sum_i D_i \cdot R_i$$

Different bounds on these terms can be calculated even when none of the variables have been assigned yet. Much stronger bounds can be obtained by a more careful reasoning on the available and required space inside the constraint.

A second method we want to explain is the use of *obligatory parts* for tasks. When the start time and the duration of a task is known, we can judge its influence on other tasks and restrict the possibilities for placing them. But even if the start and duration are not yet known, we can deduce some information, if the domains are sufficiently restricted. Figure 6 shows a typical situation. A task can start between its earliest and latest start date with a minimum and maximum duration. Regardless of the choice of values we know that the tasks will be running during the shaded period, and will therefore require resources during that period. As the domains get more and more restricted, this obligatory part will increase. But right from the beginning we can check whether obligatory parts overlap and constrain each other.

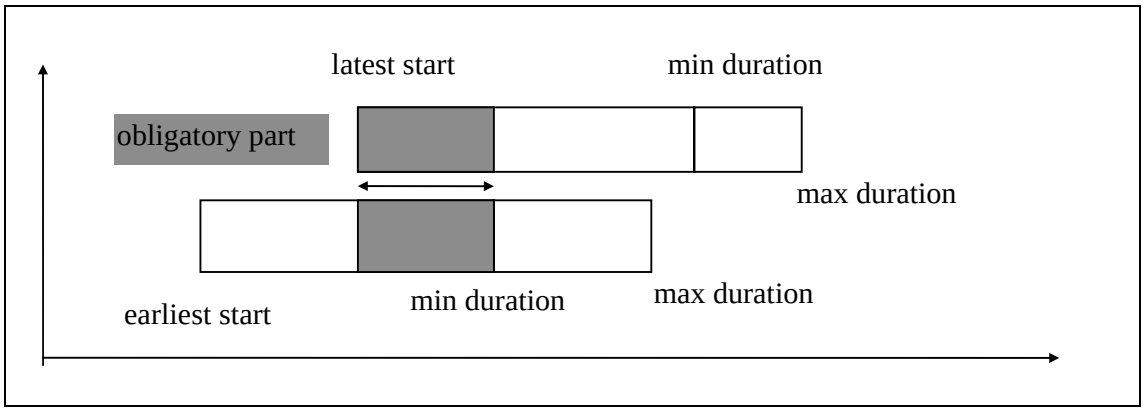


Figure 6: Obligatory part

All these different methods are used to deduce more about the constraint and to restrict the search space as early as possible. This means that even complex problems can be handled in an efficient, but still declarative manner.

7.2.2.2 diffn

The diffn constraint is another very useful construct to express that a set of n-dimensional rectangles do not overlap. For simplicity, we introduce the constraint here in two dimensions.

Formulation

The basic constraint is the following. We consider a set of *n* rectangles with lower left corner (*Xi, Yi),* with length *Li* and height *Hi*. These rectangles should not overlap. This means that for each pair of rectangles Ri and Rj, one of these four condition must hold:

- Ri is above Rj
- Ri is below Rj
- Ri is to the left of Rj
- Ri is to the right of Rj

If we want to translate this constraint into primitive conditions, we have to handle a disjunction of four arithmetic constraints. Experiments have shown that we obtain very poor propagation results, although we generate a number of constraints which is quadratic in the number of rectangles. Using the diffn constraint, we simply state

```
diffn([[X1, Y1, W1, H1], [X2, Y2, W2, H2],...,[Xn, Yn, Wn, Hn]])
```

The one argument of diffn is a list of lists, this allows to express the condition in higher dimensions as well.

Usage

Obviously, the diffn constraint can be applied for many types of *packing* and *placement* problems in two or more dimensions. In two dimensions this also corresponds to cutting problems of various forms. In three dimensions, typical problems are packing of palettes or containers with rectangular boxes. In four dimensions, we extend the three dimensional problem with a choice of the container. The first three co-ordinates describe the position of a box within the container, the fourth dimension describes in which of several containers we place each box.

The diffn constraint is also applied for *scheduling* and *assignment* problems, where it can easily model disjunctive machine assignment for tasks. The x-dimension of the constraint represents time, each value in the y-dimension stands for a possible machine assignment. Each task is represented as a rectangle, which is given by its start time, its resource assignment, its duration and a height 1 (see Figure 7). The non-overlapping condition states that no tasks can be scheduled at the same time on the same machine.

