technical report ECRC–93–5

Constraint Logic Programming
-
An Informal Introduction

Thom Frühwirth, Alexander Herold, Volker Küchenhoff,
Thierry Le Provost, Pierre Lim, Eric Monfroy, Mark Wallace

EUROPEAN COMPUTER-INDUSTRY RESEARCH CENTRE

ECRC GMBH, ARABELLASTR. 17 D-8000 MÜNCHEN 81, GERMANY - TEL +49 89/926 99 0 - FAX 926 99 170 - TLX 521 6910

About authors:
The author can be reached via e-mail: {thom, herold, volker, thierry, pierre, eric, mark}@ecrc.de

Abstract

Constraint Logic Programming (CLP) is a new class of programming languages combining the declarativity of logic programming with the efficiency of constraint solving. New application areas, amongst them many different classes of combinatorial search problems such as scheduling, planning or resource allocation can now be solved, which were intractable for logic programming so far. The most important advantage that these languages offer is the short development time while exhibiting an efficiency comparable to imperative languages. This tutorial aims at presenting the principles and concepts underlying these languages and explaining them by examples. The objective of this paper is not to give a technical survey of the current state of art in research on CLP, but rather to give a tutorial introduction and to convey the basic philosophy that is behind the different ideas in CLP. It will discuss the currently most successful computation domains and provide an overview on the different consistency techniques used in CLP and its implementations. [1] [2]

[1]This work is partially funded by the ESPRIT project CHIC,Nr.5291.
[2]This report will appear in [FHK+92].

1 Introduction

During the last decade a new programming paradigm called *"logic programming"* has emerged. The best known representative of this new class of programming languages is *Prolog*, originated from ideas of Colmerauer in Marseille and Kowalski in Edinburgh. Programming in Prolog differs from conventional programming both stylistically and computationally, as it uses logic to declaratively state problems and deduction to solve them.

It has been argued in the literature [Kow79, Ste80] that a program is best divided into two components called *competence* and *performance* or *logic* and *control*. The competence component describes factual information - statements of relationships - which must be manipulated and combined to compute the desired result. The performance component deals with the strategy and control of the manipulations and combinations. The competence part is responsible for the correctness of the program; the performance part is responsible for the efficiency. An ideal programming methodology would first be concerned with the competence (*"what"*), and only then, if at all, worry about the performance (*"how"*). Logic programming provides a means for separation of these concerns. It is based on *first order predicate logic*, and the performance component is mostly automatic by relying on a built-in computation mechanism called *SLD-resolution*.

In this way, logic programming has the unique property that its *semantics*, operational and declarative, are both simple and elegant and coincide in a natural way. These semantics, however, have their limitations. Firstly the objects manipulated by a logic program are uninterpreted structures - the set of all possible terms that can be formed from the functions and constants in a given program. Equality only holds between those objects which are syntactically identical. Every semantic object has to be *explicitly* coded into a term; this enforces reasoning at a primitive level. *Constraints* on the other hand are used to *implicitly* describe the relationship between such semantic objects. These objects are often ranging over such rich computation domains, as integers, rationals or reals.

The second problem related to logic programming stems from its uniform but simple computation rule, a depth-first search procedure, resulting in a *generate and test* procedure with its well-known performance problems for large search applications. Constraint manipulation and propagation have been studied in the Artificial Intelligence community in the late 1970s and early 1980s [Mon74, Ste80, Mac86] to make search procedures more intelligent. Techniques like local value propagation, data driven computation, forward checking (to prune the search space) and look ahead have been developed for solving constraints. These techniques can be summarised under the heading *"Consistency Techniques"*.

Constraint Logic Programming (CLP) is an attempt to overcome the difficulties of logic programming by enhancing a Prolog-like language with constraint solving mechanisms. Curiously both of these limitations of logic programming can be lifted using "constraints". However, each limitation is treated by a quite different notion of constraint. CLP has hence two complementary lines of descent.

Firstly it descended from work that aimed at introducing richer data structures to a logic programming system thus allowing semantic objects, e.g. arithmetic expressions, directly to be expressed and manipulated. The core idea here is to replace the computational heart of a logic programming system, unification, by constraint handling in a constraint domain. This scheme, called CLP(X), has been laid out in the seminal paper of Jaffar & Lassez [JL87]. X has been instantiated with several so called computation domains, e.g. reals in CLP($\mathcal{R}$), rationals in CLP($\mathcal{Q}$), and integers in CLP($\mathcal{Z}$).

Secondly CLP has been strongly influenced by the work on consistency techniques. With the objective of improving the search behaviour of a logic programming system Gallaire [Gal85] advocated the use of these techniques in logic programming. He proposed the active use of constraints, pruning the search tree in an a priori way rather than using constraints as passive tests leading to a "generate and test" or "standard backtracking" behaviour. Subsequently the different inference mechanisms underlying the finite domain part of the CLP system CHIP [DVS+88] were developed. The key aspect is the tight integration between a deterministic process, constraint evaluation, and a nondeterministic process, search. It is this active view of constraints which is exploited in CHIP to overcome the well-known performance problems of "generate and test". This new paradigm exhibits a data-driven computation and can be characterised as *"constrain and generate"*.

Constraint solving has been used in many different application areas such as engineering, planning or graphics. Problems like scheduling, allocation, layout, fault diagnosis and hardware design are typical examples of constrained search problems. The most common approach for solving constrained search problems consists in writing a specialised program in a procedural language. This approach requires substantial effort for program development, and the resulting programs are hard to maintain, modify and extend. With CLP systems a large number of constrained search problems have been solved, some of them were previously solved with conventional languages. CLP languages dramatically reduce the development time while achieving a similar efficiency. The resulting programs are shorter and more declarative and hence easier to maintain, modify or extend. The wealth of applications shows the flexibility of CLP to adapt to different problem areas. Many Operations Research problems have been solved with the CLP system CHIP [DVS+88, Van88, DSV90]. Another very promising application domain is circuit design [Sim92, FSTW91]. Extensive work has also been devoted to financial applications [Ber89, LMY87]. More recently applications in user interfaces [HHLM91] and in databases [KKR90] have been studied. As the subsequent tutorial in this summer school focusses on industrial applications of CLP, we will not further discuss them in this article.

The aim of this informal tutorial is to present the most prominent ideas and concepts underlying CLP languages. It is not intended to present the underlying theory of this new class of programming languages or to give an overview on the current state of art in CLP research. There are already technical surveys in the literature, giving more details on those aspects. In particular the article of [Van91] is worth reading. A restricted view is presented in [Coh90, Frü90] discussing work around the CLP scheme. For the usage of "consistency techniques" in CLP, [Van89] is a valuable source going from theory to application with a large number of programming example.

This tutorial is organised as follows: In the next section we will introduce the CLP scheme and review the most important computation domains that have been developed so far, linear and non-linear arithmetic and boolean constraints. Then we will introduce the concept of finite domains, consistency techniques and their extension to arbitrary domains. Next we will explore ways of extending and tuning constraint systems. Then the work on search and optimisation in CLP will be presented. Finally current CLP implementations will be reviewed, amongst them the most well-known systems: CHIP [DVS+88], CLP($\mathcal{R}$) [JMSY90] and Prolog III [Col90].

2 The CLP Scheme

In this section we will introduce in an informal way the basics of the Constraint Logic Programming Scheme (called CLP(X)), as developed by Jaffar and Lassez [JL87]. The key aspect in the CLP scheme is to provide the user with more expressiveness and flexibility concerning the primitive objects the language can manipulate. Clearly the user wants to design his application using concepts that are as close as possible to his domain of discourse, e.g. he wants to use sets, boolean expressions, integers, rationals or reals, instead of coding everything as uninterpreted structures, i.e. finite trees, as is advocated in logic programming. Associated with each computation domain are the usual algebraic operations, including set intersection, conjunction of boolean expressions or multiplication of arithmetic expressions. These computation domains also have certain relations defined on them, such as set equality, equality between boolean expressions or equality, disequality and inequality between arithmetic expressions.

The constraint logic programming scheme admits computation directly over these domains. Special function and predicate symbols are introduced into logic programming, whose interpretation in the domain of computation is fixed. The relations over the domain of discourse are termed "constraints". Formulae involving the special function and predicate symbols are called "constraint formulae". Informally the word "constraint" is used also for constraint formulae.

When constraints are introduced into logic programming, a mechanism to solve them must also be introduced. In traditional logic programming the only constraint is equality between terms, and the unification algorithm is used to solve such constraints. There are two aspects related to unification. Firstly it tells us if the equation $t_1 = t_2$ has a solution. Secondly in case there exists a solution, it gives us a most general solution, which is logically equivalent to the original equation. The important aspect of unification is the first one deciding whether a constraint (or a set of constraints) has a solution or not. In other computation domains, where such a most general solution may not exist, the system can continue manipulating the original set of constraints. Therefore in order to accommodate constraints in logic programming the unification algorithm needs to be replaced by a decision procedure telling us whether a constraint or a set of constraints is satisfiable. In the following we will call such a decision procedure a *constraint solver*.

