The CHIP System

COSYTEC

The Solution to Complex Problems

Limited Warranty:

This document is provided for information purposes only and subject to change without any prior notice. COSYTEC does not provide any warranties covering the information provided and specifically disclaims any liability in connection with this document.

Trademarks:

UNIX is a trademark of AT&T Bell Laboratories
MS-DOS is a trademark of Microsoft Corporation
Intel, Pentium, 486 are registered trademarks of Intel Corp.
Xwindows is a trademark of MIT
The CHIP C Library is the property of COSYTEC.

Authors:

Nicolas Beldiceanu
Helmut Simonis
Philip Kay
Peter Chan

Reference Number:

Reference: COSY/WHITE/002
Version: 1.2
Revision : A
Date : April 1997

COPYRIGHT:

COSYTEC SA
Parc Club Orsay Université
4, Rue Jean Rostand
F-91893 Orsay Cedex, France
Tel. +33 1 60 19 37 38
Fax. +33 1 60 19 36 20
Email. help@cosytec.fr

1. Introduction

In the last ten years, constraint logic programming **CLP** [JM94] [FHK92] 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).

2. The CHIP system

While constraints are the centrepiece of the CHIP application environment, a number of other components play an important role in developing applications. We now give an overview of this environment.

2.1 System Components

The CHIP system consists of a number of different components which together greatly simplify application development. The tool extends the functionality of a *host language* with constraints and other sub-systems. The languages supported are:

- The C++ version of CHIP is a library offering high-level constructs that encapsulates the constraint paradigm and enables easy re-use.
- The C version of CHIP takes the form of a constraint library which can be used by application programs, from within a conventional development environment.
- The Prolog based version of CHIP uses intrinsic language features like *unification* and *backtracking search* to achieve a deep integration of the constraints with the host. Constraints and search procedures can be easily defined and extended.

Other modules of the CHIP environment (see Figure 1) include a graphical library, interfaces to relational databases and foreign language interfaces. The different modules can be used together with an object modelling system to build large scale end-user systems [KS95].

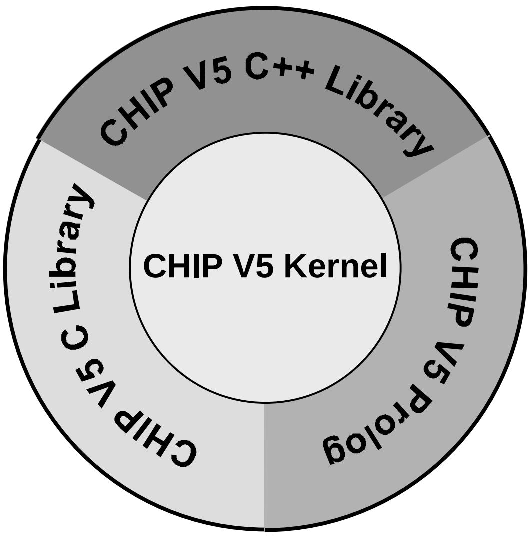


Figure 1: CHIP system components

2.2 History

CHIP [DVS88] was one of the first CLP systems together with PROLOG III [Col90] and CLP(R) [JL87]. It was developed at the European Computer-Industry Research Centre (ECRC) in Munich during the period 1985-1990. Different constraint solvers were developed and integrated in a Prolog environment.

At the same time the technology was tested on a number of example applications. The CHIP system is now being marketed and developed by COSYTEC. The early versions of CHIP were also the basis for a number of derivatives by ECRC shareholder companies.

BULL developed CHARME, a constraint based system with a C like syntax, ICL produced DecisionPower based on the ECRC CHIP/SEPIA system and Siemens is using constraints in SNI-Prolog. ECRC has also continued research on constraints. The current CHIP version V5 differs from the earlier constraint system by the inclusion of powerful global constraints [AB93] [BC94] as high-level building blocks.

3. The Finite Domain Solver

The finite domain solver is the central part of the CHIP system. This is due to its importance for many application domains, but also since many different types of constraints can be defined in this area.

3.1 Domain Concept

Constraints in the finite domain solver are expressed over domain variables, which range over a finite set of possible values. In CHIP, these domains are represented by finite subsets of the natural numbers.

