Survey: Practical Applications of Constraint Programming

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

September 1995

1 Introduction

Constraint programming is newly flowering in industry. Several companies have recently started up to exploit the technology, and the number of industrial applications is now growing very quickly. This survey will seek, by examples, to explain the current success of constraint technology, and, by showing its benefits, to add to that success.

In 1963 Sutherland introduced the Sketchpad system [118], a constraint language for graphical interaction. Other early constraint programming languages were Fikes' Ref-Arf [44], Lauriere's Alice [78], Sussmann's CONSTRAINTS [116] and Borning's ThingLab [10]. These languages already offered the most important features of constraint programming:

- declarative problem modelling
- propagation of the effects of decisions
- efficient search for feasible solutions

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

Constraint propagation was used in 1972 for scene labelling applications [130], and has produced a long line of local consistency algorithms [94, 84, 49, 60, 93, 31].

The topic of search has been at the heart of AI since GPS [97]. Influential ideas were generate and test [56], branch and bound [79], the A* algorithm [61], iterative deepening [76], tree search guided by the global problem structure [50], or by information elicited during search [87], and by intelligent backtracking [75].

Whilst there is "nothing new under the sun", the current flowering of constraint programming, owes itself to a generation of languages in which these three features are present in a new architecture that makes them easy to understand, combine and apply. The technology has matured to the point where it is possible to isolate the essential features and offer them as libraries or embedded cleanly in general purpose host programming languages.

For example isolating constraints as libraries has made possible the development of sophisticated constraint-based scheduling systems, see [134]. More generally there are commercially available libraries supporting constraint handling [68, 24].

On the other hand constraints fit hand in glove with declarative host programming languages. Three of the most influential constraint programming languages were embedded in Prolog (Prolog III [23], CLP(R) [70] and CHIP [32]). Whilst all three system are still developing further ([119, 73, 111]), there are many new constraint programming systems emerging (eg [22, 38, 63, 124, 4]).

From a theoretical point of view the extension of logic programming to *Constraint Logic Programming* (CLP) has been very fruitful: for example ALPS [85] - a form of logic programming with guards - was an extremely influential language, becoming the forerunner of the *Concurrent Constraints* paradigm [107]. Concurrent constraint programming has in turn provided a very clean model of concurrent and multi-agent computing [108, 63]. Constraints can also be modelled in terms of information systems [104], which allows us to reason about the behaviour of constraint programs at an abstract level. One application of such abstract reasoning, abstract interpretation, is already paying dividends in the global optimisation of constraint programs [74].

However the focus of this paper is on the success of constraints in building practical software solutions to end-user problems. We shall examine these successes in the next three sections, corresponding to the three features of constraint programming listed above.

Modelling One major contribution of constraints is to problem modelling. It has been claimed that "constraints are the normal language of discourse for many applications". Whilst this advantage pays off in all applications, it is central to the design and verification of VLSI circuits and to the specification, development and verification of control software for electro-mechanical systems. These applications are discussed in the next section.

Propagation A second major contribution is that constraints offer a natural way for a system to spontaneously produce the consequences of a decision. This

facility is often called *propagation* of information. (*Propagation* is defined in the dictionary as "dissemination, or diffusion of statements, beliefs, practices)"). Propagation is the most important form of immediate feedback for a decision-maker.

Propagation works very effectively in interactive decision support tools. In many applications constraint programming is used in conjunction with other software tools, taking their results as input, performing propagation, and outputting the consequences. Typically feedback from the propagation tool is given in the form of a spreadsheet interface. Some constraint-based spreadsheet applications are discussed in section 3.1 below.

Many early applications of constraint programming were related to graphics: geometric layout, user interface toolkits, graphical simulations, and graphical editors. Constraint propagation has played a key role in all these applications, with the result that control over the propagation has been thoroughly investigated: leading to a current generation of very high-performance constraint-based graphics application. Some of these applications are discussed in section 3.3 below.

Search The third, and most important, area of practical applications of constraints is in solving hard combinatorial problems. These problems are addressed by the integration of constraint modelling, propagation and search. Successes have typically been in planning and scheduling, resource allocation, routing, and more recently even in complex numerical problem solving. The applications are similar to those addressed by mathematical programming, with the difference that mathematical programmers seek a clean model of the problem - or often a simplified abstraction of the problem - whilst constraint programmers revel in the messy details of practical problems! Some applications which involve solving combinatorial problems are discussed in section 4 below.

2 Applications using Constraints for Modelling

Constraint programming is a descendant of declarative programming. Constraints are defined using a logical formalism, in a way similar to functional or logic programming. In declarative programming, or any runnable specification language, there is a *default* evaluation algorithm that turns the logical formalism into a running program. The drawbacks of a default algorithm are two: poor performance on many (indeed most) programs and frequent failure to terminate at all.

In constraint programming, by contrast, the behaviour of the constraints is equally as important as their definition. Constraints have two facets, definition and behaviour, and these can be handled separately. Once the definition is correct, the behaviour can be sorted out. The practical consequence is that the

programmer can concentrate on modelling her problem and the problems with performance and termination can be ironed out afterwards.

Thus although constraints make no new contribution to problem modelling, they ensure that the encoding of a correct model of the problem is indeed a step towards an efficient running program. The consequences can be revolutionary - with programmers actually taking modelling seriously!

2.1 Constraints as Relations

The simplest formal model of a constraint is a relation. For example in a resource allocation problem, the constraints on a task state which resources can be used to do the task. This constraint can be modelled as a binary relation, such as $canUseResource$, between tasks and resources.

In this paper we shall use names beginning with a capital letter, eg *Resource*, for variables. Values start with a small letter, eg *resource1*. Also relation names start with a small letter, and we shall use the syntax of logic programming. For example the fact that the relation $canUseResource$ holds between the task $t1$ and the resource $r1$ is written

$$canUseResource(t1, r1).$$

To express the constraint that the relation $canUseResource$ must hold between any feasible values of $T1$ and $R1$ we write

$$?- canUseResource(T1, R1).$$

For certain mathematical relations, such as $>$, an infix notation is used. For example $?- S1 > S2$ constrains any feasible values $s1, s2$ of $S1$ and $S2$ respectively to satisfy $s1$ greater than $s2$.

The advantage of such a model is that the same model underlies relational databases. Consequently it is easy to design a system in which the constraint definitions are extracted from the database.

The applications presented in this section involve complex constraints which are modelled as relations but for which the relation is not best defined by a table, as it would be in a relational database. Instead the relation is best defined intentionally. The following is a very simple example of such a constraint, from the field of circuit design and verification. In this application variables can take only two possible values 1 and 0. Here is a constraint that requires its third argument to be the conjunction of its first two:

$$and2(In1, In2, Out) \leftrightarrow Out = In1 * In2$$

This constraint is defined intentionally, in terms of a built-in binary multiplication function $X * Y$ and a built-in equality $=$.

2.2 Circuit Verification

2.2.1 Combinatorial Circuits

There exists a specialised algorithm for handling the boolean functions $X * Y$, $X + Y$ and $\neg X$ and the boolean equality $X = Y$. It handles any number of such constraints, extracting the global consequences of the complete conjunction of constraints. The algorithm is based on a special data structure *Binary Decision Diagrams* (BDD's) to represent the constraints [13] and employs a decision procedure *Boolean Unification* to extract the "most general solution" to any set of constraints defined in terms of booleans functions and relations. [15, 99].

The first specific application which directly uses constraints as a modelling tool is VLSI circuit verification. The idea behind circuit verification has its roots in an idea of Sussman [117]: complex systems can be broken down and modelled in quite different ways - into functional components, physical components, causal sequences etc. If the different models can be expressed in a common, constraint-based, formalism, then we can learn much from the interaction of these different models. For circuit verification the different models - the behavioural model and the functional model - should be equivalent. Verification reduces to proving the equivalence of the two models.