One reason for the success of CLP in recent applications has been the choice of constraint systems integrated into the different implementations. The selection of new constraint domains needs to satisfy both technical and practical criteria [DVS+88, JL87, SA89]. Most important are

- the expressive power of the computation domain,
- the existence of a complete and efficient constraint solver,
- its relevance in applications.

The constraint solver is complete if it is able to decide the satisfiability of any set of constraints of the computation domain. To achieve efficiency the constraint solver needs to be incremental, i.e. when adding a new constraint C to an already solved set of constraints S, the constraint solver should not start solving the new set $S \cup \{C\}$ from scratch.

In the following we will illustrate the operational behaviour of a CLP(X) system and the two most successful constraint domains, arithmetic and Boolean constraints. A description of other interesting domains may be found in section 6 where specific constraint languages are described.

2.1 The Arithmetic Domain

2.1.1 Linear Constraints

Providing arithmetic was one of the motivations behind the research in combining logic programming with constraints. Although Prolog has built-in facilities for evaluating arithmetic expressions the behaviour is not what one would ideally expect. Prolog cannot handle equations like `X - 3 = Y + 5`. In Prolog the term `X - 3` is not equal to the term `Y + 5` as Prolog knows only about uninterpreted structures.

The programmer needs to resort to the built-in arithmetic. And here the problems are the same as in any other programming language. Indeed the programmer needs to know which of the variables will be instantiated first and then he can use assignment (`is`) to instantiate the other. CLP($\mathcal{R}$) [JL87] was the first constraint programming language to introduce arithmetic constraints. There is a caveat. The decision procedure is only complete for linear arithmetic constraints. Nonlinear constraints are suspended until they become linear. Linear constraint handling turned out to be sufficient in many applications such as simulation of circuits and devices, decision-support systems and geometrical problems.

Linear arithmetic expressions are terms composed from numbers, variables and the usual arithmetic operators: negation ($-$), addition ($+$), subtraction ($-$), multiplication ($*$) and division ($/$). For the condition of *linearity* to be satisfied it is required that in a multiplication at most one of the components is a variable and that in a division the denominator is a number. An arithmetic constraint is an expression of the form $t_1 \ R \ t_2$ where R is one of the following predicates $\{>, \geq, =, \leq, <, \neq\}$.

There are several decision procedures for deciding a system of linear arithmetic constraints. Usually a combination of Gaussian elimination and a modified *Simplex* algorithm is employed. The Simplex algorithm is required as soon as inequality constraints need to be solved. The Simplex algorithm is used because it has quite a good behaviour on average, it is well-understood, and it can be made incremental.

We now present the execution mechanism for CLP languages informally through a small example. Consider the following problem from [Col90].

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

A CLP program (in an arithmetic domain) for solving this problem is given below.

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

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

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

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

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

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

A CLP program is syntactically a collection of *clauses* which are either *rules* or *facts*. Rules are as in Prolog, with the addition that they may contain constraints in their premises. Rules describe the conclusions that can be reached given certain premises. For our example we read "The meal consisting of foods `A`, `M` and `D` is a light meal if `A` is an appetiser (with a positive calorific value `I`), `M` is a main meal (with positive calorific value `J`), `D` is a dessert (with positive calorific value `K`) and `I + J + K` is less than or equal to 10". The premise of a rule is a conjunction of *constraints*, e.g. `I + J + K <= 10` and *atoms* e.g. `appetiser(A,I)`. Facts express known relationships. In our case, the calorific value of beef (which is a meat) is 5.

We shall describe the intermediate results of an execution of a CLP program as *computation states*. A computation state consists of two components, a *constraint store* and the remaining goals. We shall separate the constraint store from the remaining goals by the symbol ⋄. The constraint store consists of the set constraints collected during the computation so far. CLP programs are executed by reducing the goals in the computation state using the facts and rules. In each intermediate computation state the constraint store must be consistent. Consider the general query `?- lightmeal(A ,M ,D)` asking for all light meal plans. This corresponds to the initial computation state

```
⋄ lightmeal(A, M, D).
```

For our first reduction step we first have to choose an atomic goal to reduce. There is only one possibility i.e. `lightmeal(A, M, D)`. Next we need to choose an applicable rule. Again there is only one possibility i.e. the rule with the consequent `lightmeal(A, M, D)`. The next step is to form equations between variables in the consequent of the rule and the selected atom. The constraint store of the new computation state consists of the current constraint store, this equation set and the set of constraints in the premise of the rule. The atom set of the new computation state is the current atom set where the selected goal is replaced by the atoms of the premise of the rule (as in the case of Prolog). Thus our first reduction step produces the following computation state:

```
I + J + K <= 10, I > 0, J > 0, K > 0 ⋄
appetiser(A,I), main(M,J), dessert(D,K).[1]
```

A CLP system *searches* for all solution by systematically trying all possible rules (and facts) for the reduction of all the atoms in the atom set. Therefore any one possible alternative is in fact a sequence of reduction steps called a *derivation*. A derivation terminates when there are no more atoms to be reduced and the final constraint store is consistent. For the first example a *successful* derivation is the following:

```
A=radishes, I=1, 1+J+K <= 10, 1>0, J>0, K>0
⋄ main(M, J), dessert(D, K)

A=radishes, I=1, M=M1, J=I1, 1+J+K <= 10, 1>0, J>0,K>0
⋄ meat(M1, I1), dessert(D, K)

A=radishes, I=1, M=beef, J=5, M1=beef, I1=5, 1+5+K <= 10, 1>0, 5>0, K>0 ⋄ dessert(D, K)

A=radishes, I=1, M=beef, J=5, M1=beef, I1=5, D=fruit, K=2, 1+5+2 <= 10, 1>0, 5>0, 2>0 ⋄.
```

Note that the answer to this query is given by the constraint store. A simplified answer in terms of the input variables is `A=radishes, M=beef, D=fruit.`

If the constraint store becomes inconsistent, the derivation *fails*. An example of a failed derivation is now presented. We begin with the same initial computation state as above but make some different choices in the rules and facts to apply.

```
A=pasta, I=6, 6+J+K <= 10, 6>0, J>0, K>0
⋄ main(M,J), dessert(D,K)

A=pasta, I=6, M=M1, J=I1, 6+J+K <= 10, 6>0, J>0, K>0
⋄ meat(M1,I1), dessert(D,K)

A=pasta, I=6, M=beef, J=5, M1=beef, I1=5, 6+5+K <= 10, 5>0, 6>0, K>0
⋄ dessert(D,K) (inconsistency)
```

[1] In the examples trivial equations are omitted

If the last computation state for this derivation is examined it can be seen that the constraint store containing `6+5+K <= 10` and `K > 0` is not satisfiable.

The answer `A=radishes, M=beef, D=fruit` is *definite* in the sense that a constant is equated with each variable in the query. However, in general answers can also be *indefinite*, i.e. the answer consists of a set of constraints representing a possibly infinite set of solutions. An example of this kind will be presented a little later when nonlinear constraints are discussed. How to extract an understandable answer from the constraints in the constraint store is an active field of research [JMSY92].

2.1.2 Nonlinear constraints

To introduce nonlinear arithmetic constraints we shall use a program multiplying two complex numbers `R1 + I*I1`, `R2 + I*I2` taken from [JL87]:

```
zmul(R1, I1, R2, I2, R3, I3):-
        R3 = R1*R2 - I1*I2,
        I3 = R1*I2 + R2*I1.
```

If the query `zmul(1,2,3,4,R3,I3)` is given, then the nonlinear equations become linear at run time, and the answer produced by e.g. CLP($\mathcal{R}$) is:

```
R3 = -5
I3 = 10

*** Yes
```

If we ask the query `zmul(1,2,R2,I2,R3,I3)`, the solution is a conjunction of two linear equalities:

```
I2 = 0.2*I3 - 0.4*R3
R2 = 0.4*I3 + 0.2*R3

*** Yes
```

This answer is an example for an indefinite solution. The solution is an infinite set of points that is represented by a minimal set of constraints stating relations between the variables of the query. To obtain precise values for `I2` and `R2` (i.e. to obtain `I2` equal to a constant and `R2` equal to a constant), the user has to further instantiate `I3` and `R3`.

For the two previous queries, there is no need for a nonlinear solver. But for the query `zmul(R1,2,R2,4,-5,10), R2 < 3` nonlinear constraints appear in the solution. CLP($\mathcal{R}$) gives the answer:

```
R1 = -0.5*R2 + 2.5
3 = R1 * R2
R2 < 3
*** Maybe
```

This is due to the property of CLP($\mathcal{R}$), whose decision procedure can only solve linear arithmetic. When a nonlinear constraint is encountered during computation, then it is delayed until it becomes linear. For the previous query, two nonlinear equations are encountered during computation. They are delayed, but no instantiation makes them linear. So at the end of the computation CLP($\mathcal{R}$) gives back the delayed constraints without knowing if there are some solutions or not (`*** Maybe`).