3.1.1 Domain Definition

Several types of domains are provided in the following table for various host languages. The first statement defines a single domain variable *X* with domain from 1 to 10, i.e. all consecutive values are admitted. 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 within a domain, e.g. dis-equality. But as they do not have to represent the whole domain inside their data structure, they use less memory than explicit domains. The third example defines the variable *Z* with an domain which enumerates explicitly all admitted values.

The final example defines a aggregate containing three variables *A*, *B* and *C* which all have the same initial consecutive domain from 0 to 100. In C, this is the array. In C++, the ChipDvars class (which is a collection, which knows "append" methods) and in Prolog, this is the list.

<table>
<tr><td>C++</td><td><code>ChipDvar X(1,10);
ChipDvar Y(100,200, INTERVAL);
int values={2,4,7,9,42}; ChipDvar Z(values, 5);
ChipDvars array(3, 0,100);</code></td></tr>
<tr><td>C</td><td><code>DvarPtr X,Y,Z,array[3];</code></td></tr>
</table>

	`int values = {2,4,7,9,42}, size = 5;`
	`     c_create_domain(&X,1,10,CONSEC);`
	`     c_create_domain(&Y,100,200,INTERVAL);`
	`     c_create_domain_enum(&Z,values,size);`
	`     c_create_domain_array(array,3,0,100,CONSEC);`

Prolog	`X :: 1..10,`
	`Y :: 100:200,`
	`Z :: [2, 4, 7, 9, 42],`
	`[A, B, C]  :: 0..100.`

Table 1. Domain Variable Definitions.

3.1.2 Indomain

A 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 Prolog's built-in backtracking search 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.

The CHIP C Library offers the `c_labeling()` function which implements the usual set of enumeration schemes for an array of domain variables. For example:

```
c_labeling(array, size, METHOD_MOST_CONSTRAINED);
```

will enumerate the array of `size` variables using the most constrained strategy for variable selection, and use default methods for selecting values for each variable. The programmer also has full control over the value selection methods.

3.2 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.

3.2.1 Syntactic Methods

Two basic methods for finite domain constraints are *forward checking*, which is applied for dis-equality constraints, the second one is called *lookahead* and is applied to inequality constraints [HE80] [DSV87] [VH89]. Both are called *syntactic* methods, since they work by simplifying and rewriting constraints until they are satisfied and do not rely on a deeper understanding of the meaning of the constraint.

3.2.1.1 Numerical Constraints

Numerical constraints express equality and inequalities between domain variables. The constraint propagation uses a variant of the lookahead procedure. We now briefly explain its behaviour.

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 of 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.

The method can be easily generalised to linear terms over domain variables.

3.2.1.2 Symbolic Constraints

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

dis-equality

We explain the propagation of the inequality (i.e. different) constraint on a small example. Consider a dis-equality 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.

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 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

C/C++	`int array={10, 20, 5, 10, 5}, size=5;`
	`c_element(X, array, size, Y, TYPE);`
	`// TYPE=ELEMENT_COST or ELEMENT_MONOTONIC`

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

Table 2. Element constraint

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*.

Notice that in C/C++, it is possible to specify the type of propagation required for a given problem, leading to very performant applications. Some of the more complex constraints also allow this possibility.

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.

Other symbolic constraints, like *alldifferent*, *atmost*, *atleast* and *circuit* are available in CHIP. For most of them, stronger propagation results can be obtained by global constraints, as described in the following section.

3.2.2 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 three 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 circuits in directed graphs.

The constraints can be used and combined for many different problems and significantly extend the range of problems which can be handled with CLP. For space reasons, the other global constraints such as **among, precedence** and **sequence** constraints, will not described.

3.2.2.1 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 resources (e.g. 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.