Circuits whose outputs are completely determined by their inputs are called "combinatorial" circuits. The behaviour of a circuit is modelled as a single global constraint, built from the constraints which model the behaviour of its components. For example a full-adder in a digital circuit imposes constraints between its inputs and outputs which are best captured in terms of the constraints imposed by its component gates as follows [113]:

$$
\begin{array}{ll}
full\_adder & (XIn, YIn, CarryIn, Sum, CarryOut) \leftrightarrow \\
& and2(XIn, YIn, C1), \\
& xor(XIn, YIn, S1), \\
& and2(CarryIn, S1, C2), \\
& xor(CarryIn, S1, Sum), \\
& or(C1, C2, CarryOut).
\end{array}
$$

The full-adder is, in turn, used as a component in more complex circuits, such as the n-bit adder. In fact the use of recursion allows a single definition of an n-bit-adder to be used for any value of n as follows:

$$
\begin{array}{ll}
n\_bit\_adder & (XBits, YBits, CarryIn, SumBits, CarryOut) \leftrightarrow \\
& XBits = [], YBits = [] \quad | \\
& \quad SumBits = [], CarryOut = CarryIn; \\
& XBits = [X|XT], YBits = [Y|YT], SumBits = [Sum|SumT] \quad | \\
& \quad full\_adder(X, Y, CarryIn, Sum, Carry), \\
& \quad n\_bit\_adder(XT, YT, Carry, SumT, CarryOut).
\end{array}
$$

This is an example of a *guarded* clause: if there are no more bits in the input, the first guard $XBits = [], YBits = []$ succeeds, but if there are more bits, the

second guard
$XBits = [X|XT], YBits = [Y|YT], SumBits = [Sum|SumT]$
succeeds. When a guard is satisfied, the constraint is replaced by the set of goals appearing after the guard: in the latter case this is
$full\_adder(X, Y, CarryIn, Sum, Carry)$,
$n\_bit\_adder(XT, YT, Carry, SumT, CarryOut)$.

VLSI circuits are built up from a hierarchy of subcircuits, and their constraints are in turn built from the constraints associated with their subcircuits, using intensional definitions such as that above. Ultimately all the constraints are built from the underlying boolean functions we encountered in the previous section. The Boolean Unification algorithm for handling this class of constraint automatically extracts the most general solution to this global constraint: the solution is a representation of the behaviour of the whole circuit for any arbitrary input.

This representation can be used in two ways. Firstly if the specification of the intended function of the circuit is also expressed as a constraint, then the functional constraint can be compared with the behavioural constraint.

Secondly it is very often necessary to compare two circuits which are intended to show the same behaviour (in case, for example, a circuit is modified to use newer components). Again the requirement is to compare the two constraints.

The method of comparison is based on the assumption that the constraints produce unique outputs for each set of inputs. The comparison is done by using the same set of input variables for each constraint, and adding a new set of output variables, one for each output of the circuit. The value of each output variable is 0 if the two constraints produce the same output and 1 if the outputs are different. On combining the two sets of constraints (i.e. adding them into a common constraint store), if they are indeed equivalent the "most general solution" produced by the Boolean Unification algorithm will have a $V = 0$ for each new output variable V. Another constraint-based procedure for verifying the equivalence of digital circuits was recently proposed by Hooker and Yan [66].

2.2.2 Sequential Circuits

Most VLSI circuits have not only inputs and outputs but also an internal state. The output of the circuit depends on the inputs and the state. Similarly the inputs and previous state determine a new internal state. Circuits with an internal state of this kind are called "sequential" circuits.

The internal state is defined by the setting, 1 or 0, of a set of internal "pins". The number of pins is finite and so such a circuit can be modelled as a finite state automaton. The number of internal pins may be in the hundreds, and the number of possible states can reach 10^{20} and beyond [14].

The method of verifying sequential circuits is essentially the same as for combinatorial circuits, except that now it is necessary to establish that the same sequence of inputs produces the same sequence of outputs. In comparing

two sequential circuits the state pins are treated as inputs - except that they have fixed values in the initial state. However after ensuring that two sequential circuits are equivalent in their initial states, the system must now go on to check, for each input, that the resulting states are equivalent again. Each subsequent pair of states must be checked for equivalence until the sequence of inputs returns both circuits to their initial states.

The application of CLP to circuit verification has been commercialised by Siemens. The Siemens product is used internally at Siemens and sold externally as their "Circuit Verification Environment" CVE2 [8]. The constraints reflecting the behaviour of a range of standard cells are offered as libraries. New cell constraints can be added using logic programming. The structural specification of circuits - how they are built out of cells - is expressed in an industry-standard language called EDIF [39]. CVE2 automatically generates the circuit constraint from the EDIF specification and the cell constraints.

2.3 Real Time Control Systems

Constraint programming is now being exploited for building control software for electro-mechanical systems with a finite number of inputs, outputs and internal states. As with sequential circuits, each component of a complex system is only connected to a small part of the overall state of the system, and its behaviour can be captured quite simply, but when the system is considered as a whole the number of global states becomes extremely large.

Control systems are needed for applications ranging from lifts, photocopiers and car engines, to assembly lines and power stations. The applications currently being tackled with constraints technology are those at the smaller end, where it is still possible to prove that with a given control certain global properties of the system can be guaranteed. Typical properties that must be proved about the system are safety properties (eg a lift should never start to move while the doors are still closing) and fairness properties (the lift must eventually answer every request, no matter what subsequent requests may be received).

Each component of a system is first modelled in terms of its inputs, internal states and outputs. Next the possible transitions for the component are defined. Each transition is specified in terms of its preconditions, which must hold before it can fire, and its postconditions, which record what holds in the resulting state after it has fired. The precondition is a logical expression dependent on the current environment (the inputs and internal state of the component and its neighbours), and the postcondition is a set of outputs and a new internal state. Finiteness of the model is ensured by discretising the variables, so that each variable can only take a finite number of values.

The frame problem is avoided by having the environment defined by a finite set of orthogonal atomic values (inputs and internal state values): anything which is not explicitly changed by a transition retains its old value in the next state.

From such a model of the components it is possible to generate a *Finite State Automaton* (FSA) which encodes the behaviour of the whole system, and which satisfies the specification of each transition for each component. If however the constraints prevent some transition from ever taking place, the FSA compiler reports an error. The required properties of the system are expressed in a language which allows reasoning over states. Safety properties are typically properties that must hold of individual states; fairness properties demand the existence of a future state satisfying some property, after an initial state in which some input occurs. A constraint solver is used to prove that the FSA satisfies these properties. The same Boolean Unification solver is applied here as was applied for sequential circuits - indeed the two applications are similar in that they require properties to be proved of FSA's.

An extra aspect of control systems is that some assumptions are required about the effects of outputs on future states and inputs to the system. These assumptions alo involve formulae involving multiple states, and as a consequence proofs involving these assumptions are difficult. Even on a toy production cell example, current systems require up to 5 minutes to prove liveness properties involving just one or two such assumptions.

One such commercial system is the system verification environment SVE [45].

There is another role for constraint programming in control systems, and that is as a language *Timed Concurrent Constraint Programming* (TCC) for defining the behaviour of the components [106]. This language uses constraint agents both to express reasoning about the current state, and to specify agent behaviours across multiple states. The first facility makes it possible to define complex preconditions for a component transition. In particular a precondition may refer to the absence of something from the current state. To ensure the language can be compiled into a FSA, syntactic restrictions are imposed which preclude the use of recursion, for example, in defining preconditions. Support for complex preconditions makes it possible to define component behaviour without having to specify all possible internal states of a component (and its neighbours) with complete understanding in advance of the component's transitions. For example in [45] for defining one transition of a certain component, it was necessary to add an extra state *not low* to record that a component was not in the state *low*. In TCC, because it is possible to use the fact that a component is *not* in a certain state as a precondition for a transition, there is no need to add this explicitly as a new internal state.

The second facility, specifying agents behaviour across multiple states, also avoids the programmer having to introduce artificial internal states. A simple example from [106] is

```
sensor(Loc) :: always (power -> at(Loc):paper ->
                                  next (at(Loc):paper -> jam))
```

This sensor component has the behaviour that, whenever the power is on, if

the paper is at the same location in two successive states, then it outputs *jam*. Without the facility to include two states in the specification, it would be necessary to add an internal state for the sensor recording that paper was at a certain location in the previous state.

The third contribution of constraint programming to the specification of component behaviours is that it allows independent hierarchies of components to be specified *independently*. Using constraint agents within the definition of other constraint agents, a component's behaviour can be expressed in terms of the behaviour of its subcomponents. Also local variables allow similar components to be specified without interference.