This introduces the need for nonlinear arithmetic solvers in constraint logic programming. Nonlinear constraints arise for instance in computational geometry [PS85], and financial applications. Several algorithms can be used to solve nonlinear constraints. Their capacities and complexities are quite different (see [Mon92a] for a comparison of different solvers). For example Gröbner bases [Buc85] treat only equations whereas quantifier elimination [Col75] can handle all (well formed) formulae over the reals at, sometimes, considerable extra cost.

For the first two queries of the previous example (multiplication of complex numbers) the answer given by nonlinear solvers is the same as the one from CLP($\mathcal{R}$). But the last query `zmul(R1, 2, R2, 4, -5, 10), R2 < 3` is completely solved, and the answer is definite:

```
R1 = 1.5
R2 = 2
```

Gröbner Bases are used in CAL [AH92], and in the system of [Mon92b]; and an improved version of quantifier elimination [Hon90] is used in RISC-CLP [Hon92].

2.2 The Boolean Domain

The most prominent applications of boolean constraints are in the area of circuit design [Sim92], here in particular hardware verification [FSTW91], and in *theorem proving* in the domain of propositional calculus [SD90, Col90]. Such applications motivated the incorporation of boolean constraint solvers into constraint logic programming languages.

Boolean terms are built from the *truth values* (*false* and *true*, represented sometimes also by 0 and 1), from *variables* and from *logical connectives* (e.g. $\vee$, $\oplus$[2], $\wedge$, *neg*). The only constraint between boolean terms is the equality (=). In some implementations (e.g. CHIP) additional constants can be used in the construction of terms. This is particularly important in hardware verification as these constants can be used to represent symbolic names for input arguments of circuits.

Each of the systems mentioned above employs quite different ways of handling boolean constraints. A *Boolean unification* algorithm [BS87] is used in the case of CHIP. In the literature a number of different unification algorithms for Boolean constraints are reported [MN90, Bue88]. Another possibility is to implement boolean constraint solving as a special case of numerical constraint solving. A modified version of the Gröbner bases algorithm [ASS+88] is used in CAL. Prolog III uses a saturation method to solve boolean constraints [Col90]. This method does not compute a most general solution and is hence not easily applicable to circuit verification. Since boolean constraint solving provides a decision procedure for propositional calculus and is therefore NP-complete, any algorithm for boolean constraints has an exponential worst case complexity. It is thus very important to use a compact description of boolean terms to achieve efficiency. CHIP [DVS+88], for example, represents boolean terms as directed acyclic graphs, which are manipulated by special purpose graph algorithms [Bry86].

The following classic example coming from hardware verification illustrates how boolean constraints can be solved by boolean unification.

% Full-adder circuit example

$add(I1, I2, I3, O1, O2) :-$
$\quad X1 = I1 \oplus I2,$
$\quad A1 = I1 \wedge I2,$
$\quad O1 = X1 \oplus I3,$
$\quad A2 = I3 \wedge X1,$
$\quad O2 = A1 \vee A2.$

[2] $\oplus$ is the exclusive or

Figure 1: Full Adder Circuit

The computation of an answer to the query $add(a, b, c, O1, O2)$ gives the following set of intermediary constraints:

$$
\begin{aligned}
&X1 = a \oplus b\\
&A1 = b \wedge a\\
&O1 = a \oplus b \oplus c\\
&A2 = c \wedge (a \oplus b)\\
&O2 = a \wedge b \oplus a \wedge c \oplus b \wedge c.
\end{aligned}
$$

The boolean solver hence produces the answer:

$$O1 = a \oplus b \oplus c, \quad O2 = a \wedge b \oplus a \wedge c \oplus b \wedge c$$

which describes the logical function of the piece of hardware. The output parameters are expressed as boolean expressions constructed from the input parameters. These boolean expressions can now be compared with the specification of the circuit, which is also expressed in terms of boolean expressions.

In case of hardware verification the full power of boolean unification is needed. But obviously boolean unification is a very costly method. For simulation tasks for instance, where the input parameters are not symbolic constants but the ground values 0 or 1, this power is not needed and other methods are more efficient. In section 3.2 and 4.2 we will describe such other techniques.

3 Consistency Techniques

3.1 Finite Domains

Consistency techniques were first introduced for improving the efficiency of picture recognition programs, by researchers in artificial intelligence [Wal72]. Picture recognition involves labelling all the lines in a picture in a consistent way. The number of potential labellings can be huge, while only very few are consistent.

Consistency techniques effectively rule out many inconsistent labellings at a very early stage, and thus cut short the search for consistent labellings. These techniques have since proved to be effective on a wide variety of hard search problems, made even wider since their integration into a logic programming framework in CHIP and subsequent CLP implementations.

The handling of constraints using consistency techniques is unlike constraint solving in the CLP Scheme, as described earlier, in that it does not guarantee to detect inconsistency of the (global) constraint store until the labelling of the problem variables is complete. Instead consistency techniques provide an efficient way to extract from the constraint store new information about the problem variables.

3.1.1 A Scheduling Example

To illustrate one such consistency technique let us take a very simple scheduling problem, with six tasks to be scheduled into a five-hour day, where each task takes an hour. The following diagram shows tasks on the left which must precede other tasks on the right:

```
      T2
     /
   T1        T6
     \      /
      T3--T5
         /
       T4
```

In addition we impose the constraint that tasks $T2$ and $T3$ cannot be scheduled at the same time.

To express this as a constraint satisfaction problem, we associate a variable Ti with the start time of each task, whose domain of possible values is $\{1, 2, 3, 4, 5\}$. We then impose the constraints

```
before(T1,T2)
before(T1,T3)
before(T2,T6)
before(T3,T5)
before(T4,T5)
before(T5,T6)
notequal(T2,T3)
```

Consistency techniques work by *propagating* information about the variables via the constraints between them. For example given that $T1 \in \{1, 2, 3, 4, 5\}$ and that $T2 \in \{1, 2, 3, 4, 5\}$, then based on the constraint `before(T1,T2)` our consistency technique deduces the information that $T1 \in \{1, 2, 3, 4\}$ and $T2 \in \{2, 3, 4, 5\}$. The value 5 is removed from the domain of $T1$ because there is no value in the domain of $T2$ which is consistent with it - that satisfies the constraint `before(T1,T2)`. The value 1 is removed from the domain of $T2$ for the same reason. (This consistency technique, which removes values inconsistent with a single constraint between two variables, is termed *arc consistency* [Mac77].)

Propagation continues until no further new domain reductions can be extracted from the constraints. The effect of applying arc consistency in our example is to reduce the domains associated with the tasks' start times as follows:

$$T1 \in \{1,2\}, T2 \in \{2,3,4\}, T3 \in \{2,3\}, T4 \in \{1,2,3\}, T5 \in \{3,4\}, T6 \in \{4,5\}$$

Consistency techniques alone can rarely be used to solve a problem, since in general there remain combinations of values in the resulting domains which are inconsistent. For example the constraint `before(T1,T2)` has been used during propagation to reduce the domains of $T1$ and $T2$, but it is still not satisfied by all the values of the resulting domains of $T1$ and $T2$. (Although $T1 = 2$ is consistent with some values in the domain of $T2$, it is not consistent with the value $T2 = 2$.)

To find a solution to this scheduling problem the system therefore performs some search, by labelling a variable with some value in its domain (search is discussed in detail in section 5 below). This choice (which may prove later to have been erroneous), allows further propagation to be attempted. For example suppose $T1$ is labelled with the value $T1 = 2$. Propagation yields $T2 \in \{3,4\}$ and $T3 \in \{3\}$. At this point the constraint `notequal(T2,T3)` is used actively for the first time to produce the information $T2 \in \{4\}$. Propagation continues until the following information has been extracted: $T1 \in \{2\}, T2 \in \{4\}, T3 \in \{3\}, T4 \in \{1,2,3\}, T5 \in \{4\}, T6 \in \{5\}$.

3.1.2 Propagation versus Solving

The treatment of the `notequal` constraint with arc consistency is a typical example of how and why consistency techniques differ from constraint solving. If variables X and Y each have domains with more than one value, then the constraint `notequal(X,Y)` will not yield any new information. The reason is that every value in the domain of Y will be consistent with at least one value in the domain of X, and vice versa. Propagation on the constraint `notequal` can be implemented very efficiently. The constraint yields no information until one of the variables has a domain with only one remaining value. This value is immediately removed from the domain of the other variable, and the constraint is satisfied. It can never again yield new information.

However if the constraint `notequal` is handled by a constraint *solver* it can yield more information than propagation. For example suppose variable X, Y and Z all have two-value domains: $X \in \{1,2\}, Y \in \{1,2\}, Z \in \{1,2\}$. The constraints

```
notequal(X,Y)  notequal(Y,Z)  notequal(Z,X)
```

are not satisfiable. Although this is detected by a solver for the `notequal` constraint, arc consistency yields no information.