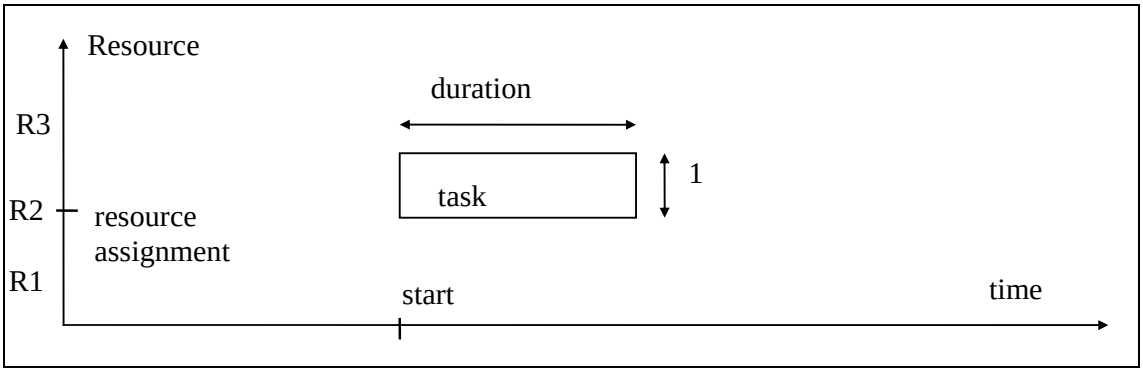


Figure 7: Machine assignment with diffn

The difference to the use of the cumulative constraint lies in the fact that we not only control that enough resources are available globally, but that we assign each task to a particular machine. This type of constraint also occurs in *time tabling problems* and personnel assignment situations.

Other constraints that can be expressed with the diffn constraints are *set-up times* in scheduling problems [Sim95] or *location continuity* for transport problems [Sim94]. Set-up times occur when a machine must be stopped between two operations for cleaning or to change tools. If the time between two tasks depends on the sequence of operations as well as the type of machine, then we can use the diffn constraint to model this complex condition. Location continuity constraints occur in transport problems if a resource may move between different locations.

We can also use the diffn constraint for routing problems like the VLSI channel routing [Sim92].

Methods
Similar to the cumulative constraint, the implementation of the diffn constraint uses a combination of different methods from OR and discrete mathematics. These methods are combined with the internal use of spatial data structures. One of the methods which is applied is a generalisation of the obligatory parts explained above to several dimensions.

The reasoning methods of the diffn and cumulative constraints are orthogonal, i.e. they rarely use the same reasoning and propagation methods. For many assignment problems it is worthwhile to combine a diffn constraint with cumulative constraints which describe the *cumulative relaxation* of a problem.

7.2.2.3 cycle
The cumulative constraint was inspired by methods from OR and scheduling, the diffn constraint uses geometrical methods and placement algorithms. Another global constraint, the *cycle constraint* [BC94], uses ideas from graph theory and combinatorics. The constraint is applicable to many transportation problems, where we have to plan sequences of visits to different places by one or several agents.
Mathematically speaking, this problem corresponds to finding one or several cycles in a directed graph. Figure 8 shows a simple example. We have to visit the places 1 to 6 by 2 agents. From each place, only some connections to other locations are possible. They are

represented by directed arcs. To express this problem with constraints, we use the cycle constraint.

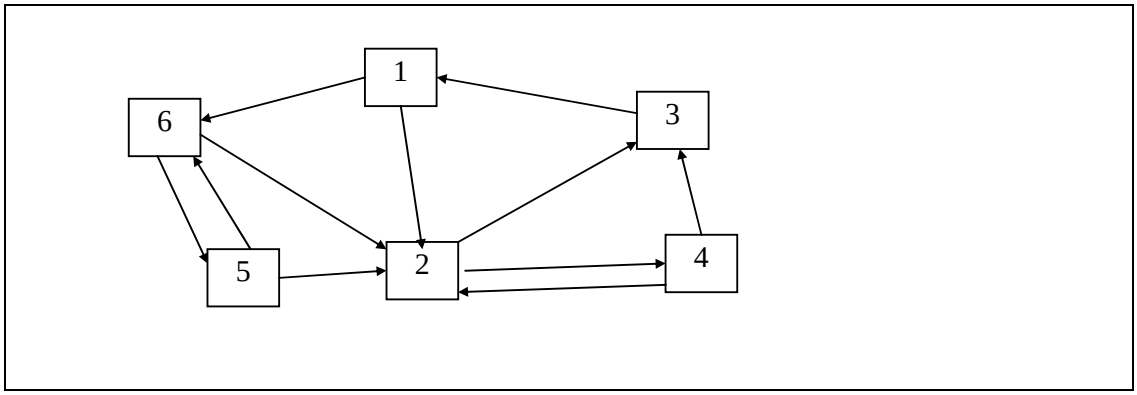


Figure 8: Directed graph example

We represent each node in the graph by a domain variable and number the nodes from 1 to *n*. The domain of a node variable is defined by the numbers of all nodes which are directly connected to the node. In Figure 8, the variable V1 has domain [2, 6] and the variable V2 has domain [3, 4] for example. The program to find two cycles in this graph is shown below:

```
graph(X1, X2, X3, X4, X5, X6):-
        X1 :: [2, 6], X2 :: [3, 4], X3 :: [1], X4 :: [2, 3], X5 :: [2, 6], X6 :: [2, 5],
        cycle(2,[X1, X2, X3, X4, X5, X6]),
        labeling([X1, X2, X3, X4, X5, X6]).
```

The program finds the solution [2, 4, 1, 3, 6, 5] which consists of two cycles, one containing the nodes 1, 2, 4 and 3, the other containing the nodes 5 and 6.

Usage

The cycle constraint can be used for many problems from the logistics area. Tour planning and travelling salesman problems (TSP) can be expressed quite simply. While such programs are not yet competitive with specialised algorithms for pure problems like TSP, they have the advantage that they can handle many additional constraints which can not be expressed inside specialised solutions for pure problems.