The consequence is that components can be substituted for each other in different systems and it is simple to define their behaviour in each system. For example it is not necessary to rewrite the transitions of a component every time its neighbours are replaced by others.

This flexibility is becoming crucial with the increasing demand for faster time to market, and for greater flexibility in product lines. For example traditional product development methods for reprographic machines (photocopiers, printers, faxes machines etc.) involve dozens, if not hundreds, of mechanical, electrical, software and systems engineers [129].

One final horizon on the application of constraint programming to control systems: instead of proving properties of a FSA compiled from the constraints, it becomes possible to prove more and more properties from the constraints themselves. Already, using TCC, it is possible to encode safety properties as preconditions of transitions. This ensures that these properties will be satisfied in any FSA generated from the TCC specification, without further proof. The next horizon is to develop a proof procedure for proving fairness and reachability properties from software specifications expressed as constraint programs.

2.4 Constrained Objects

A section on modelling cannot be complete without some discussion of object data models. In the database world, many advantages are perceived for object databases over relational databases. However relational databases have much more developed facilities for maintaining integrity constraints [120].

When managing integrity constraints in object databases, the difficulty is that an update to one object may violate constraints on another object. Consequently integrity checking must be applied to the result of a whole transaction which may involve many objects. This violates the requirement that the admissible behaviours of an object are defined by its interface alone: an atomic behaviour is not the result of sending of a message to a single object but is instead a whole transaction [128].

Constraint programming, however, has a very well-developed notion of a constrained object. The simplest constrained object is a variable: its possible values are given by the constraints upon it. A variable is, of course, a very

different kind of object from those found in object databases. But constraint programming admits more complex constrained objects. For example a *resource* in a scheduling application is a constrained object. A typical constraint on resources is that no two tasks can use the same resource at the same time. A vivid example from an existing application is that no two trains can run on the same track at the same time [57].

A typical constraint applicable to an object is to constrain a property (or slot) of the object. For example a particular parking bay may not admit vehicles of more than a certain dimension. Such a constraint may be defined on a particular object, or on a whole class - all parking bays of a certain class for example [68].

A first proposal for integrating constraint and object models is in [12], and an application of constraints in a visual interface to object-oriented databases is [30]. Indeed object-oriented constraint programming has been applied to graphical interfaces, and editors, for many years as we shall see in section 3.3 below.

3 Applications using Constraints for Producing Consequences

Constraint stores are sets of constraints whose global consistency is checked by a constraint solver. In the applications of the last section, the constraint store was the set of boolean constraints, and they were solved using boolean unification.

Constraint agents are processes that continually access and update the constraint store. A simple example of such a constraint agent is one that might be used in any application for making diagrams or drawings on a computer. If the end-user is drawing a vertical line, then as she moves the cursor down the screen a constraint agent extends the vertical line keeping its end at the same horizontal level as the cursor. Another example of a constraint agent, in quite a different context is one that updates the domain of the variables X and Y to reflect the constraint that $X \geq Y$. The constraint agents have the role of keeping the constraint store in step, by reflecting the effect of their constraints due to changes in the constraint store. This behaviour is called propagation.

Constraint propagation is used in two quite different ways in constraint programming. Firstly it is used to reflect the consequences of adding some new information to the constraint store. This information is a refinement of the store, and the result of propagation is further refinement of the store. If the new information, or the result of propagation, is inconsistent with the previous information in the store, then all propagation ceases: effectively the inconsistent store is over-defined and nothing more can be added to it. The agent reflecting the constraint $X \geq Y$ above is an example of this kind of propagation.

Secondly constraint propagation is used to reflect the consequences of updat-

ing the constraint store. In this case the new information effectively overwrites any previous information that was inconsistent with it. Propagation adds new updates to the store so as to achieve a steady state where all the consequences of the update have been made explicit in the new constraint store. The constraint maintaining the position of the end of the vertical line, described above, is an example of this second kind of propagation.

3.1 Constraint-based Spreadsheets

The spreadsheet is one of the most prevalent decision support software tools in use today. The current generation of spreadsheets suffer from two limitations. Firstly they only compute in a fixed directions: certain cells are designated as input cells and the others, whose entries are defined as functions of the input cells, are the outputs. The second limitation is that the outputs can only be defined from precisely defined inputs.

The idea of applying constraint technology to overcome these limitations of spreadsheets has been taken up in several systems, for example in the Visilog European collaborative project. Indeed the spreadsheet metaphor has been built into some constraint programming systems as a generic interface [40]. The commercial spreadsheet market is not yet dominated by constraint-based spreadsheets, but there are a number of practical applications based on this very idea.

The *Short Term Planning* (STP) application at Renault is an excellent example of constraint-based spreadsheets in action [20]. The problem is to assign product orders to factories so as to minimise transportation costs for delivering the finished products (cars!) to the customer. According to the above description, this is a standard OR problem (which can be formulated as a *transportation* problem and solved by a linear programming algorithm). However (as usual) the real problem is more complicated, involving a variety of *side-constraints* which have to be satisfied in any solution. Moreover the problem as formulated is typically unsolvable: some of the constraints have to be relaxed, but there is no fixed priority order, or cost function, which can guide the choice of which constraints to relax. Indeed the context of the problem continually varies, so that the objectives change from month to month.

The decision as to what constraints to relax is based on an evaluation of what can be modified at the production level. It requires compromises based on some kind of negotiation: consequently it is up to the end user to take such a decision.

A constraint-based spreadsheet was developed to help the user in this task [20]. The spreadsheet presents to the end user the production figures at each factory. For each combination of car model, version, country and transmission, it shows the number of vehicles allocated to each factory. However during the planning process, the figures held in the spreadsheet are not precise numbers, but intervals: $[Min, Max]$. Constraints are attached to different cells and are

used to propagate the consequences of modifications imposed by the user.

The user can either enter a specific value for a given cell, or simply restrict the range of possible values, by narrowing the interval. Other interfaces allow the user to define new side constraints. As the user progressively narrows the intervals and/or defined new constraints the system propagates the consequences making them explicit in the spreadsheet: either an inconsistency is flagged, or tighter bounds are deduced for the constrained variables.

The user continues to adjust values and constraints until the problem formulation is satisfactory. Then she invokes a linear programming algorithm to produce an optimal solution, with regard to manufacturing and transportation costs, within the selected intervals and satisfying the selected side-constraints.

Another very promising application of constraint-based spreadsheets is to financial planning [67]. The spreadsheet allows the financial planner to explore potential investments. Current spreadsheets support a trial and error approach, but interval spreadsheets allow the planner to start with a fairly general planning policy, and to refine this "top-down" into a plan by narrowing intervals, and seeing the effect on the other cells in the spreadsheet.

Financial applications often involve non-linear constraints which are extremely hard to handle with black-box mathematical programming techniques. Reasoning on intervals provides a handle on non-linear problems, since it is possible to calculate the maximum and minimum result of a function - even a non-linear one, given maxima and minima for each of its inputs. Nevertheless the intervals must be narrowed to a single point before interval reasoning can guarantee the consistency of a set of constraints.

Thus the top-down refinement of intervals in a financial application serves two purposes: firstly it refines an investment plan into a set of specific investments, and secondly it extracts a set of numeric values that satisfy the constraints on the problem. Sometimes, of course, a refinement will violate the constraints, and then it is necessary to widen the intervals again. Typically inconsistency is dealt with by backtracking, but another option is to relax the intervals in a way that is driven by the violated constraints [67].

3.2 Explanation and Debugging

Mixed initiative problem solving is essential in a whole variety of practical applications: the end user must be able to cooperate with the software in developing solutions. To date much software has operated as a black box producing a solution like the magician's rabbit out of his hat! Constraint propagation supports precisely the kind of feedback which makes mixed initiative problem solving possible.

After each search step, the consequences of the particular choice are made explicit by the propagation engine. The user can then either reject the choice, and step back; or she can use the propagated information to help guide her next

choice; or she can allow the software to continue the search automatically - until she interrupts it and takes over control again.

For example constraint programming was used to address the timetabling problems of Banque Bruxelles Lambert [129, 37]. The system implementor (DECIS) writes the following about user interaction.