For simple examples, such as this, the solver can detect the inconsistency at little cost. However non-trivial problems involve a reasonably large number of constraints and domains containing a reasonably large number of values; and in this case the cost of solving the constraints increases very quickly (exponentially) with the number of variables involved. For such problems it is often too expensive to attempt constraint solving on the `notequal` constraints, and constraint propagation proves to be a more effective technique.

3.1.3 Constraint Driven Computation

Consistency techniques extend the notion of data driven program execution. The arrival of "data" no longer means the arrival of a specific value for a variable, but rather any reduction of the domain associated with the variable. We call it *constraint driven*. In this framework new "data" may arrive many times on a single variable - each time its domain is reduced. Much research has been published on constraint propagation and its complexity, and we list some important references [MH86, Mon74, Fre78, HE80, Mac77, MF85].

For handling constraints defined extensionally as relations, there is a range of standard consistency techniques. However for particular constraints, specialised consistency techniques can be applied which take advantage of their particular semantics. The specialised techniques can support more efficient constraint propagation than the standard techniques [DV91].

For problems modelled using integers (like the scheduling example above), the constraints most often required are equations and inequations between mathematical expressions (involving the predicates $=$, $>$ $\geq$, $<$ and $\leq$). These can be efficiently handled by reasoning on maxima and minima. For example suppose X, Y and Z each have domain $\{1, \ldots, 10\}$. Reasoning on the constraint `2*X + 3*Y + 2 < Z` we use maxima and minima to remove inconsistent values from the domains of all three variables. Since 10 is the maximum possible value for Z, we can deduce that $2 * X + 3 * Y < 8$. Since the minimum value for Y is 1, it follows that $2 * X < 5$. Consequently the domain of X can be reduced to $X \in \{1, 2\}$. Similarly $Y \in \{1\}$. Finally by reasoning on the minima of X and Y we conclude that $Z \in \{8, 9, 10\}$.

Of particular importance for current day computing systems is that constraint propagation can be performed in parallel. Propagation on the different constraints can occur concurrently and asynchronously, and as long as it continues until no more domain reductions are possible the result is independent of the precise behaviour.

3.1.4 Embedding in CLP

We now illustrate the embedding of consistency techniques in a logic programming system, by expressing a couple of problems in the CHIP language (see section 6 for information about CLP languages).

The above example can be encoded in CHIP as follows:

```
?- [X,Y,Z]::1..10,
2*X + 3*Y + 2 #< Z,
indomain(X), indomain(Y), indomain(Z).
```

First the finite domain variables `X`, `Y` and `Z` over the subrange `1..10` are declared. Then the constraint that must hold between `X`, `Y` and `Z` are stated as a goal[1]. This goal is recognised by its syntax to be a constraint that will be handled by propagation. Finally the search for admissible values of `X`, `Y` and `Z` is expressed using the goal `indomain`. This goal instantiates its argument to a value in its current domain. This instantiation will cause constraint propagation to take place, which may reduce the domains of the remaining variables, or even cause a failure. If this choice proves later to have been wrong, and the system backtracks, another value in the domain will be chosen, until all the alternatives have been exhausted.

Because the domains are pruned by propagation, the two admissible combinations of values are found without any wrong guesses. For this simple example, it is an interesting exercise to write a logic program without constraints that avoids unnecessary search. For real-life problems, such an exercise is no longer interesting, and it can easily lead to unmaintainable and even incorrect logic programs. Using CLP however, we use a simple standard program structure and rely on consistency techniques for efficiency. The structure is as follows:

- Declare problem variables and their finite domains
- Set up the constraints
- Search for a solution

Notice that consistency techniques are deterministic, as opposed to the search which is non-deterministic (and usually entails backtracking). This standard structure ensures that deterministic computation during propagation is performed as soon as possible and non-deterministic computation during search is used only when there is no more propagation to be done. The importance of prioritising deterministic computation has been recognised as an important principle in the logic programming community.

It is also possible to specify user-defined predicates as constraints for propagation, by a declaration such as `lookahead`.[2]

[1] The symbol `#<` stands for `<` on finite domains.
[2] "Looking ahead" is another name used for consistency techniques [HE80]

Thus in the following program goals for the predicate `less` will be treated using consistency techniques, whilst goals for the predicate `gteq` will be treated by choice and backtracking in the normal fashion of logic programming.[3]

```
lookahead less(d,d).
less(1,1).
less(1,2).
less(2,2).
less(2,3).

gteq(2,1).
gteq(3,2).
```

The query

```
?- [X,Y,Z]:: [1,2,3,4,5], less(X,Y), less(Y,Z), gteq(X,Z)
```

is evaluated as follows. As soon as the constraints (`less(X,Y), less(Y,Z)`) are set up, the domains of the variables are reduced by consistency techniques to $\{1,2,3\}$. Now the goal `gteq(X,Z)` is invoked; the system selects the first clause defining `gteq` and attempts to add the constraints $Z = 1, X = 2$ to the constraint store. Arc consistency on `less(X,Y)` reduces the domain of Y to $Y \in \{2,3\}$, then propagation on `less(Y,Z)` reveals an inconsistency. Thus the attempt to match the first clause for `gteq` fails, and the second clause is tried. This fails similarly, and so the whole query fails.

As usual for consistency techniques, the evaluation of the constraint goals `less` are constraint driven, and there is no backtracking on these goals.

3.2 Generalised Propagation

The study of constraint propagation has been recently extended to remove the requirement for finite domains associated with the variables. One step in this direction is to admit intervals instead of finite domains (eg $1 < X < 10$ for real X) [Dav87]. However, more radically, it is possible to perform propagation without requiring either domains or intervals to be associated with the problem variables. This technique has been named *generalised propagation* [LW92a]. Generalised propagation integrates the CLP scheme, described in section 2 above, and constraint satisfaction techniques, described in this section.

In the CLP scheme an answer to a goal is a (consistent) set of constraints on the problem variables. Standard logic programming is a particular instance of the CLP scheme, where answers are expressed using equations on terms. Thus if predicate `p` is defined by

```
p(1,1).
p(2,2).
```

the query `?- p(X,Y)` has two answers $X = 1, Y = 1$ and $X = 2, Y = 2$. The idea of generalised propagation is to enable `p(X,Y)` to be used as a constraint, even though there are no domains or intervals associated with its arguments. Instead of extracting information in the form of reduced domains for X and Y, the information extracted is in the form of constraints in the current computation domain - i.e. equations between terms.

As with finite domain propagation, the information extracted must not exclude any answers to `p(X,Y)`. Thus generalised propagation only extracts information common to *all* the answers to `p(X,Y)`. Over this computation domain, the information extracted from a goal is technically the "most specific generalisation" of all the answers to the goal. In this case the most specific generalisation is $X = Y$.

[3] The `d` in `lookahead less(d,d)` signifies that this argument of `less` is a domain variable.

If `p` is handled as an ordinary predicate in the query `?- p(X,Y), p(V,W), Y=V, notequal(X,W)`, the system will backtrack four times before failing. To use `p(X,Y)` and `p(Y,Z)` as constraints for generalised propagation, it is merely necessary to annotate the query as follows:

```
?- propagate p(X,Y), propagate p(V,W), Y=V, notequal(X,W)
```

The annotation `propagate Goal` tells the system to perform generalised propagation on `Goal`, instead of treating it as an ordinary logic programming goal. Generalised propagation will immediately deduce that $X = Y$ and $V = W$. Consequently when the goals `Y=V, notequal(X,W)` are executed, the inconsistency will be detected without any backtracking.

Another example of generalised propagation is its application to the predicate `and`, defined as follows:

```
and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1).
```

Consider the query `?- propagate and(X,Y,Z), Rest` where `Rest` is some goal that performs search, eventually yielding further information about the variables X, Y and Z. Initially no information can be extracted from the constraint `and(X,Y,Z)`. However as further information is added to the constraint store, during evaluation of `Rest`, interesting propagations on `and(X,Y,Z)` may become possible. For example if the constraint $X = 0$ is added to the constraint store, generalised propagation on `and(X,Y,Z)` immediately yields the new equation $Z = 0$. Alternatively if $X = 1$ is added to the constraint store, generalised propagation yields $Z = Y$.

Like propagation, generalised propagation is a form of constraint driven computation. As more information about the problem variables becomes available, via the constraint store, further information is extracted from the constraints. All the extracted information is added to the constraint store, which enables further propagation to take place. Propagation is repeatedly attempted on all constraints until there is no more information to be extracted.

In section 4.2 below, it is described how the user can explicitly program the handling of constraints, so as to achieve a similar constraint driven behaviour for the constraint `and(X,Y,Z)`. The advantage of generalised propagation is that such constraint driven behaviour is achieved by a single annotation, and without risk of incorrectness or potential omission of possible propagation steps.

Generalised propagation yielding equality constraints, as in the above examples, has been implemented in a system called Propia [LW92b]. Programming in Propia has shown three advantages of generalised propagation.