Another large application area for the cycle constraint consist in planning personnel rotations for example in the airline domain. The nodes of the graph are tasks to be assigned to personnel, connections exist if two tasks can be done consecutively by the same person. Constraints on location, working time, precedence etc can all be included in the model [BKC94].

Specific constraints in many scheduling problems can be expressed with the cycle constraint. An example is the handling of sequence dependent compatibility or set-up time conditions [Sim95].

The problem of finding cycles in graphs is closely related to finding permutations with certain properties. The cycle constraint can be used for such problems as well [BC94].

Methods

The methods which are integrated in the cycle constraint mainly come from graph theoretical algorithms. One typical method calculates the number of connected components in the graph. This is a lower bound for the number of cycles that we can find. If there is no connection from one node to another node (via a number of intermediate nodes), then obviously these nodes must belong to different cycles. A number of other methods are used which for example remove arcs from the graph when they can not be used by any cycle.

7.2.2.4 among

The among constraint expresses conditions on *sequences* and sub-sequences of domain variables. We can express the condition that values in the set *V1, V2, ..., Vm* should occur atleast *K* times and atmost *L* times in any sub-sequence of length *P* of the sequence *X1, X2, ..., Xn*.
The condition is expressed with the constraint

```
among(K, L, P, [X1, X2, ..., Xn], [V1, V2,..., Vm])
```

With the constraint

```
among(3, 3, 5, [X1, X2, X3,..., X10], [9])
```

we state that each of the following sets contains the value 9 exactly three times.

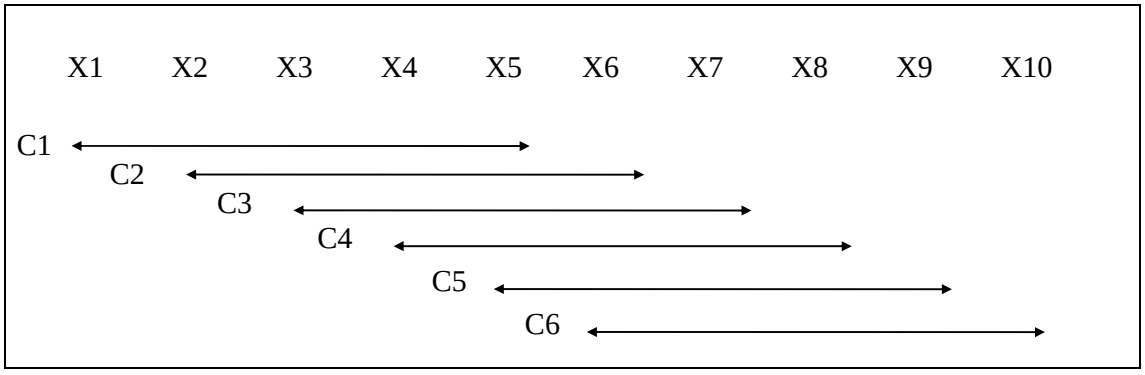


Figure 9: Overlapping periods for among

Usage

The constraint is very useful for *time tabling problems* or general planning problems. In this context, the value Xi correspond to the type of operation performed in time interval i. The different values of these variables mean the different activities which can be performed, like work or rest periods. With the among constraint we can express for example constraints that every interval of a certain length should contain atleast K1 rest activities or atmost L2 heavy duty work activities. The constraint is enforced for every sub period of length P.
Another typical use of the among constraint is to *balance the resource use* over a longer time span by enforcing that for short periods only a maximum number of uses occur. In this way balancing heuristics can be implemented.

Methods
The constraint reasoning uses the interaction between the constraints on each sub-sequence to deduce additional, stronger conditions which must be satisfied by the ensemble of constraints. If, in the example of Figure 9, the variable X2 receives the value 9, then the propagation will enforce that X7 also has the value 9, since X3 to X6 must contain the value 9 twice (constraint C2) and X3 to X7 contain the value three times (constraint C3).

7.3 Control

So far we have talked about the different constraints which can be used to express conditions on finite domain problems. Given a set of variables and the constraints on them, the different propagation techniques of the finite domain solver will reduce the search space for the variables, but will normally not find a solution or detect inconsistency without search. We need an enumeration strategy to search through different possible assignments. As we can program this strategy ourselves, we have full control over the complexity of this search. A simple, built-in method can be used for small problems. Very complex strategies can emulate problem specific methods which are derived from human behaviour.
In this section we will look at two different aspects of search, heuristic search for one solution and optimisation under an optimisation criterion.

7.3.1 Heuristics

In many problems we are interested in finding a single solution which satisfies all constraints. Instead of blindly enumerating alternatives, we can, by making the right choices, quickly guide the program through the search tree to a solution. The search for such heuristics is common to both OR and artificial intelligence methods. We want to show how we can express different types of heuristics easily within the CLP framework.
To enumerate solutions of a finite domain problem, we have two main alternatives.

- The first method consists in assigning values to the variables until a ground solution is found and all constraints are satisfied. This assignment method creates a search tree which is traversed until a leave node, a solution is found.
- A second method consists in forcing choices among constraint alternatives and adding constraints to the system until the system allows only one solution and the propagation finds this solution. The cutting plane method from OR is such a technique. This approach depends on particular properties of the constraint system and is therefore very system specific An application to CLP is shown in [DSV90].