> "It seems to us that the end user of any timetabling application should be allowed to interact with the construction of the timetable as some information will necessarily require *human involvement*, be it the choice of constraints to relax when the problem is impossible or too hard, in *hand-made* assignments indicating last minute changes in the data, or simply human initiative to drive the search for a valid timetable.
>
> In order to allow such human involvement in the resolution process we need two ingredients:
>
> - a graphical interface allowing ergonomic presentation of the partial results during the search process and an easy modification of the data structures
> - hooks in the resolution engine enabling the user to pause during search, modify the configuration and continue the search."

The importance of mixed initiative programming in practical applications is confirmed in the proceedings of two industrial CLP user group meetings [25, 69]. For example the assignment of cashiers for El Corte Ingles in Spain includes both assignment - by a constraint program - and consolidation - by the end user. This consolidation is then fed back to the constraint program and used to constrain further assignments [43].

The application of constraint programming to gate allocation at Changi airport supports several different Gantt chart displays (as well as over one hundred other interface screens!). Its interactive requirements are summed up in the following terms [5]:

> "The resource assignments can be manipulated graphically on the Gantt chart... The actions allow users to provide additional information to the system and to override the systems allocations...The system can be used in automatic, advisory or manual mode. In automatic mode, the system will automatically reassign gates to maintain a 'good' set of assignments. In advisory mode, the system prompts the operator on the responses required. In manual mode, the operator is completely free to modify the assignments as (s)he wishes."

In the application of constraints to aircraft production scheduling *interactivity* is a basic requirement [18]. The SAVEPLAN production management

software package uses constraint programming for its scheduling component because it supports an interactive user interface [55]. A crucial issue in scheduling for the process industries is how to handle over constrained problems [83] - the very same problem faced by Renault in the STP application (above).

The feedback from constraint propagation must be presented vividly to the end user. The spreadsheet approach is just one example of a user interface. More typically the state of the constraint solver will be presented to the end user within a graphical interface that reflects her specific application. For example an application to vehicle scheduling will probably reflect the planned positions of the vehicles on an electronic map. A scheduling application will typically use a Gantt chart, as well as graphs reflecting resource utilisation, work in progress and so on.

Very broadly the state of the solver is reflected in the value - or the remaining possible values - of the *driving variables* of the problem. These variables are the ones whose final values define a complete solution.

When presented to the end user, for the purpose of mixed initiative problem solving, these variables are embedded in an application-specific graphical model of the problem. However the very same information, and the same facility for the user to take over the initiative, is extremely useful for developing constraint programs. In this case it is not essential that the variables be embedded in a sophisticated interface. They can be placed in a multi-dimensional array, or in a set of matrices. The role of the feedback tends less to explanation and more to debugging - though this is not a very clear distinction. The user now explores different choices - which variable to label next and what value to give it - different constraints - local, global, redundant - and different constraint behaviours - passive checking, forward checking, or full arc-consistency enforcement. From the results of her interactive problem solving, the user learns which automatic techniques work well on the current specific application. A system supporting this kind of "performance debugging" is described in [88].

3.3 Constraint-Based Graphical Interfaces

The most venerable application of constraint programmng is its application to graphical interfaces. The role of constraint propagation in this application is to keep all the graphical objects in the correct relation to each other, after some update to the graphical window - typically via a mouse.

The objective in this application is no longer to find an optimal solution, but to find a "natural" solution, and to find it relatively quickly. In many cases this is quite straightforward. The first example in the current section - keeping the end of a vertical line on the same level as the cursor - can quite easily be handled by sampling the value of the Y-coordinate of the cursor, and extending, or contracting, the line so it ends at the same Y-coordinate.

However more complicated problems quickly arise in case the graphical objects are constrained with respect to each other. For example in an electronic

room planning application, it might be possible to drag a graphical chair across an graphical room, but the constraints should prevent it from occupying a position which overlaps with the position of any other piece of furniture in the room [64].

Constraints also provide a way of stating many user interface requirements, such as maintaining consistency between the underlying data and a graphical depiction of that data, maintaining consistency among multiple views, and specifying formatting requirements.

A commercial constraint library for constructing interactive graphical user interfaces is presented in [9]. Object Technology International has used the library to build a graphical editor for a large database. technology for graphical editors. Most modern tools, be they database browsers, calendar managers or systems for producing overhead projector presentations, have some sort of graphical editor. Simple graphical editors can be implemented quite straightforwardly, but it is much harder to construct graphical editors in which the underlying semantic objects have complex relationships which must be modeled in the manipulation of the graphical objects. However, with constraints for modelling these objects, and a built-in constraint solver, these editors become extremely easy to build.

The propagation of constraints in graphical interface applications has to satisfy two special requirements:

- It has to be extremely efficient, to support low latency and high bandwidth feedback during direct manipulation [86]
- It has to propagate changes, not just refinements, and inconsistencies must be handled by making further changes [48]

To meet the first objective it is necessary to handle the constraints as functions. However a prerequisite for handling a constraint in this way is to know in what direction it propagates information. This directional information has been called the *dataflow graph* of the constraints. Building the dataflow graph is typically more expensive than using it to propagate the actual data values, and efficient propagation depends upon reusing the same dataflow graph whenever possible. In a graphical interface application, the dataflow graph is typically built when the user first grabs some graphical object using the mouse.

The dataflow idea works best when there are no cycles in the graph. However the new propagation algorithms are increasingly able to handle cycles [103]. For example sets of linear equations and inequations - which cannot in general be reduced to a cycle-free constraint graph - can still be normalised so that minimal work is done to reflect the effects on the remaining variables of changing the value of one specific variable. Again, this normalisation process is expensive and should be done as few times as possible [62].

To meet the second objective - propagating changes so as to eliminate inconsistencies - it is necessary to establish which values can be changed and which

constraints can be relaxed. This is essentially the knowledge revision problem [53], emerging in a very practical context. The two main approaches to solving it in the context of constraint programming are

1. to impose a hierarchy on the set of current constraints (the value of a variable is here viewed as an equality constraint on the variable)

2. to define a cost function which associates a global cost with the set of changes from the previous state, and to minimise that cost

Work on propagating changes under complex constraints, including inequality constraints for example, where the constraint graph contains cycles, is still producing exciting new results. As yet the technology has not scaled up sufficiently to allow such general constraints to be tackled in large graphical applications such as commercial CAD systems (but see [90] for a specialised application, and [54, 36] for ongoing developments).

4 Applications using Constraints for Solving Combinatorial Problems

The success of constraint programming on combinatorial problems is due to its combination of high level modelling, constraint propagation and facilities for controlling search behaviour.

Combinatorial problems are those where a solution results from taking a whole series of interdependent choices: the correctness or optimality of a given choice is not usually apparent until a number of other choices have been made, which may in turn depend on further choices. The result is that the correctness of the very first choice is not usually confirmed until the last choice has been made. Typically each choice admits only a finite number of alternatives (otherwise there would be no chance of finding the best choice).

4.1 A Few More Words about Modelling

For many of the applications described in this section the constraint program is short. For example the program for Digital Signal Processing in cc(FD) [121] shows excellent results in actual applications and comprises just four pages of code. There is nothing new about high-level programs being short. However these short programs have two properties which are quite rare in combination: they are both intelligible, in the sense that the problem specification is evident from the code, and they are efficient.

The first consequence of this combination is that constraint programming can be used to explore different ways of solving a problem very quickly. This is reflected in the speed with which constraint programming approaches have caught up with, and overtaken, special purpose codes for job-shop scheduling

[17]. The second consequence is that programs can be quickly modified when the application changes. Constraint programming code is genuinely reuseable [72].

4.2 Backtrack Search under Constraints

Backtrack search augmented by constraint propagation is a basic technique for tackling combinatorial problems [96, 125]. Each choice made by the search engine, imposes new constraints whose consequences are extracted by the propagation engine. The information yielded by propagation is then used to focus the remainder of the search. In case the constraints become inconsistent, the current choice is abandoned and the search engine makes another choice. Many choices are ruled out in advance as a result of propagation. When all the remaining alternatives for a given choice have been tried unsuccessfully, the search engine backtracks to the previous choice.

An excellent tutorial by Pascal Van Hentenryck describing how to augment the search with different kinds of constraint propagation is [122].