- It is relatively simple to encode the constraints of real problems in Propia, and there is no need to explicitly add finite domains. (In fact current systems only admit finite domains of integers which implies an extra encoding step).
- It is very natural to encode a problem using a logic program without regard for efficiency. To turn such a program into a Propia program utilising generalised propagation, it is merely necessary to add annotations as in the above example. Consequently it is easy to experiment with different ways of executing the program by changing the annotations. The final program still has the same structure as the original logic programming "specification" and is therefore easy to maintain.
- Even some problems which involve finite domains prove to be solved more efficiently when encoded in Propia, than is achieved with finite domain propagation. For propositional logic problems, which can be encoded using finite domains with two values, generalised propagation produces more information than arc consistency. In fact Propia turns out to be broadly as efficient as specialised programs on a current benchmark of such problems. For problems which involve large finite domains, on the other hand, generalised propagation scores again by its simplicity: it extracts less information but it avoids wasting storage and execution time doing so. Consequently Propia can solve problems which are too big and too slow to run on existing CLP systems with finite domains.

4 Extending and Specialising the Constraint System

A given constraint system supports certain computation domains, and certain consistency techniques, enabling it to solve a range of problems efficiently. However, specialised problems may require specialised constraints, with specialised solving and consistency techniques. Two different approaches have been developed to tackle this problem. The first approach consists in identifying frequently occurring constraints and offering them via a system library. The second consists in offering the user a language to define his own constraints and the necessary propagation.

4.1 Specialised Constraints

In the past a variety of frequently occurring constraints, have been identified, which caused problems if they are encoded using the standard built-in constraints. For these, specialised constraint solving algorithms have been developed. We shortly mention some of those, developed within the CHIP system. Note that the user of these constructs need not be concerned about the implementational aspects, as they all have a declarative reading.

4.1.1 The Element Constraint

Many constraint problems use the notion of a cost function associated with a choice. This can describe the cost which we want to optimise, or it can be just an internal figure that has to be kept within certain limits. For example in a production unit switching a job from one machine to another involves a certain setup time. Now the overall time needed is restricted by some constraints. These constraints provide a pruning on the possible choices of jobs. An efficient implementation of arc-consistency for the functional constraint between choices (jobs in this example) and their costs (here the setup times) is supported via a special *element* constraint. It has the following structure: `element(N, List, Value)`, with the reading: $Value$ is the N-th value of the list $List$.

```
[M1,M2,M3] :: 1..5,       % 3 Machines, 5  jobs
alldistinct([M1,M2,M3]),  % no job is done twice
element(M1,[3,2,6,8,9],C1),
element(M2,[4,6,2,3,2],C2),
element(M3,[6,3,2,5,2],C3),
C1+C2+C3 #=  Cost,
Cost #<= 9.
```

Running this query will give the result that $M1$ does Job 1 or 2, $M2$ can do all jobs except Job 2, and $M3$ all except Job 1. The cost is guaranteed to be between 6 and 9.

Note that this constraint works in all directions, e.g. restrictions of the possible values also prune the associated index.

A variety of special constraints on lists of choices have been developed. They express e.g. that all the elements have to be different (`alldistinct`); certain values may not occur more than a certain number of times (`atmost`, as exampled below); that only one variable may take a certain value, etc. A special constraint - *cumulative* - developed for scheduling and loading problem has been recently presented in [AB92].

```
[chipc 7]: [A,B,C] :: 0..5, atmost(1,[A,B,C],5),B=5.

A = $_267  [0..4]
C = $_287  [0..4]
B = 5
yes.
```

4.2 User Defined Constraints

The implementation of special constraints can only be done by the system designer. But as it is useful to have special constraint solving mechanisms available the trend now is to develop tools to allow the constraint solving behaviour necessary for the specific application to be defined by the application programmer. Given these tools provide means for a simple declarative specification, they once again support one key concept behind logic programming: the programming time is reduced, different possibilities can be tested easily, and support of the software becomes easier.

In this section we discuss facilities for the user to control the evaluation of constraints, to specify constraint-driven computation, and to define constraint solvers for new constraints.

4.2.1 Delay Declarations

We saw above that for certain constraints (like non-linear constraints in CLP($\mathcal{R}$)) it is necessary to delay their handling until certain variables have a specific value. In some systems the delaying of the appropriate constraints is built into the system. Often, however, the user needs to be able to control the delaying of goals and constraints. An example of a declaration to delay the handling of a goal till a certain condition is satisfied is

```
delay employee(Nr,Sal) until ground(Nr)
```

Such a declaration will prevent the system from trying to look up salaries for employees until a specific employee number is known. The declaration would also postpone the application of consistency techniques to this goal, in case `employee` was a constraint.

Declarations are annotations applied to a program which refer to the program text. As such they are termed "meta-commands" to distinguish them from commands within the program which manipulate the data.

4.2.2 Guards

There is another approach to providing user control based on the concept of a *guard*. The guard defines a logical condition, and is part of the program itself rather than a meta-command. An example of a guarded clause is

```
and(X,Y,Z) <=> X=0 | Z=0
```

The guard is $X = 0$. When the current set of goals include an atomic goal of the form `?- and(A,B,C)` the guard is used to control when, or if, the clause can be applied. Specifically it can be applied as soon as the constraint store contains, or implies that, $X = 0$. As soon as this is true, the atomic goal can be rewritten into its body (in this case $Z = 0$). Hence the query `?- and(X, Y, Z),X = 0` will result in a constraint store `X = 0, Z = 0`.

A special feature of definitions by guarded clauses is that when a guard is satisfied, the system *commits* to the clause and there is never any backtracking to alternative clauses. This means that guarded clauses define a computation with "don't care" nondeterminism, rather than the "don't know" nondeterminism of logic programming which involves backtracking to check the other alternatives. The declarative semantics of logic programming is sacrificed with the move to don't care nondeterminism, unless strict conditions are met by the guarded clauses as given in [Mah87]. An advantage is that the guards can be evaluated concurrently, which is why guarded clauses are interesting for concurrent CLP, discussed later in this section.

The control offered by the guards is precisely constraint driven computation, without backtracking, as needed to explicitly encode constraint propagation

4.2.3 Example

We shall take as an example the `and` constraint used earlier in our discussion of generalised propagation.

Declaratively `and` is defined as follows:

```
and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1).
```

We can specify a propagation behaviour for handling `and` goals using the following guarded clauses:

```
and(X,Y,Z) ⇔ X=0 | Z=0.
and(X,Y,Z) ⇔ Y=0 | Z=0.
and(X,Y,Z) ⇔ Z=1 | X=1,Y=1.
and(X,Y,Z) ⇔ X=1 | Y=Z.
and(X,Y,Z) ⇔ Y=1 | X=Z.
and(X,Y,Z) ⇔ X=Y | Z=X.
```

Notice that the information $Z = 0$ is not sufficient to allow any further consequences to be extracted from the `and` constraint. Thus if the constraint store only contains $Z = 0$ none of the guards are satisfied. In this case more information on X or Y will be needed before any of the clauses can fire.

Consider the full-adder circuit

```
add(I1,I2,I3,O1,O2) :-
    xor(I1,I2,X1),
    and(I1,I2,A1),
    xor(X1,I3,O1),
    and(I3,X1,A2),
    or(A1,A2,O2).
```

together with rules for the logical gates (as was exemplified by the rules for the and-gate).

The query `add(I1,I2,0,O1,1)` will produce `I1=1,I2=1,O1=0`. The computation proceeds as follows: Because `I3=0`, the result of the and-gate with input `I3`, the output `A2`, must be 0. As `O2=1` and `A2=0`, the other input `A1` of the xor-gate must be 1. Because `A1` is also the output of an and-gate, its inputs `I1` and `I2` must be both 1. Hence the output `X1` of the first xor-gate must be 0, and therefore also the output `O1` of the second xor-gate must be 0.

In this particular case the same behaviour is obtained by applying generalised propagation to the declarative specification of `and`. However the facility to define explicitly what propagation is to take place on a given goal means that tailored propagation behaviour can be obtained for particular applications.

4.2.4 Embedding in CLP

CHIP was the first constraint logic programming language to introduce constructs to specify user-defined constraint propagation. Their need was realised in applications for diagnosis and test pattern generation of digital circuits [SD87, Sim89]. They have been called "demon constructs" [DVS+88] because of their event-driven activation. CHIP introduces in addition conditional propagation with the `if-then-else` construct. A framework for using guarded rules for constraint handling is given in [Smo91].

4.2.5 Constraint Solving

To express constraint *solving* it is necessary to be able to handle the interaction of multiple constraints. Consequently a multi-headed guarded rule is introduced. A unified approach encompassing single- and multi-headed guarded clauses has been developed under the name Simplification Rules [Frü92]. Two rules encoding a solver for the `greater` constraint are as follows:

```
greater(X,Y) <=> X=Y | fail                          % irreflexivity
greater(X,Y), greater(W,Z) => Y=W | greater(X,Z)     % transitivity
```

(If the second clause is executed it does not *replace* the goal with the body, it merely *augments* the current set of remaining goals with the clause body.)