In the following we will only talk about the first alternative.

7.3.1.1 Variable Selection and Value Selection

If we choose a variable enumeration method, we have to make two types of decisions. Given the set of variables, we have to choose which variable from the set to assign next and which value to try for this variable. The *variable selection* rule decides which variable is taken next, and the *value selection* rule state which value is tested next. Both of them may be deterministic or non-deterministic. This leads to two variants of the variable assignment methods.

7.3.1.2 Value Choice

In this case we choose the variable from the set deterministically and branch on the possible values of the variable. This *value choice* is a standard enumeration method of artificial

intelligence. We can apply different heuristics to select the right variable and we can try the possible values of this variable in different order to quickly find a solution.

- A well known strategy of value choice is the *first fail* [HE80] [VH89] method. We select the variable which has the smallest number of possible alternatives left in its domain. This limits the branching factor of the search tree, as we only have to consider a few alternatives for the variable.
- Another method is the *most constraining* selection rule. We pick the variable which occurs in the largest number of constraints and therefore has a strong influence on many other variables.
- Yet another selection rules is the *maximum regret* rule, where we choose the variable where the cost difference between making the cheapest choice and choosing another value is maximal.

Most of these heuristics are also well known from specialised algorithms. Note that all these heuristics lead to *complete* search methods, where all possible solutions are examined on backtracking.

7.3.1.3 Variable Choice

Another type of selection mechanism is known from many OR problems. Instead of choosing one variable and branching on its possible values, we choose non-deterministically from all variables and assign this variable deterministically. A typical example is found in certain scheduling problems, where we choose one task among all unassigned tasks, fix it to its earliest compatible start date and enforce that no other task may start before it. On backtracking, instead of trying different start-dates for this task, we choose another task to start before all other alternatives. This *variable choice* strategy often explores a much smaller search space, but its completeness is *not* guaranteed. This question must be studied on a case-by-case basis.

An important question for many problems is the elimination or reduction of symmetries in the solutions. This removal can be handled either by adding special constraints, like enforcing a partial order on the values of variables to remove the symmetries from the problem, or may be handled inside the assignment part by excluding certain branches of the search tree.

7.3.2 Branch and Bound

For some problems, we want to find an optimal solution which satisfies all constraints and minimises a cost function. Note that many optimal solutions with the same cost may exist. We are typically interested in just one of these solutions. If we have a routine to explore feasible solutions, called the generator, and a cost function which gives a cost value for each ground solution, we can use the *min_max* meta predicate to find optimal solutions. This meta predicate works as follows:

> The generator predicate is called to find a solution. If a solution is found, a new constraint is added that the improved cost should be better than the value which has been found. The generator is recalled to find a better solution. If no (more) solution is found, the best solution so far is returned or the systems fails if none has been found at all.

This *re-starting* optimisation is a very simple, but quite powerful optimisation method. Other methods have been investigated or implemented [VH89][DVS88a].

Parallel search for optimal or feasible solutions is possible. This has been proposed for either multi-processor systems with shared memory or for networked workstations. If the search tree is cut at a node close to the root, then the different sub problems will be quite large and linear speedups are easily obtained. Due to the exponential nature of most finite domain problems, this linear speed-up has little influence on the type or size of problems which can be efficiently handled.

8. Constraint Systems

In this section we discuss the history and features of different CLP systems. A (simplified) family tree of CLP systems and developments is shown in Figure 10.