Early practical applications of constraint programming used built-in constraint propagation algorithms which approximated arc-consistency. Examples, implemented in CHIP [32], included scheduling tasks on a construction project [34], sequencing of cars on an assembly line [35] and optimal cutting of raw materials to satisfy customer orders [33]. Whilst these applications were excellent demonstrations of the potential of constraint programming, it soon became apparent that these demonstration programs would not scale up to large applications [28]. The built-in propagation reduced the depth in the search tree that the backtrack search algorithm reached before detecting failed branches, but even the reduced search tree could grow too large to be completely explored in any realistic timescales.

Nevertheless arc-consistency techniques and backtrack search have sufficed for a number of practical appliactions of constraint programming. Three examples are a university timetabling application [47], an industrial disposing problem [6], and a vehicle scheduling system [80].

4.3 Specialised Constraint Behaviours

For different applications different constraint behaviours are appropriate. Thus for circuit diagnosis, instead of the standard arc-consistency algorithm, CHIP used constraints called "demons" with a special behaviour [112], and for simulation of hybrid circuits CHIP used constraints called "forward rules" with another special behaviour [58].

A nice example of specialised constraint behaviour comes from an application to the design of keys and locks. The problem is, given a site (with typically 1000-10000 keys), how to design the different keys and locks so each key opens a required set of locks. For this application it was possible to "compile" the

constraints yielding an eager algorithm producing solutions guaranteed to satisfy the constraints [27].

Another application, developed by the same author, is an aid for cartographers [28]. The cartographer must place names of cities and rivers on a map so that they don't overlap. The names should also not cover other features of the map, and it should be clear to which town, or river, each name applies. This is in practice an overconstrained problem - there are no feasible solutions. Constraint programming was used to detect which cities had to be removed to get a feasible solution. These cities could then be handled by a specialised program - or the end-user. In this case the constraint propagation was specifically designed to detect the inconsistency: including "an edge-consistency algorithm that would propagate all information relevant to incompatibilities" [27].

A feature that comes for free with CLP is the facility to handle *dynamic Constraint Satisfaction Problems* (dynamic CSP's), were the set of problem variables may grow as search progresses. A typical application of dynamic CSP's is configuration [92]. In this case the choice of a particular type of component, will introduce new variables for each (unknown) property of the component. Later choices will then instantiate the variables. For dynamic CSP's special constraints called "activity" constraints were introduced, whose role was to turn variables on or off. However these constraints can be handled in CLP as guarded clauses (introduced in section 2.2.1 above) which, if their guard is satisfied, can declare new variables and constrain then appropriately.

Another specialised behaviour was used in a train routing and timetabling application by Gosselin [57]. In this application a complete route is composed of a sequence of sections, each of which has several choices. One approach would be to compute all possible routes in advance - this corresponds to the modelling technique of column generation for mixed integer programming. However this causes an (exponential) explosion in the input. Gosselin's approach is to represent possible routes intentionally, but allow "disjunctive" reasoning about the set of choices for a section of the route. Thus if a train takes time k to pass through a section, and if all possible sections following a given section are blocked at time t, then propagation will remove the time $t - k$ from the set of available times for the current section of the route.

At an early stage in the history of constraint programming, it seemed that each new application required new specialised constraints, but in fact some specialised constraints have proved to be useful in a broad range of applications. One area of intense study has been resource constraints. Essentially a resource constraint states that at any time, the number of resources being used by tasks running at that time is no more than the available number of resources. This constraint is a "global" constraint in that it applies to an arbitrary number of tasks and resources - which could include all the tasks and resources in the whole problem.

Such a specialised constraint is available in CHIP called the *cumulative* constraint [1]. It has been used in timetabling [7], in scheduling network repair

operations [29], and in a variety of commercial CHIP applications [109]. A similar generic constraint is the ILOG SCHEDULE library [81], which has been applied on a range of scheduling applications [98, 42].

The algorithms underlying these constraints are based on ones developed in the Operations Research community. A comparative study of some of these algorithms is in [2]. Such constraints have made constraint programming competitive with the best specialised algorithms available on these applications, with the added advantage that the constraint programming can easily be extended to handle side-constraints equally efficiently [16, 126].

To support the implementation of such specialised constraint behaviours some generic techniques have been developed. Such a language - Constraint Handling Rules - has been used for supporting terminological reasoning [52].

Indeed the Concurrent Constraints framework [107] supports just this requirement for cleanly implementing well-behaved constraints! A concurrent constraints language which has been used for tackling combinatorial problems is cc(FD) [124]. We will encounter cc(FD) again in the next section.

4.4 Redundant Constraints

Local consistency is cheap to enforce, but weak. Often it is possible to achieve more powerful propagation by adding redundant constraints. For example the constraints that $B1 + B2 = 1$, $S1 + B1 = T1$, $S2 + B2 = T2$ and $S1 + S2 = 5$ entail that $T1 + T2 = 6$, but this consequence is not yielded by reasoning on the domains of the variables. For this reason in many applications (eg. [121, 59]) it proves useful to introduce extra variables for aggregates: (eg $T = T1 + T2$ and $S = S1 + S2$), and to add redundant constraints on the aggregate variables (eg $T = S + 1$).

For resource constraints, in which several tasks share a resource, it is very powerful to add redundant constraints on the sum of the durations of the tasks [127, 17]. The specialised constraints - *cumulative* and *SCHEDULE* - introduced above are low-level implementations of consistency constraints on sets of tasks and resources.

It has been pointed out in [17] and [122] that such redundant constraints can be used to achieve the same effect as facet-inducing cuts in mixed integer programming. On the other hand there are simple applications where local propagation has a behaviour which is extremely hard to achieve in the mixed integer programming framework [114, 82].

For optimisation problems, where branch and bound is being used as the optimisation algorithm, it is particularly important to achieve good propagation on constraints involving the cost. The lower bound of the variable representing the cost is a safe estimate of the cost of the full solution. Achieving an accurate lower bound is crucial to the behaviour of branch and bound - and other constructive search techniques! Tailoring the behaviour of the constraints, and

adding redundant constraints are the two main techniques available in constraint programming to achieve this aim.

4.5 Constraint Relaxation

It is claimed in [89] that "real problems are usually overconstrained". This claim is born out by studying reported practical applications of constraints. A set of tests which happen to illustrate this phenomenon is described in [7]. The application is to build a timetables for a set of around 300 examinations, taking place over a limited time period in a limited set of examination rooms. On 100 tests the system produced answers within 20 seconds in 55 cases; in the other 45 no answer was produced within 5 minutes. It was reported that

> "for these [45] tests it would suffice to relax two or three constraints in order to efficiently solve the problem".

The challenge is how to detect which constraints to relax.

One approach, probably the most realistic for many applications, is to pass the buck to the end-user, since she knows best which constraints are most important at any particular moment in time. The developers of the Renault application of section 3.1 above take this approach.

Another approach is to associate a cost with each constraint violation (known in mathematical programming as "Lagrangian relaxation") and to minimise this cost. The main drawback of this approach is that constraint violations have to be valued on the same scale as the optimisation criteria. Not only does this require the application designer to make some rather arbitrary judgements, but also it becomes hard to maintain in the light of ever-changing priorities. Lagrangian relaxation is adopted in many practical applications for example [133, 65].

A more natural approach is to compare the constraints with each other, forming some kind of constraint hierarchy. (Nevertheless there remains the decision whether to relax one more constraint in order to get an improved optimum solution.) Hierarchical constraints can be solved optimistically, by first searching for a solution satisfying all the constraints and then relaxing more and more constraints - the weakest first - until a solution is found [11]. The optimistic approach has been applied in some timetabling applications [89, 41]. As yet this approach does not appear to have scaled up enough to be viable for real applications.

Alternatively they can be solved pessimistically be first searching for a solution satisfying the strongest constraints and then seeking to satisfy more and more constraints, strongest first [51]. The pessimistic approach was applied to a diagnosis problem [101]. The main idea is to find the assignment that minimises the number - or value - of constraints violated, and this is the diagnosis.

Constraint violations are an important focus in practical scheduling systems. In ISIS [46], the decision as to which constraints to relax, and how, are an

important aspect of the search technique. In OPIS [115], the whole focus of scheduling is on repair, and the repair is driven by constraint violations.