The above rules capture the transitivity and irreflexivity of `greater` but not its semantics: "less" is also transitive and irreflexive! We now add one further guarded rule to check that `greater` is indeed the same as the built-in comparator ">":[1]

```
greater(X,Y) <=> ground(X), ground(Y) | X>Y
```

4.2.6 Concurrent Constraints

User-defined constraint propagation and simplification is a very active area of research in constraint logic programming. A framework including a powerful set of constraint constructors is described in [VD91]. The concept of constraint agents, and their transformational semantics underlies much ongoing work, e.g. [Sar92, Van91]. The idea behind all these approaches is to express constraint evaluation in terms of concurrent computations. The first such concurrent constraint logic programming language has been suggested in [Mah87]. In [Sar92] a general framework for these languages has been developed based on the notion of ask & tell. The basic operation in these languages, besides telling (adding) a constraint to the constraint store and deciding its consistency, is to ask for a constraint, i.e. to decide if this constraint is entailed (implied) by the constraint store. Algorithms for constraint entailment are extensions of constraint solving algorithms. In case of demons above this simplifies to deciding whether the variables in the guard have certain values or not.

[1] Since *groundness* is a meta-concept, some people prefer to use the delay declaration instead of a guard for this control. The framework of simplification rules supports control by both guards and delays

5 Search and Optimization in CLP

As outlined above the key idea behind constraint reasoning systems is to tackle complex tasks by incrementally inferring properties of the problem solutions and using this information to enforce consistency [Van89]. This deterministic knowledge is acquired in an explicit form. It is therefore possible to prune the space of possible alternatives, i.e. excluding certain cases (choices) that need not be considered in the future.

As in general the solution cannot be inferred right away after the deterministic reasoning steps some assumptions about the problem solution have to be made. Those assumptions are fed back into the constraint reasoning scheme, thus yielding more information about the solution. This process continues until a solution is obtained.

If an inconsistent solution description is obtained the assumptions have to be withdrawn. In this case the process has to be continued with alternative assumptions. This process is usually referred to as backtracking. The nature of this is another reason why constraint propagation fits well in the Prolog language, which supports a backtracking mechanism.

Note that the *generate and test* approach uses the same schema, but the inference engine is only used when complete solutions are obtained, i.e. only a test is done, if a complete solution candidate has been produced.

The constraints reasoning schema depends crucially on two aspects:

- The inference power of the reasoning engine.
- The strategy to make the assumptions.

In this section we will concentrate on the second aspect. In general this is referred to as the *search*. We will concentrate on the *finite domain* case, where this process is also called *labelling*.

5.1 Aspects of Search

In AI, problem solving is classically seen as a state space search: solving a problem is to find a path from an initial state to the goal state - representing the solution. Within that framework search is the general mechanism that is used when no other, better method is known.

Similarly in constraint reasoning we refer to search, if the constraint handler cannot provide us with more information. But note that we deal here with partial solutions: e.g. in each state of the search we know some variable values but not all. In a traveling salesman problem (TSP), for example, the instantiated variables represent known parts of the route. Once we do a search step we assume that a certain city should be visited best at a certain point of the trip.

Taking a search step within a constraint reasoning framework involves two decisions:

1. On which aspect of the problem do we want to make an assumption ?
2. What should that assumption should be ?

5.1.1 The Right Granularity

In general it is important that the granularity and the strategy of the search process fits well with the constraint handler. The right choices here are crucial for the performance of the overall system. The assumptions made during search perform two roles. First they are queries about the solution. Secondly, and even more important, they provide input to the constraint handler which performs reasoning on the constraints and their impact on other problem variables. With the right input the solver will be able to prune large parts of the search space, thus yielding a good problem solving performance.

5.1.2 Declarativeness

Within the approaches discussed below the strategy for selecting variables/values can be defined declaratively. This means that the complexity of the program used to define the strategy is independent of the complexity of the strategy itself. This has the important consequence that certain real world problems can still be tackled with this declarative technology, while specialised procedural constructs are hard to build. In fact it is has been our experience that CLP solves problems that are new in the sense that they have not been solved systematically by software so far - despite the fact that specialised algorithms have been known.

5.2 Labelling Strategies

Within the CHIP system the user is free to program his own search strategy. This can be done easily with the support of the underlying Prolog system. As some general approaches have given good results they are already incorporated into the system. They make labelling based

- on individual problem variables and
- on single values for those variables.

The problem variables in the TSP example are the stops on the tour, the values are the location of those stops.

In many cases it is most effective to use the variable with the smallest remaining domain for labelling. This principle is often referred to as *first fail principle* as with fewer choices possible we will find out earlier if those were right or wrong. Alternatively the variable which occurs in most constraints can be chosen. Several combinations of these principles are possible [Van89].

```
% label(Problem_variables)
label([]).
label(Problem_variables) :-
        deleteff(Var, Problem_variables, Rest_vars),
             % choose var with minimal choices
        indomain(Var),
             % choose a value from its domain
        label(Rest_vars).
```

Which value then to give that chosen variable is harder to answer in general. For some problems it is possible to define a metric, with the 'smallest' values being most promising.

For the map colouring problem good results have been obtained by rotating the colours used for labelling. I.e. for the country A use the first colour, for country B the second and so on. This approach has the effect of the intuitively appealing idea of using different colours whenever possible, as connected countries have to have different colours.

```
% special labelling routine for map colouring example
% label_colour(Countries, Available_Colours)

label_colour([],_).
label_colour([First|Rest],Colours):-
        member(First,Colours),
        rotate(Colours,Colours1),
        label_colour(Rest,Colours1).

rotate([A,B,C,D],[B,C,D,A]).
```

Within the generalised propagation schema it can be very natural to use an entire tuple of values that satisfies a constraint, as the tuples satisfying / defining a constraint are usually available. E.g. if we want to solve a crossword puzzle, it makes sense to put (assume) a word in a certain position, which means labelling a set of variables with characters at the same time.

5.2.1 Labelling with several values

In some cases selecting a specific value for a variable can be a very strong assumption. It can therefore be better to make an assumption on the set of possible values of that variable. The classical approach here is to make a binary chop of the domain. This means that we cut the domain in two halves and then assume that the value is in one half. This can be done by stating an additional constraint which excludes the other half. This technique has been used successfully for the cutting stock application [DSV88].

```
% binary chop labelling routine

label_chop([]).
label_chop([X|Vars]):-
        mindomain(X,Min),
        maxdomain(X,Max),
        Mid is (Min + Max)//2,
        above_or_below(X,Mid),
        label_chop(Vars).

above_or_below(X,Mid) :-
        X #<= Mid.  % set up additional constraint
above_or_below(X,Mid) :-
        X #> Mid.
```

5.3 Branch and Bound

Due to the incremental approach of constraint solving branch and bound strategies fit well with it. For a constraint problem with minimization the current minimal value of the target function is maintained. As soon as a choice / search step is done that would increase that value again, this is rejected. Thus parts of the search tree need not be considered. If a new minimum value has been obtained a new branch and bound run with that value can be invoked. Note that the previously considered combinations need not be considered again, as the current minimum is known to be optimal with regard to the search space already considered.

Given the classical setup of a CHIP program:

```
solve(Vars) :-
        define_vars(Vars),
        setup_constraints(Vars),
        label(Vars).
```

the program for the minimal solution can be written easily: a labelling routine that produces the cost values is combined with the minimise declaration.

```
% 2-dimensional cutting stock example
% Vertical and horizontal cuts, Waste Produced

label_min(Vert_Hor_Cuts) :-

        minimise(
           label_waste(Vert_Hor_Cuts, Waste),
```

```
        % labelling routine that also computes the waste
Waste).
        % minimise Waste value
```

As seen in the example in CHIP the declaration to use the branch and bound minimization schema is very simple to be added to a program. For some problems this approach gives quite good performance results.

5.4 Optimization and Advanced Search

For optimization problems it is not always easy to infer deterministic information about the optimal solution. If fewer inferences can be made the proper choice of assumptions will become more important.

5.4.1 Local Search

One approach is to improve the current assumptions by local search. The idea is here that an initial solution - satisfying the constraints - is improved in terms of the cost function to be minimised. An operator is defined that maps one solution to others that are similar (in the sense that most of the variables retain the same value). The operator must, of course, ensure the constraints are still satisfied.

Search in this framework means applying the operator to the current solution. If the new solution has lower cost it becomes the current solution, and search continues until a solution is reached which cannot be improved upon by a single application of the operator. The final solution is better than its immediate neighbours, but there may be still better solutions in another part of the search space. In other words the final solution may only be a "local" optimum. This approach works well for the unconstrained traveling salesman problem [LLKS90], where a typical operator is one that exchanges two edges of a tour. To apply this approach to constrained problems, it is necessary to impose constraints on the operator that maps solutions to new solutions. Currently available systems do not offer this feature.