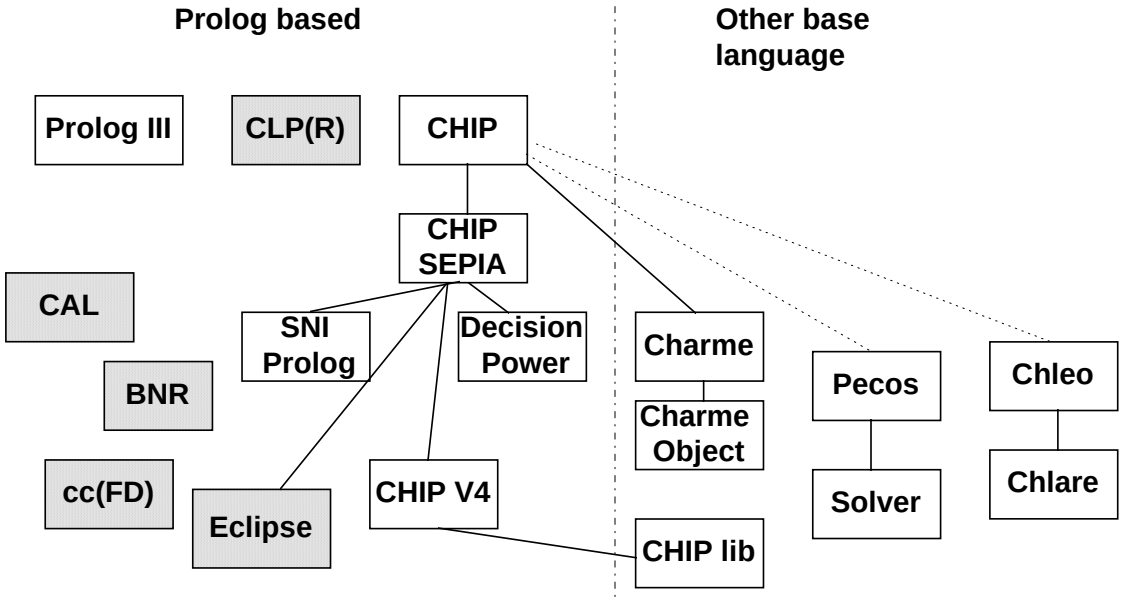


Figure 10: CLP tool history

Commercial systems are outlines in black, research systems are filled in grey. We will now discuss some of these systems in more detail.

8.1 Prolog III

Prolog III [Col90] was one of the first CLP systems. It is now a commercial product of Prologia in Marseilles. It contains three different constraint solvers

- a complete solver for linear arithmetic over rationals
- a complete solver for Boolean terms based on saturation
- an incomplete solver for equality constraint over lists

A successor, called Prolog IV, is currently under development.

8.2 CLP(R)

CLP(R) [JMS90] was developed at IBM and Monash University as a demonstrator for the CLP(X) [JL87] scheme. It contains a

- complete constraint solver for linear terms over real numbers (represented by floating point numbers)

An updated, compiled version is also available.

8.3 CHIP

CHIP [DSV88a] was another of the first CLP systems. It was first developed at ECRC in Munich, and is now being marketed and developed by COSYTEC. It contains

- an incomplete solver for finite domains,
- a complete solver for linear terms over rationals, and
- both complete and incomplete Boolean solvers.

The current version CHIP V4.1 includes a number of global constraints [AB93] [BC94] and offers others features for a complete application environment (objects, graphics, interfaces). The early versions of CHIP were the basis for a number of other developments. ECRC shareholder companies developed different products, CHARME (see below) by BULL, DecisionPower by ICL and SNI-Prolog by Siemens. ECRC has also continued research on constraints. A new system, called Eclipse is currently available as a research platform.

8.4 CAL

The Japanese research centre ICOT has produced CAL [ASS88], a constraint system written in Prolog with constraint solvers for

- Boolean constraints (complete solver based on Groebner bases)
- Linear terms (Simplex based, complete)
- Non-linear constraints (Groebner bases)

A parallel version GDCC is also available.

8.5 BNR

The BNR [OV90] system is available from Bell Northern Research. It contains a

- incomplete solver for intervals

which is based on bound propagation.

8.6 Non logic based

Motivated by the success of CLP techniques, in particular finite domain constraints, a number of non-logic based constraint systems have appeared on the market. These offer mainly finite domain constraints as C or C++ libraries. Since the host language does not support backtracking, they either restrict the possibilities of developing user-written labeling routines or provide complex control mechanisms to simulate backtracking in a user program. More details on these systems can be found in [Cra94].

8.6.1 CHARME

The first such system was CHARME by BULL. It is based on the finite domain solver of an early CHIP version, but uses a proprietary, C-like language to express constraints. A successor, CHARME object, has been discontinued.

8.6.2 Solver

The French company Ilog is offering Solver, a C++ library of finite domain constraints.

8.6.3 Chleo/Chlare

Another finite domain C library is available form the French company Axia.

8.6.4 CHIP library

The finite domain and the rational solvers of CHIP are also available as C/C++ libraries from COSYTEC.

9. Application examples

In this section, we show different industrial application of CLP. A large number of studies and research prototypes using CLP have been reported. An overview of these activities can be found in [JM94]. Particular examples are shown for example in [CL94] [CG93] [LW93] [Rue93] [Wal94].

One of the first large scale applications of CLP was the **HIT** scheduling system developed for a container terminal at Hong Kong harbour. The system, developed by ICL, used CHIP, later DecisionPower to schedule and assign berths for ships arriving at the harbour. The system used finite domain constraints to express various conditions on the schedule.
Besides the HIT application, DecisionPower [Eva92] has been used for

- Fleet management (Cathay Pacific)
- Training staff scheduling and allocation (National Westminster Bank)

An in-house application developed at Siemens uses the Boolean constraint solver inside SNI-Prolog for digital circuit verification. The program **CVE** [FST91] provides interfaces to different design tools and checks application specific integrated circuits (ASIC) against formal specifications.

The constraint system CHARME has been applied to a number of problems, among them

- Planning production and packaging in an oil refinery (Elf)
- Determination of product mix & manufacturing of animal nutrition products (Rhone-Poulenc)
- Scheduling of spare parts arrival in a car assembly factory (MCA/Renault)
- Resource allocation in a crisis management system (Albertville 92)

The constraint library of ILOG Solver has been used for

- Planning “personnel evolution policy” (French Army)
- Fleet management - Locomotives (SNCF)
- Maintenance scheduling - Locomotives (SNCF)
- Manpower planning for shift operation (Banque Bruxelles Lambert)

9.1 Applications using CHIP

In this section we concentrate on large scale, industrial systems developed with CHIP. These examples show that constraint logic programming based on Prolog can be used efficiently to develop end-user applications. The constraint solver is a small, but crucial part of these applications. Other key aspects are graphical user interfaces and interfaces to data bases or other programs. Some background on how such applications can be developed with logic programming is given in [KS95].