Formulation

This manpower limitation can be modelled with a simple form of the cumulative constraint. For each of the *N* tasks, the procedure `define_task` will initialize domain variables *Si* for the start time, *Di* for the duration and *Ri* for the resource use of the task i:

C++	ChipDvars starts;
	ChipCumEntries e;
	for(int task=0; task<N; task++) {
	   ChipDvar start,dura,nbRess;
	   define_task(start,dura,nbRess);
	   e.append(ChipCumEntry(start,dura,nbRess));
	}
	ChipPost(ChipCumulative(e));

C	DvarPtr S[N], D[N], R[N], limit;
	for(task=0; task<N; task++)
	      define_task(S[task],D[task],R[task]);
	c_cumulative(N,S,D,R,limit);

Prolog	S= [S1, S2,..., Sn], D= [D1, D2,..., Dn], R = [R1, R2,..., Rn], define_task(S,D,R); cumulative(S,D,R,Limit).

Table 3. Cumulative Constraint

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 time point i between the start of the first task and the end of the last task, the total resource use by all tasks running at this time point is less than or equal to the available resource limit. We can visualise the meaning of the constraint with a little diagram. Figure 2 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.

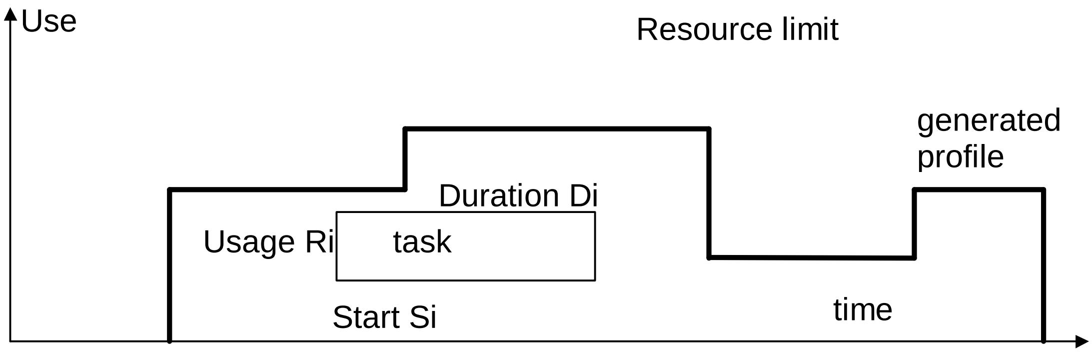


Figure 2: Cumulative resource profiles

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 of 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.

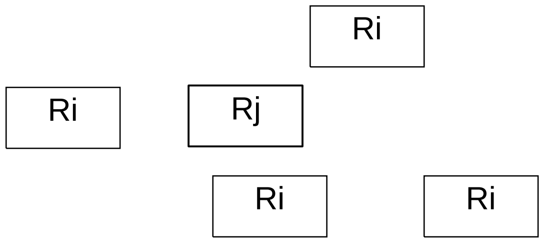
3.2.2.2 Diffn

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, at least one of these four conditions must be true:

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



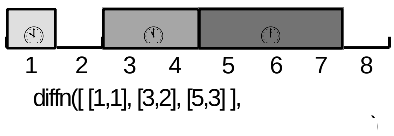
If we want to translate this constraint into primitive conditions, we have to handle a disjunction of four arithmetic constraints. Experiments have shown that we obtain very poor propagation results, although we generate a number of constraints which is quadratic in the number of rectangles. Using the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

C/C++	`c_diffn(n,m,rectangles);`
Prolog	`diffn([[X1, Y1, L1, H1], [X2, Y2, L2, H2],...,[Xm, Ym, Lm, Hm]])`

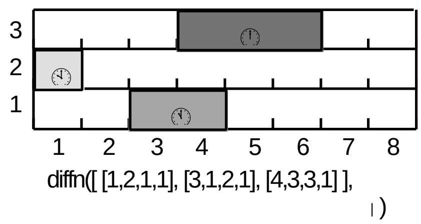
Table 4. Diffn Constraint

This constraint function takes as argument the dimension n: this allows to express the condition in higher dimensions as well, as illustrated below.