5.4.2 Novel Search Techniques

The so called 'novel search techniques' suggest different ways of moving through the search space, while more or less implicitly information about the solution is acquired and used to further guide the search. Currently they offer the best approaches to solving many important classes of optimization problems, as e.g. the TSP. For an overview of these techniques see [Küc92b].

The main disadvantage of these techniques is their missing completeness and correctness properties. There is no guarantee that a certain mechanism will ever find an optimal or even constraint satisfying solution. Therefore these approaches are often ruled out for real world applications where certain requirements - hard constraints - definitely have to be met. On the other hand it is not necessary to always obtain *the* optimal solution with regard to the cost function - which may be very hard to compute - but rather a good solution can be sufficient [HT85]. Many real world problems are a mix of constraint satisfaction problems and optimization problems. A classical example is the vehicle scheduling problem. A fleet of vehicles has to deliver goods to customers with minimal effort. The problem is related to the TSP-optimization problem, but additional constraints also have to be met. Those are for example the capacity constraints of the vehicles. It may thus be permitted to offer a solution that is not optimal with regard to the length of the proposed tour, but in any case none of the vehicles may be overloaded.

In an ideal system for constraint optimization an advanced search mechanism is combined with a constraint solver. It is a current research topic to consider such a combination in detail.

6 CLP systems

This section reviews some constraint programming systems and discusses briefly their most important features. This list cannot be complete and is not intended to be. The objective of the single descriptions is not to be exhaustive, but rather to give a rough idea of the presented system. The interested reader is referred to the cited literature for each of the systems.

6.1 CHIP

The Constraint Logic Programming language CHIP [DVS+88] has been developed at the European Computer-Industry Research Centre (ECRC). The most important feature of the CHIP system is the introduction of arithmetic constraints over finite domains solved by consistency techniques. In addition CHIP provides a rich set of symbolic constraints. Minimization is done by a branch and bound technique.

Beside constraints over finite domains CHIP provides the following constraint solvers:

- Boolean constraints are solved with a Boolean unification algorithm
- Linear rational constraints are handled by an extended Simplex algorithm.

Finally as already mentioned CHIP gives the user the possibility to define his own constraints and control their execution. The demon rules are most prominent. Conditional propagation based on an if-then-else construct is another way to control the evaluation of constraints.

Based on the CHIP technology there are currently four different commercial products available or under development. Bull is offering the finite domain technology within its CHARME system, ICL has a product called DECISION POWER based on the CHIP/SEPIA compiler [MAC+89, AB91]. Siemens-Nixdorf Informationssysteme are currently developing their new version of SNI-Prolog, which will incorporate the whole CHIP technology. Finally the CHIP interpreter has been productised by the French company COSYTEC.

CHIP's finite domain constraints, and generalised propagation, have been integrated into the OR-parallel logic programming platform ElipSys [DSVX91]. Currently a successor to CHIP is under development at ECRC. It will provide integration of new constraint solvers [Mon92b]; generalised propagation [LW92b] working on various computation domains; constraint simplification rules [Frü92]; and novel search techniques [Küc92a].

6.2 CLP($\mathcal{R}$)

The Constraint Logic Programming language **CLP($\mathcal{R}$)** [JMSY90] has been developed as a demonstrator for the CLP(X) scheme at Monash University, IBM Yorktown Heights and Carnegie Mellon University. The constraint domain of CLP($\mathcal{R}$) is real linear arithmetic. As already mentioned non-linear constraints are delayed. The underlying constraint solver is an extended Simplex algorithm. Currently there are two implementations available from IBM / Carnegie Mellon University, an interpreter and since recently a compiler-based version.

6.3 Prolog-III

PROLOG III [Col90] is the CLP language developed at the University of Marseille and at Prologia in France. It includes three new constraint domains: linear rational arithmetic, boolean terms and finite strings (or finite lists).

- Linear rational arithmetic is handled via an extended Simplex algorithm.
- The boolean constraint-solver is based on a saturation method.

- The facilities of PROLOG III for finite string (lists) processing is explained below. The constraint solver is based on a restricted string unification algorithm.

For finite strings there exists a single function to concatenate two strings, denoted by "." and the only constraint is the equality constraint. To illustrate how these finite strings may be used consider the following problem (from [Col90]).

Find the string(s) `Z` *such that* `<1,2,3>.Z = Z.<2,3,1>`

There are in fact an infinite number of solutions. Hence Prolog III delays the evaluation of such constraints until their length is known. Let us consider the string length 10 (the length operator is infix in Prolog-III and denoted by the operator `::`).

```
{ Z :: 10, <1,2,3>.Z = Z.<2,3,1> } ;
```

The system comes back with the single solution:

```
{ Z = <1,2,3,1,2,3,1,2,3,1> }
```

PROLOG III is a commercial product of Prologia, Marseille.

6.4 Trilogy

Trilogy [Vod88] is a constraint programming language developed at Complete Logic Systems in Vancouver. The constraint domain of Trilogy is integer arithmetic, i.e. it allows linear equations, inequations, and disequations over integers and integer variables to be expressed. The solver is based on a decision procedure for Presburger Arithmetic. Unlike other CLP systems TRILOGY is not integrated into a Prolog environment, but it is based on an own "theory of pairs" [Vod88]. Trilogy is compiled into native code for PCs as target machines. It can be acquired via Complete Logic Systems in Vancouver.

6.5 CAL and GDCC

CAL [ASS+88] (Constrainte Avec Logique), developed at ICOT, Tokyo, was the first CLP language to provide non-linear constraints. During the last few years a parallel version of CAL has been developed at ICOT, called GDCC [AH92]. The system can handle constraints in the following domains:

- Non-linear real equations are solved with a Gröbner Base algorithm.
- The constraint solver for boolean constraints is based on a modified Gröbner Base algorithm.
- Linear rational arithmetics are again solved with a Simplex algorithm. A branch and bound method has been implemented on top of this constraint solver to solve integer optimization problems.

Both CAL and GDCC are available from ICOT, Tokyo.

6.6 BNR-Prolog

BNR-Prolog [Bel88] has been developed at Bell-Northern Research, Ottawa. It has been specifically designed for Apple Macintosh. The interesting feature of BNR-Prolog from a CLP point of view is the introduction of the so called relational arithmetic. This new constraint domain is based on a new interval variable representing a real number lying between lower and upper bound of this interval. The constraint handler is based on interval arithmetic [OV90]. The system can be acquired from Bell-Northern Research, Ottawa.

6.7 RISC-CLP

RISC-CLP [Hon92] is a prototype system in the domain of real arithmetic terms. It has been developed at the RISC, Linz. It can handle any arithmetic constraints over the reals. The constraint solver behind is an improved version of Tarski's quantifier elimination method [Hon90].

7 Conclusion

This paper aimed at giving an informal introduction into the different concepts of CLP. It tried to explain the philosophy behind the main ideas in CLP and illustrate them by examples. Emphasis has been put on the practically relevant parts.

CLP is successfully employed in a large variety of applications, in particular ones that can be expressed as constrained search problems. While keeping the main features of logic programming, i.e. declarativeness and flexibility, CLP brings into these languages

- the efficiency of special purpose algorithms written in imperative languages and
- the expressiveness of the different constraint domains it embodies.

The main advantage of CLP compared to other approaches is that it drastically reduces development time and provides more flexibility offered by the solution while showing an efficiency comparable to solutions written in procedural languages.

Constraint Logic Programming is moving out of the research labs into the commercial world. A number of products based on this technology are offered today. These products have been applied in a large range of very different application which are in use. Amongst them:

- At Hongkong International Terminal and in the Harbour of Singapore the resource planning and scheduling system controls ships, cranes, containers and stacks.
- At Cathay Pacific the Movement Control Systems supports the planning and scheduling of their entire fleet.
- At the French national railway SNCF movements of empty waggons are optimised.
- Within Siemens CLP is supporting the circuit designers with their Circuit Verification Environment.
- In the ESPRIT projects APPLAUSE, CHIC and PRINCE a large number of applications is currently under development.

CLP is still very much under development. The main practical systems are implemented to run on a single processor. Many researchers are studying concurrent constraint handling and parallel implementations of CLP. Secondly constraints in existing systems need to be well-understood by the end user if he is to obtain maximum benefit of them in his programs. The development of cleaner and simpler ways to specify constraint behaviour will be essential for its future industrial acceptance. Thirdly the practical requirement to integrate constraint handling with other software techniques and systems is becoming pressing. The current work on integrating CLP with data base technology is an important step in this direction.

Acknowledgements

The authors would like to thank the following people for the discussions they had with them and for their encouragement and support: Abder Aggoun, Nicolas Beldiceanu, Françoise Berthier, Gérard Comyn, Mehmet Dincbas, Hervé Gallaire, Thomas Graf, Micha Meier, Joachim Schimpf, Helmut Simonis, Pascal Van Hentenryck, André Veron. Many thanks to Norbert Eisinger for reading a draft of this paper and suggesting improvements.

Bibliography

[AB91] A. Aggoun and N. Beldiceanu. Overview of the CHIP Compiler System. In K. Furukuwa and P. Deransart, editors, *Proceedings of the 8th International Conference on Logic Programming*, pages 775–789, June 1991.