ATLAS, [SC94] a tool for detailed production scheduling in a chemical factory, is based on a constraint model using numerous constraints on machines and consumable resources. The constraint solver is embedded in a user-friendly graphical environment and uses a relational database to exchange information with other information system tools. The program was co-developed by the Belgian company Beyers and Partners and COSYTEC.

FORWARD is a simulation and scheduling tool for oil refineries. In this application, CHIP is linked to a simulation tool written in FORTRAN which calculates results from a schedule obtained with the CHIP solver. The basic constraints express a non-linear optimisation problem over a continuous domain. The program was developed by the engineering company TECHNIP and COSYTEC and is currently in used at two refineries.

TACT combines planning, scheduling and assignment aspects for the operational fleet management in the food industry. The system uses several interacting solvers to create a schedule for drivers, lorries and other resources in a transportation problem. The system is used both to create working schedules and to handle ‘what-if’ decision support scenarios. The application was developed by COSYTEC.

Dassault Aviation has developed several applications with CHIP. **PLANE** [BCP92] is a medium-long term scheduling system for aircraft assembly line scheduling. **MADE** [CDF94] is controlling a complex work-cell in the factory, and **COCA** schedules micro-tasks in the production process.

CHIP is used inside the **TFS** program from TFP as part of an optimisation module for a process scheduling package. The constraint solver is used as a back-end invisible to the user

In a similar way, **Saveplan** from Sligos uses CHIP as a scheduling engine for a shop-floor scheduling system. Saveplan is a FORTRAN scheduling package, which has been re-engineered in Delphia Prolog.

The **LOCARIM** application is capable of designing a computer/phone network in large buildings. It is used by France Telecom to design the cabling and estimate cabling costs for new buildings. The constraint model corresponds to an extended warehouse location problem. The application was developed by Telesystemes together with COSYTEC.

EVA is used by EDF to plan and schedule the transport of nuclear waste between reactors and the reprocessing plant in Le Hague. This problem is highly constrained by the limited availability of transport containers and the required availability of the reactors. The program was co-developed by EDF and the software firm GIST.

PILOT [BKC94] is a planning system for crew assignment in an airline. It schedules and reassigns pilots and cabin crews to flights and aircraft respecting physical constraints, government regulations and union agreements. The program is intended for day-to-day management of operations with rapidly changing constraints.

SEVE is a portfolio management system for CDC, a major bank in Paris, developed in-house. It uses non-linear constraints over rationals and finite domain constraints to choose among different investment options. The system can be used for "what-if" and "goal-seeking" scenarios, based on different assumptions on the development of the economy.

Other application development with CHIP has been reported in [DVS88] [VC88] [DS91] [SD93] [CF94] [Ber90] [VH89] [DSV92] [DSV88] [Sim92].

10. To learn more

This tutorial only offers a first introduction to CLP. To learn more about this field, the following reviews and articles are recommended:

[JM94] gives a comprehensive overview of the field, discussing methods and theory as well as giving an application overview.
[FHK92] gives an informal overview of CLP, with emphasis on user definable constraints.
The book [VH89] is useful as a source of many examples and a introduction to finite domain constraints.
The commercial report [Cra94] gives an overview of the non-academic tools available in the field.
Books like [BC93][Bor94] give an overview of current research topics.
The book [Sar93] presents the field of concurrent constraint languages in a theoretical framework.
For a more general overview of constraint programming techniques see [FM94].
In addition, the newsgroup *comp.constraints* offers a list of FAQ (frequently asked questions) which contains an up-to-date list of all available academic CLP systems and which gives pointers to current issues in constraint logic programming.

11. References

[AB93] A. Aggoun, N. Beldiceanu
Extending CHIP in Order to Solve Complex Scheduling Problems
Journal of Mathematical and Computer Modelling, Vol. 17, No. 7, pages 57-73
Pergamon Press, 1993

[ASS88] A. Aiba, K. Sakai, Y. Sato, D. Hawley, R. Hasegawa
Constraint Logic Programming Language CAL
Proc Int Conf on Fifth Generation Computer Systems 1988, 263-276, 1988

[Bar94] P. Barth
Simplifying Clausal Satisfiability Problems
in J. Jouannaud (Ed.) Constraints in Computational Logics,
LNCS 845, Springer Verlag, 1994

[BKC94] G. Baues, P. Kay, P. Charlier
Constraint Based Resource Allocation for Airline Crew Management
ATTIS 94, Paris, April 1994

[BC94] N. Beldiceanu, E. Contejean
Introducing Global Constraints in CHIP
Journal of Mathematical and Computer Modelling, Vol 20, No 12, pp 97-123, 1994

[BCP92] J. Bellone, A. Chamard, C. Pradelles
PLANE -An Evolutive Planning System for Aircraft Production.
First International Conference on the Practical Application of Prolog.
1-3 April 1992, London.