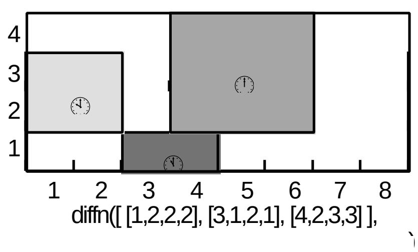
Usage



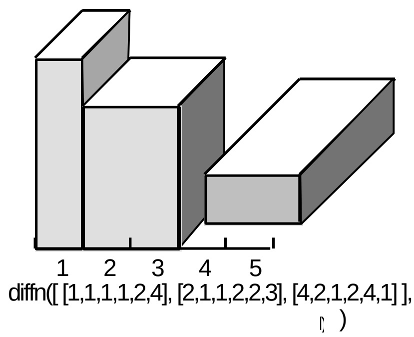
(A)



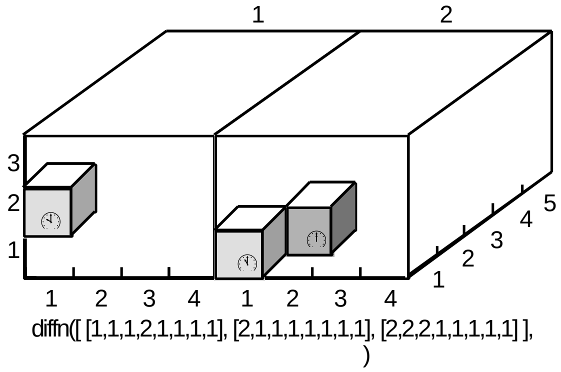
(B)



(C)



(D)



(E)

Figure 3: Diffn examples

The diffn constraint can be applied for many types of *packing* and *placement* problems in one or more dimensions. Figure 3 shows some examples of its use. The one dimensional case expresses cutting stock problems (Figure 3A). In two dimensions this constraint also corresponds to cutting problems of various forms (Figure 3C). In three dimensions, typical problems are packing of palettes or containers with rectangular boxes (Figure 3D).

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 container we place each box (Figure 3E).

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 3B, Figure 4). The non-overlapping condition states that no tasks can be scheduled at the same time on the same machine.

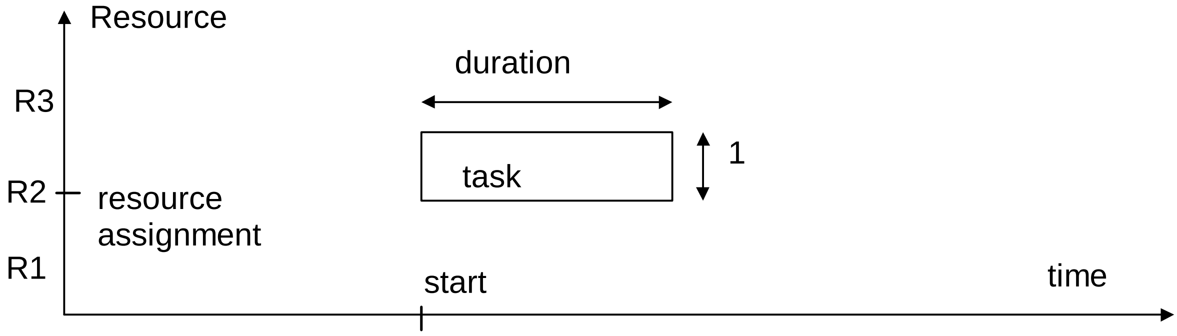


Figure 4: Machine assignment with diffn

The difference to the use of the cumulative constraint lies in the fact that we do 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 or *location continuity* for transport problems.

3.2.2.3 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, this problem corresponds to finding one or several circuits in a directed graph. Figure 5 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.

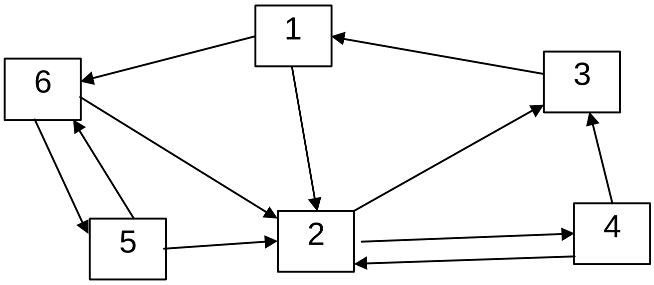


Figure 5: Directed graph example

We represent each node of 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 5, the variable V1 has domain [2,6] and the variable V2 has domain [3,4] for example. The program to find TWO circuits in this graph is shown below:

C++	`int val1={2,6}; ChipDvar X1(val1,2);`
	`...`
	`int val6={2,5}; ChipDvar X6(val6,2);`
	`ChipDvars X(X1,X2,X3,X4,X5,X6);`
	`ChipPost(ChipCycle(ChipDvar(TWO),X));`

C	`DvarPtr X[size], Nb;`
	`int val1={2,6}, val2={3,4}; val3={1},`
	`    val4={2,3}; val5={2,6}; val6={2,5};`
	`c_create_domain_enum(&X[0], val1,2);`
	`...`
	`c_create_domain_enum(&X[5], val6,2);`
	`c_create_domain(&Nb,    TWO,TWO,CONSEC);`
	`c_cycle(Nb, size, X);`

Prolog	`X1 :: [2, 6],`
	`X2 :: [3, 4],`
	`X3 :: [1],`
	`X4 :: [2, 3],`

	`X5 :: [2, 6],`
	`X6 :: [2, 5],`
	`cycle(TWO,[X1, X2, X3, X4, X5, X6]).`

Table 5. Cycle Constraint.

With standard labelling, the program finds the solution [2,4,1,3,6,5] which consists of two circuits, 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 also be expressed with the cycle constraint. An example is the handling of sequence dependent compatibility or set-up time conditions. Finally, 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].

3.2.3 User Defined Constraints

In CHIP, most constraints of a typical problem can be expressed with a combination of the built-in constraints explained above. But sometimes it is necessary to define specialised constraints as part of the application program. In CHIP, three different methods for such user-defined constraints are available.

A basic *coroutining* mechanism can be used for executing different parts of the program concurrently. In delay declarations, the user can specify when certain program parts should be executed depending on the binding of some variables.

Callback predicates can be linked to the modification of domain variables. As soon as the domain of a variable is updated, a user-written *update demon* routine is called. This technique is not only useful for writing constraints, but can be used for graphical output as well in order to visualise the propagation results inside a program.

Another method is the use of *conditional propagation* which calls some routine as soon as certain constraints are *entailed* by the system. The entailment check sees whether a constraint is always satisfied or unsatisfied by the current constraint set.

3.3 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.

There are different types of assignment strategies. The most common type is the *variable assignment* method, which repeatedly selects a variable and a value for the variable until either all variables are assigned or the constraints fail and the system backtracks to the last choice. We can decide either to select a variable deterministically and choose among its possible values or pick alternative variables non-deterministically and assign them to fixed preferred values. To simplify writing these search strategies, a number of primitives like *delete* are provided in CHIP. They help to define search heuristics like *first-fail* or *maximum regret* by selecting variables (or values) according to some criteria.

CHIP also provides primitives to search for an optimal solution of a problem. The *min_max* primitive implements a depth-first branch and bound search.

4. The Rational Constraint Solver

The rational constraint solver [Col90] [DVS88] provides a complete solver over a continuous domain. Rational numbers instead of floating point numbers are used since they offer exact representation and arithmetic.

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.

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 dis-equality 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.

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. Syntactical methods based on term manipulation and constraint simplification.

The rational constraint solver is often used for solving financial or economic problems. Another use is found for example in blending problems in the petro-chemical industry, where the solver is combined with *meta-programming* techniques and *constraint relaxation* methods to solve over-constrained problems.

5. Application examples

In this section we present some 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 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.