Currently the way overconstrained problems are handled in most other applications is problem specific. Typical descriptions of applications report that "simple constraint relaxations are proposed to the user" [3] or that

> "There are many forms of interactions between constraints and conflicts between constraint types that cannot be resolved automatically. The system is used as a decision support system, which helps the human scheduler, but does not replace him" [109].

4.6 Advanced Search

In several of the reported applications using backtrack search, the program returns answers quickly, when there are any, but continues to compute for a very long time if there are no (more) answers. The many studies on the *phase transition* in NP-complete problems [19], have strongly suggested that each problem has a "really hard" area, where most (or all) algorithms will have difficulty proving success or failure. In fact all the reported applications of constraint programming, except circuit verification, have an optimisation component. It seems likely that the optimisation drives most problems into the hard region sooner or later! Consequently the so-called "proof-of-optimality" is a major difficulty for any complete search technique.

For many users of constraint programming, it comes as a disappointment that the backtrack search behaviour can vary dramatically with small changes in the data of the problem. For example in the tests described in the previous section, the tightening of the data appears to take 45 instances of the problem into the really hard region.

Simonis points out that backtracking is most important for developing complex assignment strategies in the problem solver [110]. Backtracking is used, for example, in Micro-Boss [102], but a great deal of computational effort is put into each choice, to avoid reaching *deadend* states where backtracking is necessary.

However for many practical applications it is impossible to find a solution which is guaranteed to be the best: such a complete coverage of the set of possibilities is ruled out by the sheer size and complexity of the problem. One way to avoid an explosion in the search space is to explore a strictly limited number of alternative states, for example by using *beam* search. This is the approach taken in the ISIS system, originally applied for scheduling the manufacture of turbine blades [46].

However constructive search techniques, which work on partial solutions, can have little information about good or bad combinations of values for their unlabelled variables. Repair-based search techniques, by contrast, have a "current" complete, though often infeasible, solution as a starting point. Weaknesses in the current solution are repaired, either by a problem-specific *move* opera-

tion, as in hill-climbing and simulated annealing, or by labelling, as in conflict minimisation [91] or weak-commitment [131].

A move operation takes a complete labelling of the variables and modifies it, yielding another complete labelling. The move is designed so that if the original labelling satisfied certain constraints, then so will the resulting labelling. A typical example is the 2-swap, a move which maps one tour to another for solving the travelling salesman problem. Repair based on move operators is used in scheduling systems [115], and in applications that use simulated annealing [28, 21, 26]. Integrating constraint propagation with repair-based search using move operations is an ongoing research theme, but there are applications where they are loosely connected. In OPIS [115] the problem constraints are crucial for determining the scope of the conflict which has to be repaired and the most appropriate repair operator for the job. In the map-labelling application [28] constraint propagation is used as a preprocessing step, to detect which parts of the problem cannot be solved by the annealer.

Repair based on labelling is closely related to constructive labelling. In this case the previous labelling is used as a heuristic when selecting values for the variables in the new labelling. Secondly the previous labelling is used in the selection of which variable to label next - typically one whose previous value conflicts with one or more constraints. In conflict minimisation the repair labels only one variable [91]. In weak commitment relabelling continues until there is no way to change a variable so as to make it consistent with the other relabelled variables [131]. Repair based on label operators has proven very effective on a range of problems. These techniques are now being proven on practical applications. Conflict minimisation has been incorporated into the scheduler used for the Hubble telescope [71], and weak commitment is being applied on a network path assignment application [132].

5 Conclusion

This paper has skimmed the surface of a range of practical applications of constraint programming. It is already impossible to cover the full range - a survey of timetabling applications alone would require a full paper! Moreover the more commercial an application is, the less information is available, and consequently some commercial constraint-based systems have not been presented here.

The currently exploited applications are mostly in the areas of scheduling, timetabling and resource allocation. In these areas constraint programming has proved its worth in the flexibility it offers for coping with specific constraints and its support for mixed initiative problem solving: the end users is typically supported by the system rather than being replaced by a black-box solution.

Nevertheless this paper has also focussed on areas such as industrial control software where the potential of constraints programming is still to be realised. This represents a quite different role for constraint programming: to automate

the generation of correctly functioning embedded software from high-level specifications.

Instead of breaking down the survey by application areas, it proved more coherent to break them down by technologies - modelling, propagation, backtrack search, specialised constraint behaviours, redundant constraints, constraint relaxation and advanced search. However the main role of constraint programming is not to support particular technologies, programming styles or techniques, but to accommodate the diverse requirements of the application at hand. At the risk of trivialising, constraint programming provides the glue that holds together a variety of diverse components. For example constraint propagation is a form of data-driven computation, embedded in a traditional procedure invocation programming environment. Operational research and mathematical programming algorithms are further examples of components integrated within the single constraint programming environment. No less important are the graphical facilities which support mixed initiative programming.

Some surprising successes have been achieved by the simple combination of constraint propagation and search: for example constraint propagation techniques have recently enabled interval reasoning to achieve some spectacular results [123]. The exciting prospect about such results is that they are in no sense cut off, nor do they exclude the other features of constraints programming. They can be taken just as they are, included in any constraint programming system, and integrated with whatever other facilities may be required in solving a given practical problem.

Many of the developments in constraint programming are as much pulled by the requirements of applications as pushed by technological breakthroughs. One current frontier is the embedding of repair-based search into constraint programming. This represents a clear response to application needs, but is proving a fruitful area of research. Another frontier is the use of constraint programming to bring software engineering techniques to the mathematical programming community: constraints programming may play an important role in the mathematical modelling of complex problems.

With the integration of constraint programming and databases [77], we see the paradigm being brought right to the heart of commercial information processing. Constraint programs will be used for database queries so that database access will become just one feature of a complete decision support functionality. At that point constraint programming will have finally come of age.

References

[1] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Journal of Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[2] P. Baptiste and C. Le Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In C. Mellish, editor, *Proc. IJCAI*, volume 1, pages 600–606, 1995.

[3] J. Bellone, A. Chamard, and A. Fischler. Constraint logic programming decision support systems for planning and scheduling aircraft manufacturing at dassault aviation. In Roth [100].

[4] Frédéric Benhamou, David McAllester, and Pascal Van Hentenryck. Clp(intervals) revisited. In *Proc. ILPS-94*, pages 124–138, Ithaca, November 1994.

[5] R. Berger. Constraint-based gate allocation for airports. In ILOG [69]. ILOG SA, 12, Avenue Raspail, BP 7, 94251 Gentilly Cedex, France.

[6] R. Bisdorff and S. Laurent. Industrial disposing problem solved in CHIP. In *ICLP'93: Proceedings 10th International Conference on Logic Programming*, pages 830–831, Budapest, 1993. (Abstract; not presented).

[7] P. Boizumault, Y. Delon, and L. Péridy. Planning exams using constraint logic programming. In Leon Stirling, editor, *Proc. 2nd International Conference on the Practical Applications of Prolog*, April 1994.

[8] J. Bormann, J. Lohse, M. Payer, and R. Scmid. Circuit verification environment cve2. Technical Report Version 1.72, ZFE BT SE 12 Siemens AG, 1995. (User Manual and System Description).

[9] A. Borning and B.N. Freeman-Benson. The OTI constraint solver: A constraint library for constructing graphical user interfaces. In *Proc Principles and Practice of Constraint programming CP'95*. Springer Verlag, 1995.

[10] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[11] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5:223–270, 1992.

[12] A. Brodsky and Y. Kornatsky. The LyriC language: Querying constraint objects. In *Proc. SIGMOD*, pages 35–46, San Jose, 1995.

[13] R.E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[14] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, June 1992.

[15] W. Büttner and H. Simonis. Embedding Boolean expressions into logic programming. *Journal of Symbolic Computation*, 4:191–205, October 1987.

[16] Y. Caseau. Constraint programming and operations research: Mixed solutions for mixed problems. In *Proc. Principles and Practice of Constraint Programming*, 1993.

[17] Yves Caseau and Francois Laburthe. Improved clp scheduling with task intervals. In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming*, Santa Margherita, 1994. MIT Press.