[BC93] F. Benhamou, A. Colmerauer (Editors)
Constraint Logic Programming: Selected Research
MIT Press, 1993

[BMV94] F. Benhamou, D. McAllester, P. Van Hentenryck
CLP(Intervals) Revisited
ILPS, Ithaca, NY., November 1994

[Ber90] F. Berthier
Solving Financial Decision Problems with CHIP
Proc 2nd Conf Economics and AI, Paris 223-238, June 1990

[Boc93] A. Bockmayr
Logic Programming with Pseudo-Boolean Constraints
in [BC93]

[Bor81] A. Borning
The Programming Language Aspects of ThingLab, A Constraint Oriented Simulation Laboratory
ACM Transactions of Programming Languages and Systems, 3(4), 252-387, October 1981

[Bor94] A. Borning (Ed)
Principles and Practice of Constraint Programming
PPCP 94, Rosario, Orcas Island, WA, May 1994
Springer Verlag LNCS 874

[BS87] W. Büttner, H. Simonis
Embedding Boolean Expressions into Logic Programming
Journal of Symbolic Computation, 4:191-205, October 1987

[CL94] Y. Caseau, F. Laburthe
Improved CLP Scheduling with Task Intervals
Proc 11th ICLP 1994,
Italy, June 1994. MIT Press

[Col90] A. Colmerauer
An Introduction to Prolog III
Communications of the ACM 33(7), 52-68, July 1990

[CG93] M. Carlsson, M. Grindal
Automatic Frequency Assignment for Cellular Telephones Using Constraint Satisfaction Techniques
Proc 10th Int Conf Logic Programming, Budapest, Hungary, pp 647-665, 1993

[CDF94] A. Chamard, F. Deces, A. Fischler
A Workshop Scheduler System written in CHIP
2nd Conf Practical Applications of Prolog, London, April 1994

[CF94] C. Chiopris, M. Fabris
Optimal Management of a Large Computer Network with CHIP
2nd Conf Practical Applications of Prolog, London, April 1994

[CL94] C. Chiu, J. Lee
Towards Practical Interval Constraint Solving in Logic Progrmming
in ILPS 94, Ithaca, NY, November 1994

[Cra94] J-Y. Cras
A Review of Industrial Constraint Solving Tools
AI Perspectives Report
Oxford, UK, 1994

[Dav84] R. Davis
Diagnostics Reasoning Based on Structure and Behaviour
Artificial Intelligence, (24):374-410, 1984

[DVS88a] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier.
The Constraint Logic Programming Language CHIP.
In Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88), pages 693-702, Tokyo, 1988.

[DVS88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun and T. Graf.
Applications of CHIP to Industrial and Engineering Problems.
In First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Tullahoma, Tennessee, USA, June 1988.

[DSV90] M. Dincbas, H. Simonis and P. Van Hentenryck.
Solving Large Combinatorial Problems in Logic Programming.
Journal of Logic Programming - 8, pages 75-93, 1990.

[DS91] M. Dincbas, H. Simonis
APACHE - A Constraint Based, Automated Stand Allocation System
Proc. of Advanced Software Technology in Air Transport (ASTAIR'91)
Royal Aeronautical Society, London, UK, 23-24 October 1991, pages 267-282

[DSV92] M. Dincbas, H. Simonis, P. Van Hentenryck
Solving a Cutting-Stock Problem with the Constraint Logic Programming Language CHIP
Journal of Mathematical and Computer Modelling, Vol. 16, No. 1, pp. 95-105,
Pergamon Press, 1992

[DSV87] M. Dincbas, H. Simonis, P. Van Hentenryck.
Extending Equation Solving and Constraint Handling in Logic Programming.
In Colloquium on Resolution of Equations in Algebraic Structures (CREAS), Texas, May 1987.

[DSV88] M. Dincbas, H. Simonis, P. Van Hentenryck.
Solving the Car Sequencing Problem in Constraint Logic Programming.
In European Conference on Artificial Intelligence (ECAI-88),
Munich, W. Germany, August 1988.

[Eva92] O. Evans
Factory Scheduling Using Finite Domains
In Logic Programming in Action LNCS 636, 45-53, 1992

[Fik70] R. Fikes
REF-ARF: A System for Solving Problems stated as Procedures
Artificial Intelligence, 1, 27-120, 1970

[FST91] T. Filkorn, R. Schmid, E. Tiden, P. Warkentin
Experiences from a Large Industrial Circuit Design Application
ILPS, San Diego, Ca., October 1991

[FHK92] T. Fruewirth, A. Herold, V. Kuchenhoff, T. Le Provost, P. Lim, M. Wallace
Constraint Logic Programming - An Informal Introduction
In Logic Programming in Action LNCS 636, 3-35, 1992

[GJ79] M.R. Garey and D.S. Johnson.
Computers and Intractability: A Guide to the Theory of NP-Completeness.
Freeman and Co., pages 236-244, 1979.

[Ger94] C. Gervet
Conjunto: Constraint Logic Programming with Finite Set Domains
ILPS, Ithaca, NY., November 1994

[HE80] R. Haralick, G. Elliot
Increasing Tree Search Efficiency for Constraint Satisfaction Problems
Artificial Intelligence, 14:263-314, October 1980

[Hon93] H. Hong
RISC-CLP(R): Logic Programming with Non-linear Constraints
in [BC93]

[JL87] J. Jaffar and J.L. Lassez.
Constraint Logic Programming.
In Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages, Munich, 1987.