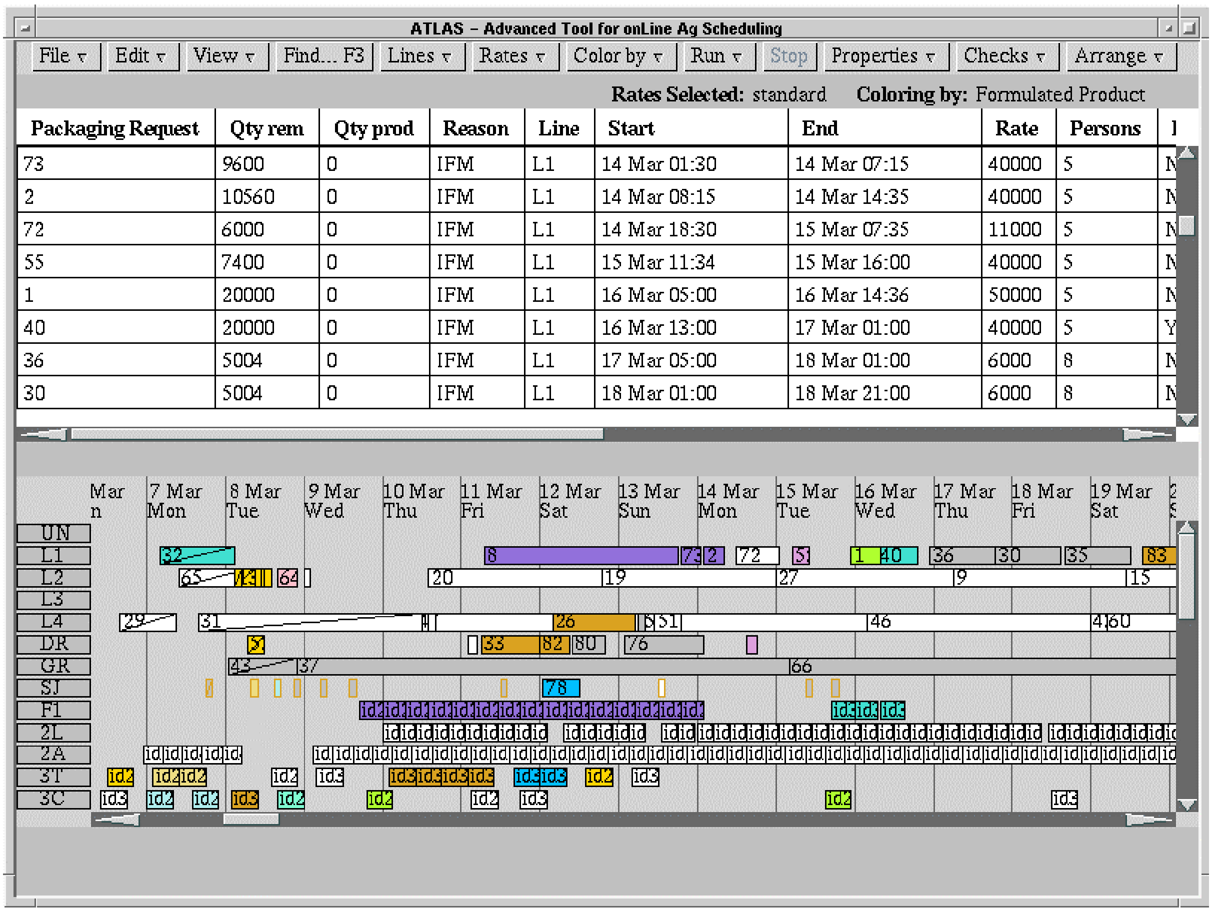


Figure 6 : The ATLAS application

In Figure 6, we show an example screen dump of the ATLAS application. The schedule is shown both in the form of a Gantt chart and as a textual list. The user can interact with the program via direct manipulation of items on the screen. A number of different views is provided to manipulate schedules, resource availability or stock levels.

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 flow rates and yields 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 used at two refineries.

TACT combines planning, scheduling and assignment aspects for the operational transport 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** 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.

The **LOCARIM** application is capable of designing a computer/phone network in large buildings. It is used by France Telecom to propose a cabling and estimate cabling costs for new buildings. 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 La Hague. This problem is highly constrained by the limited availability of transport containers and the required level of availability for the reactors.

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.

6. Summary

In this paper we have presented the constraint logic programming system CHIP. It combines a number of different constraint solvers in one environment and uses either Prolog, C or C++ as host language. Its technical advantage lies in the inclusion of several powerful global constraints to easily state complex conditions to efficiently solve large problem instances. These global constraints are very high level building blocks to develop applications declaratively and rapidly.

A significant number of real-life, custom built applications have been developed with CHIP. This shows the applicability of the approach to a wide range of combinatorial search and optimisation problems.

7. 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

[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

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

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

[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

[DVS88] 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.

[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

[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.

[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

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

[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

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

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

[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

[[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

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

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

[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