[18] A. Chamard and C. Pradelles. CHIP applications at dassault aviation. COSYTEC SA, Parc Club Orsay Universite, 4, rue Jean Rostand, 91893 Orsay Cedex, France, November 1994.

[19] P. Cheeseman, R. Kanefsky, and W.M. Taylor. Where the *really* hard problems are. In J. Mylopoulos and R. Reiter, editors, *Proc. IJCAI*, pages 331–337. Morgan Kaufmann, 1991.

[20] T. Chew and David J.-M. A constraint-based spreadsheet for cooperative production planning. In *AAAI Sigman Workshop in Knowledge-Based Production Planning, Scheduling and Control*, 1992.

[21] T.L. Chew, J.-M. David, A. Nguyen, and Y. Tourbier. Solving constraint satisfaction problems with simulated annealing; the car sequencing problem revisited. In *Proc. 12th International Conference on AI, Expert Systems and Natural Language*, 1992.

[22] P. Codognet, F. Fages, and T. Sola. A metalevel compiler of clp(fd) and its combination with intelligent backtracking. Technical Report LACS-93, Thompson-CSF, 1993.

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

[24] COSYTEC. CHIP C library. COSYTEC SA, Parc Club Orsay Universite, 4, rue Jean Rostand, 91893 Orsay Cedex, France.

[25] COSYTEC. Proceedings of the CHIP users club. COSYTEC SA, Parc Club Orsay Universite, 4, rue Jean Rostand, 91893 Orsay Cedex, France, November 1994.

[26] B Crabtree. Resource scheduling - comparing simulated annealing with constraint programming. *BT Technology Journal*, 13(1), 1995.

[27] J.-Y. Cras. Using constraints in servicing: A few short tales. Presented at the 2nd CHIC-PRINCE Workshop, ECRC, October 1993.

[28] J.-Y. Cras. Using constraint logic programming: A few short tales. In M. Bruynooghe, editor, *Proc. International Symposium on Logic Programming*, 1994.

[29] T. Creemers, L.R. Giralt, J. Riera, C. Ferrarons, J. Roca, and X. Corbella. Constraint-based maintenance scheduling on an electric power distribution network. In Roth [100], pages 135–144.

[30] I. Cruz. *Expressing Constraints for Data Display Specification: A Visual Approach*. In Saraswat and Van Hentenryck [105], 1994.

[31] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[32] Dincbas, M., 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, Japan, December 1988.

[33] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving a cutting-stock problem in constraint logic programming. In *Fifth International Conference on Logic Programming (ICLP'88)*, Seattle, USA, August 1988. M.I.T Press.

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

[35] M. Dincbas, H. Simonis, and 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.

[36] S. Donikian and G. Hegron. *Constraint Management in a Declarative Design Method for 3D Scene Sketch Modelling*. In Saraswat and Van Hentenryck [105], 1994.

[37] A. Dresse. A constraint programming library dedicated to timetabling. In ILOG [69]. ILOG SA, 12, Avenue Raspail, BP 7, 94251 Gentilly Cedex, France.

[38] ECRC. ECLiPSe 3.5. Technical report, ECRC, 1995. http://www.ecrc.de/eclipse/eclipse.html.

[39] EDIF. EDIF electronic design interchange format. Technical Report Version 2, Electronic Industries Association, 1989. Reference Manual.

[40] O. Evans. How to use the spreadsheet manager. Technical report, ICL, Lovelace Road, Bracknell, England, 1993. CHIC, Esprit project 5291, Report T3.5/W2.2.

[41] F. Fages, J. Fowler, and T. Sola. Handling preferences in constraint logic programming with relational optimisation. In M. Hermenegildo and J. Penjaam, editors, *Proc. 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 261–276, Madrid, 1994.

[42] J. Feldman, A. Hoyos, N. Sekas, and D. Vergamini. Scheduling engine for the long island lighting company. In ILOG [69]. ILOG SA, 12, Avenue Raspail, BP 7, 94251 Gentilly Cedex, France.

[43] J. Fernandez and E. Sanchez. A cashier assignment system. In ILOG [69]. ILOG SA, 12, Avenue Raspail, BP 7, 94251 Gentilly Cedex, France.

[44] R. E. Fikes. REF–ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[45] T. Filkhorn, H.-A. Schneider, A. Scholz, A. Strasser, and P. Warkentin. SVE system verification environment. Technical Report SVE, ZFE BT SE Siemens AG, 1995.

[46] M. Fox. *ISIS:A Retrospective*, pages 3–28. In Zweben and Fox [134], 1994.

[47] H. Frangouli, V. Harmandas, and P. Stamatopoulos. UTSE: Construction of optimum timetables for university courses - a CLP-based approach. In Roth [100], pages 225–243.

[48] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[49] E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, November 1978.

[50] Eugene Freuder. Exploiting structure in constraint satisfaction problems. In B. Mayoh, E. Tyugu, and J.Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, NATO Advanced Science Institute Series, pages 54–79. Springer-Verlag, 1994.

[51] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[52] Thom Frühwirth and Philipp Hanschke. *Terminological Reasoning with Constraint Handling Rules*, pages 361–384. In Saraswat and Van Hentenryck [105], 1994.

[53] P. Gärdenfors. *Knowledge in Flux*. MIT Press, 1988.

[54] M. Gleicher. *Practical Issues in Graphical Constraints*. In Saraswat and Van Hentenryck [105], 1994.

[55] Y. Gloner. A scheduling engine. COSYTEC SA, Parc Club Orsay Universite, 4, rue Jean Rostand, 91893 Orsay Cedex, France, November 1994.

[56] S.W. Golomb and L.D. Baumert. Backtrack programming. *Journal of the ACM*, 12:516–524, 1965.

[57] V. Gosselin. Train scheduling using constraint programming techniques. In *Proc. 13th International Conference on Artificial Intelligence, Expert Systems and Natural Language*, pages 401–413, Avignon, 1993. EC2, 269-287, rue de la Garenne, 92024 Nanterre, France.

[58] T. Graf, P. van Hentenryck, C. Pradelles, and L. Zimmer. Simulation of hybrid circuits in constraint logic programming. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 72–77, Detroit, 1989.

[59] El-Sakkout H. and Wallace M.G. Modelling a fleet assignment application in constraint logic programming. Technical report, 1995.

[60] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, October 1980.

[61] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Tansactions on Systems, Science and Cybernetics*, 4(2), 1968.

[62] R. Helm, T. Huynh, C. Lassez, and K. Marriott. A linear constraints technology for user interfaces. In *Graphics Interface*, pages 301–309, Vancouver, 1992.

[63] M. Henz, G. Smolka, and J. Wuertz. Object-oriented concurrent constraint programming in oz. In *Priniciples and Practice of Constraint Programming*, pages 27–48. MIT Press, 1995.

[64] L. Hermosilla and G. Kuper. Towards the definition of a spatial object-oriented data model with constraints. In Kuper and Wallace [77].

[65] Hon Wai Chun. Solving check-in counter constraints with ILOG solver. In ILOG [69]. ILOG SA, 12, Avenue Raspail, BP 7, 94251 Gentilly Cedex, France.

[66] J.N. Hooker and H. Yan. *Verifying Logic Circuits by Benders Decomposition.* In Saraswat and Van Hentenryck [105], 1994.

[67] E Hyvőnen. Interval constraint spreadsheets for financial planning. In *Proc. First International Conference on Artificial Intelligence Applications on Wall Street.* IEEE Computer Society Press, 1991.

[68] ILOG. ILOG SOLVER: Object-oriented constraint programming. ILOG SA, 12, Avenue Raspail, BP 7, 94251 Gentilly Cedex, France.

[69] ILOG, editor. *Proceedings of the ILOG User-group meeting.* ILOG, 1995. ILOG SA, 12, Avenue Raspail, BP 7, 94251 Gentilly Cedex, France.

[70] Joxan Jaffar, Spiro Michayov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[71] M. Johnston and S. Minton. *Analyzing a Heuristic Strategy for Constraint-Satisfaction and Scheduling*, chapter 9, pages 257–289. In Zweben and Fox [134], 1994.

[72] P. Kay and H. Simonis. Building industrial CHIP applications from reuseable software components. In Roth [100], pages 355–369.