[JM94] J. Jaffar M. Maher
Constraint Logic Programming: A Survey
Journal of Logic Programming, 19/20:503-581, 1994

[JMS90] J. Jaffar, S. Michaylov, P. Stuckey, R. Yap
The CLP(R) language and System
IBM Tech report RC 16292, November 1990

[KS95] P. Kay, H. Simonis
Building Industrial CHIP Applications from Reusable Software Components
3rd International Conf. Practical Applications of Prolog
Paris, April 1995

[Kow79] R. Kowalski
Logic for Problem Solving
North Holland, New York, 1979

[Lau78] J. Lauriere
A Language and a Program for Stating and Solving Combinatorial Problems,
Artificial Intelligence, 10, 29-127, 1978

[LL91] B. Legeard, E. Legros
Short Overview of the CLPS System
PLILP 91, Passau, Germany, August 1991

[LW93] T. Le Provost, M. Wallace
Generalized Constraint Propagation over the CLP Scheme
Journal of Logic Programming, 16, 319-359, 1993

[Mac77] A. Mackworth
Consistency in Networks of Relations
Artificial Intelligence, 8(1):99-118, 1977

[MF85] A. Mackworth, E. Freuder
The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems

Artificial Intelligence, 25:65-74, 1985

[Mon74] U. Montanari
Networks of Constraints: Fundamental Properties and Applications to Picture Processing
Information Science, 7(2):95-132, 1974

[NM89] T. Nipkow, U. Martin
Boolean Unification - The Story So Far
Journal of Symbolic Computation, 7:275-293, 1989

[OV90] W. Older, A. Vellino
Extending Prolog with Constraint Arithmetics on Real Intervals
Canadian Conference on Computer and Electrical Engineering, Ottawa, 1990

[Rue93] M. Rueher
A First Exploration of Prolog III's Capabilities
Software- Practice and Experience 23, 177-200, 1993

[Sar93] V. Saraswat,
Concurrent Constraint Programing
MIT Press, 1993

[SC94] H. Simonis, T. Cornelissens
Modelling Producer/Consumer Constraints
ILPS Post Conference Workshop on Constraint Languages/Systems and their Use in Problem Modelling, Ithaca, NY, Nov 1994

[Sim92] H. Simonis
Constraint Logic Programming as a Digital Circuit Design Tool
COSYTEC Technical Report, 1992

[Sim94] H. Simonis
The Use of Exclusion Constraints to Handle Location Continuity Conditions
COSYTEC Technical Report, November 1994

[Sim95] H. Simonis
Modelling Machine Set-up Time with Exclusion Constraints
COSYTEC Technical Report, March 1995

[SD93] H. Simonis, M. Dincbas
Propositional Calculus Problems in CHIP
In A. Colmerauer and F. Benhamou, Editors,
Constraint Logic Programming - Selected Research, pages 269-285
MIT Press, 1993

[Sut63] I. Sutherland
A Man-Machine Graphical Communication System
PhD Thesis, MIT Boston, Ma, January 1963

[SS80] G. Sussman, G. Steele
CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions
Artificial Intelligence, 14:1-39, 1980

[Tsa93] E. Tsang
Foundations of Constraint Satisfaction
Academic Press, 1993

[VD91] P. Van Hentenryck and Y. Deville.
The Cardinaliy Operator: A New Logical Connective for Constraint Logic Programming.

In Proc. Eighth International Conference on Logic Programming. pages 745-759, Paris, France, June 1991.

[VH89] P. Van Hentenryck.
Constraint Satisfaction in Logic Programming.
MIT Press, Boston, Ma, 1989.

[VH91] P. Van Hentenryck.
Constraint Logic Programming
The Knowledge Engineering Review, 6(3):151-194, 1991

[VSD92] P. Van Hentenryck, H. Simonis, M. Dincbas
Constraint Satisfaction using Constraint Logic Programming.
Journal of Artificial Intelligence, Vol.58, No.1-3, pp.113-161, USA, 1992

[VSD92a] P. Van Hentenryck, V. Saraswat, Y. Deville
Constraint Logic Programming over Finite Domains - The Design, Implementation and Applications of cc(FD).
Technical Report, Brown University, 1992

[VC88] P. Van Hentenryck, J-P. Carillon.
Generality versus Specificity: an Experience with AI and OR Techniques.
In American Association for Artificial Intelligence (AAAI-88), St. Paul, Mi, August 1988.

[VG90] P. Van Hentenryck, T. Graf.
Standard Forms for Rational Linear Arithmetic in Constraint Logic Programming.
Proceedings of Internatioanl Symposium on Artificial Intelligence and Mathematics,
Fort Lauderdale, Fl., 1990

[Wal94] M. Wallace
Applying Constraints for Scheduling.
In B. Mayoh, E. Tyugu, J. Penjaam (Eds) Constraint Programming, Springer Verlag, 1994

[Wal72] D.Waltz
Generating Semantic Descriptions from Drawings of Scenes with Shadows.
MIT Technical Report AI271, November 1972