[73] Kelly, Macdonald, Marriott, Sondergaard, Stuckey, and Yap. An optimizing compiler for CLP(R). In U. Montanari and F. Rossi, editors, *Proc. Principles and Practise of Constraints Programming, CP'95*, pages 222–239. Springer Verlag, 1995.

[74] A.D. Kelly, A. Macdonald, K. Marriott, H. Sondergaard, P. Stuckey, and R. Yap. An optimizing compiler for CLP(R). In Montanari and Rossi [95].

[75] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. In C. Mellish, editor, *IJCAI*, pages 541–547, Montreal, Canada, 1995.

[76] R.E. Korf. Optimal path-finding algorithms. In *Search in Artificial Intelligence*, pages 223–267. Springer, 1988.

[77] G. Kuper and M. Wallace, editors. *Proc. 1st International CONTESSA Workshop on Constraint Databases and their Applications.* Springer, 1995.

[78] J-L. Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[79] E.L. Lawler and D.E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14:699–719, 1966.

[80] J. Lazaro and P. Aristondo. Using SOLVER for nurse scheduling. In ILOG [69]. ILOG SA, 12, Avenue Raspail, BP 7, 94251 Gentilly Cedex, France.

[81] Claude Le Pape. Implementation of resource constraints in ILOG SCHEDULE. *Intelligent Systems Engineering*, 3(2), 1994.

[82] J. Little and K. Darby-Dowman. The significance of constraint logic programming to operational research. Presented at APMOD'95, 1995.

[83] M. Loos. Optimized scheduling for the process industry. COSYTEC SA, Parc Club Orsay Universite, 4, rue Jean Rostand, 91893 Orsay Cedex, France, November 1994.

[84] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[85] Michael J. Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *ICLP'87: Proceedings 4th International Conference on Logic Programming*, pages 858–876, Melbourne, 1987. MIT Press.

[86] J. Maloney, A. Borning, and B. Freeman-Benson. Constraint technology for user interface construction in thinglab ii. In *Proc. ACM OOPSLA*, pages 381–388, 1989.

[87] F. Maruyama, Y. Minoda, Sawada S., and Y. Takizawa. Constraint satisfaction and optimisation using nogood justifications. In *Proc. 2nd Pacific Rim Conf. on AI*, 1992.

[88] M. Meier. Debugging constraint programs. In *Proc. Principles and Practice of Constraint Programming CP'95*. Springer, 1995.

[89] F. Menenez and P. Barahona. *An Incremental Hierarchical Constraint Solver*, pages 291–318. In Saraswat and Van Hentenryck [105], 1994.

[90] G. Mezzanatto, M. Foglino, P. Giordanengo, M. Apra, and G. Gullane. Using ai techniques to design and install electrical bundles. In *Proc. 13th International Conf. on Artificial Intelligence, Expert Systems and Natural Language*, Avignon, 1993. EC2, 269-287, rue de la Garenne, 92024 Nanterre, France.

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

[92] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. In *Proc. 11th IJCAI*, pages 1395–1401, 1989.

[93] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[94] U. Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

[95] U. Montanari and F. Rossi, editors. *Proc Principles and Practice of Constraint Programming CP'95*. Springer, 1995.

[96] B. Nadel. *Tree Search and Arc Consistency in Constraint Satisfaction Algorithms*, chapter 9. Springer, 1988.

[97] A. Newell and H.A. Simon. GPS, a program that simulates human thought. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 279–293. New York: McGraw-Hill, 1963.

[98] J.-F. Puget. Applications of constraint programming. In Montanari and Rossi [95], pages 647–650.

[99] A. Rauzy. Notes on the design of an open boolean solver. In P. Van Hentenryck, editor, *Proc. International Conference on Logic Programming ICLP'94'*. MIT Press, 1994.

[100] Al Roth, editor. *Proc. Third International Conference on the Practical Applications of Prolog*, Paris, 1995.

[101] D. Sabin, M. Sabin, R. Russell, and E. Freuder. A constraint-based approach to diagnosing software problems in computer networks. In Montanari and Rossi [95], pages 463–480.

[102] N. Sadeh. *Micro-Opportunistic Scheduling*, chapter 4. In Zweben and Fox [134], 1994.

[103] Michael Sannella. *The SkyBlue Constraint Solver and Its Applications.* MIT Press, 1994.

[104] V. Saraswat. The category of constraint systems is cartesian-closed. In *Proc. Logic in Computer Science LICS'92.* IEEE, 1992.

[105] V. J. Saraswat and P. Van Hentenryck, editors. *Principles and Practice of Constraint Programming.* MIT Press, 1994.

[106] Vijay Saraswat, Radha Jagadeesan, and Vinheet Gupta. Programming in timed concurrent constraint languages. In B. Mayoh, E. Tyugu, and J.Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, NATO Advanced Science Institute Series, pages 361–410. Springer-Verlag, 1994.

[107] Vijay A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[108] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *POPL 91*, 1990.

[109] H. Simonis. Application development with the CHIP system. In Kuper and Wallace [77].

[110] H. Simonis. Application development with the chip system. In *Proc. 1st International CONTESSA Workshop on Constraint Databases and their Applications.* Springer, 1995.

[111] H. Simonis. The CHIP system and its applications. In U. Montanari and F. Rossi, editors, *Proc. Principles and Practice of Constraints Programming, CP'95*, pages 643–646. Springer Verlag, 1995.

[112] H. Simonis and M. Dincbas. Using logic programming for fault diagnosis in digital circuits. In *German Workshop on Artificial Intelligence (GWAI-87)*, pages 139–148, Geseke, W.Germany, September 1987.

[113] H. Simonis, H. N. Nguyen, and M. Dincbas. Verification of digital circuits using chip. In G.J. Milne, editor, *Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, July 1988. IFIP, North-Holland.

[114] B. Smith, S. Brailsford, P.M. Hubbard, and H.P. Williams. The progressive party problem: Integer linear programming and constraint propagation compared. In *Proc. APMOD*, Brunel, London, April 1995.

[115] S. Smith. *OPIS: A Methodology and Architecture for Reactive Scheduling*, pages 29–66. In Zweben and Fox [134], 1994.

[116] G. J. Sussman and G. L. Steele. CONSTRAINTS — a language for expressing almost–hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, 1980.

[117] G.J. Sussman and G.L. Steele. Constraints-a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, 1980.

[118] Ivan Sutherland. *A Man Machine Graphical Communication System.* PhD thesis, Massachusetts Institute of Technology, January 1963.

[119] Touraivane. Constraint programming and industrial applications. In U. Montanari and F. Rossi, editors, *Proc. Principles and Practise of Constraints Programming, CP'95*, pages 640–642. Springer Verlag, 1995.

[120] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[121] P. Van Hentenryck. *Scheduling and Packing in the Constraint Language cc(FD)*, chapter 5. In Zweben and Fox [134], 1994.

[122] P. Van Hentenryck. Constraint solving for combinatorial search problems: A tutorial. In Montanari and Rossi [95], pages 564–587.

[123] P. Van Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 1995. to appear.

[124] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(FD). In A. Podelski, editor, *Constraint Programming: Basics and Trends*, pages 293–316. Springer, 1994.

[125] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, Cambridge, MA, 1989.

[126] M. Wallace. Constraints as a tool for distributed scheduling. In H. Richards, editor, *Proc. International Conference on Improving Manufacturing Performance in the Distributed Enterprise: Advanced Systems and Tools*, Edinburgh, 1995.

[127] Mark Wallace. Applying constraints for scheduling. In B. Mayoh, E. Tyugu, and J.Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, NATO Advanced Science Institute Series, pages 161–180. Springer-Verlag, 1994.

[128] M.G. Wallace. Compiling integrity checking into update procedures. In *Proc. IJCAI*, Sydney, 1991.

[129] M.G. Wallace, editor. *Proc. Conf. on Practical Applications of Constraints Technology*, Paris, 1995.

[130] D. L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*. McGraw Hill, 1975.

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

[132] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In Montanari and Rossi [95], pages 88–102.

[133] P.-A. Yvars and H. Roussou. SARTRE: A computer-aided design tool for robotised production lines. In ILOG [69]. ILOG SA, 12, Avenue Raspail, BP 7, 94251 Gentilly Cedex, France.

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