[AB92] A. Aggoun and N. Beldiceanau. Extending CHIP in order to solve complex scheduling problems. Technical report, COSYTEC, 1992.

[AH92] A. Aiba and R. Hasegawa. Constraint Logic Programming Systems - CAL, GDCC and Their Constraint Solvers. In *Proceedings of FGCS 92*, pages 113–131, 1992.

[ASS+88] A. Aiba, K. Sakai, Y. Sato, D. J. Hawley, and R. Hasegawa. Constraint Logic Programming Language CAL. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-88), ICOT, Tokyo*, pages 263–276, december 1988.

[Bel88] Bell-Northern Research Ltd BNR. BNR-Prolog User Guide. Technical report, Bell-Northern Research Ltd., 1988.

[Ber89] F. Berthier. A financial model using qualitative and quantitative knowledge. In F. Gardin, editor, *Proceeedings of the International Symposium on Computational Intelligence 89*, Milano, Italy, September 1989.

[Bry86] R. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

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

[Buc85] B. Buchberger. Gröbner Bases: an Algorithmic Method in Polynomial Ideal Theory. In N. K. Bose Ed., editor, *Multidimensional Systems theory*, pages 184–232. D. Reidel Publishing Company, Dordrecht - Boston - Lancaster, 1985.

[Bue88] W. Buettner. Unification in finite algerbas is unitary (?). In *Proceedings CADE-9*. LNCS 310, Springer-Verlag, 1988.

[Coh90] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[Col75] G. E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *Proceedings of the Second GI Conference on Automata Theory and Formal Languages*, pages 515–532. Springer Lecture Notes in Computer Science 33, 1975.

[Col90] Alain Colmerauer. An introduction to prolog-III. *Communications of the ACM*, 33(7):69–90, July 1990.

[Dav87] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[DSV88] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving a Cutting-Stock Problem in Constraint Logic Programming. In *Fifth International Conference on Logic Programming*, Seattle, WA, August 1988.

[DSV90] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):74–94, 1990.

[DSVX91] M. Dorochevsky, K. Schuerman, A. Véron, and J. Xu. Constraints Handling, Garbage Collection and Execution Model Issues in ElipSys. In Springer Verlag, editor, *Parallel Execution of Logic Programs, ICLP'91 Pre-Conference Workshop Proceedings*, pages 17–28, Paris, June 1991.

[DV91] Y. Deville and P. Van Hentenryck. An efficient arc consistency algorithm for a class of csp problems. In *Proc. of the 13^{th} IJCAI*, Sydney, Australia, August 1991.

[DVS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings on the International Conference on Fifth Generation Computer Systems FGCS-88*, Tokyo, Japan, December 1988.

[FHK+92] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint logic programming - an informal introduction. In *Logic Programming Summer School, Zürich*, September 1992. Springer Lecture Notes.

[Fre78] E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, November 1978.

[Frü90] Thom Frühwirth. Constraint logic programming - an overview. Technical Report Technical Report E181-2, Christian Doppler Laboratory For Expert Systems, August 1990.

[Frü92] Thom Frühwirth. Simplification rules. Technical report, ECRC, Munich, Germany, 1992.

[FSTW91] T. Filkorn, R. Schmid, E. Tiden, and P. Warkentin. Experiences from a large industrial circuit design application. In *ILPS*, San Diego, California, October 1991.

[Gal85] H. Gallaire. Logic programming: Further developments. In *IEEE Symposium on Logic Programming*, pages 88–99. IEEE, Boston, July 1985.

[HE80] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, October 1980.

[HHLM91] R. Helm, T. Huynh, C. Lassez, and K. Mariott. A linear constraint technology for user interfaces. Technical Report RC 16913, IBM Yorktown Heights, 1991.

[Hon90] Hoon Hong. *Improvements in CAD-Based Quantifier Elimination.* PhD thesis, Ohio State University, Computer and Information Science Research Center, Colombus, Ohio, USA, 1990.

[Hon92] H. Hong. Non-linear Constraints Solving over Real numbers in Constraint Logic Programming (Introducing RISC-CLP). Technical report, RISC, Linz, 1992.

[HT85] J. Hopfield and D. Tank. 'Neural' computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[JMSY90] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) language and system. Technical Report RC 16292 (#72336) 11/15/90, IBM Research Division, November 1990.

[JMSY92] J. Jaffar, M. Maher, P. Stuckey, and R. Yap. Output in CLP($\mathcal{R}$). In *Proceedings the FGCS'92*, Tokyo, 1992.

[KKR90] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. In *Proceedings of PODS 90*, pages 299–313W, 1990.

[Kow79] R Kowalsi. *Logic for Problem Solving.* North-Holland, New York, Amsterdam, Oxford, 1979.

[Küc92a] Volker Küchenhoff. Clp and novel search techniques: an integration. Technical report, ECRC, Munich, Germany, 1992.

[Küc92b] Volker Küchenhoff. Novel search techniques - an overview. Technical report, ECRC, Munich, Germany, January 1992.

[LLKS90] E. Lawler, J. Lenstra, R. Kan, and D. Shmoys. *The Traveling Salesman Problem.* John Wiley and Sons, 1990.

[LMY87] Catherine Lassez, Ken McAloon, and Roland Yap. Constraint logic programming and options trading. *IEEE Expert, Special Issue on Financial Software*, (3):42–50, August 1987.

[LW92a] T. Le Provost and M. Wallace. Constraint Satisfaction Over the CLP Scheme. Technical Report ECRC-92-1, ECRC, 1992. to appear.

[LW92b] T. Le Provost and M. Wallace. Domain Independent Propagation. In *Proceedings on the International Conference on Fifth Generation Computer Systems 1992 FGCS-92*, pages 1004–1012, Tokyo, Japan, June 1992.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Mac86] A.K. Mackworth. Constraint satisfaction. In *Encyclopedia of Artifical Intelligence*, 1986.

[MAC+89] M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D. Henry de Villeneuve, A. Herold, P. Kay, B. Perez, E. van Rossum, and J. Schimpf. SEPIA - An Extendible Prolog System. In *Proceedings of the 11th World Computer Congress IFIP'89*, San Francisco, August 1989.

[Mah87] M. J. Maher. Logic semantics for a class of committed-choice programs. In *Proc. 4th International Conference on Logic Programming*, pages 858–876, Melbourne, Australia, May 1987.

[MF85] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[MH86] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[MN90] U. Martin and T. Nipkov. Boolean unification - the story so far. In C. Kirchner, editor, *Unification*. Academic Press, 1990.

[Mon74] U. Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

[Mon92a] E. Monfroy. A Survey of Non-Linear Solvers. Technical Report 91-15i, ECRC, January 1992. also CHIC (ESPRIT project) report D5.1.2.1.

[Mon92b] E. Monfroy. Non Linear Constraints: a Language and a Solver. Technical Report ECRC-92, ECRC, Munich, Germany, 1992. to appear.

[OV90] W. Older and A. Vellino. Extending prolog with constraint arithmetics on ral intervals. In *Canadian Conference on Computer and Electrical Engineering*, Ottawa, Canada, 1990.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[SA89] K. Sakai and A. Aiba. A Theoretical background of Constraint Logic Programming and its Applications. *Journal of Symbolic Computation*, 8(6):589–603, December 1989.

[Sar92] V. A. Saraswat. *Concurrent Constraint Programming Languages*. MIT Press, 1992.

[SD87] 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.

[SD90] H. Simonis and M. Dincbas. Propositional calculus problems in chip. In H. Kirchner, editor, *Proceedings of the 2nd International Conf on Algebraic and Logic Programming*, Nancy, France, October 1990. CRIN and INRIA-Lorraine, Springer Verlag.

[Sim89] H. Simonis. Test Generation Using the Constraint Logic Programming Language CHIP. In *Proceedings of the 6th International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

[Sim92] H. Simonis. *Constraint Logic Programming as a Digital Circuit Design Tool*. PhD thesis, 1992. (submitted).

[Smo91] Gerd Smolka. Residuation and guarded rules for constraint logic programming. Technical report, Digital Equipment Paris Research Laboratory Research Report, June 1991.

[Ste80] G. L. Steele. The definition and implementation of a computer programming language based on constraints. Technical Report MIT-AI TR 595, Dept. of Electrical Engineering and Computer Science, M.I.T., August 1980.

[Van88] P. Van Hentenryck. A Constraint Approach to Mastermind in Logic Programming. *ACM Sigart*, (103), January 1988.

[Van89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, Cambridge, MA, 1989.

[Van91] Pascal Van Hentenryck. Constraint logic programming. *The Knowledge Engineering Review*, 6(3):151–194, 1991.

[VD91] Pascal Van Hentenryck and Yves Deville. The cardinality operator: A new logical connective for constraint logic programming. In *Proc. of the* 8^{th} *Int. Conf. on Logic Programming*, pages 745–759, Paris, France, 1991. MIT Press.

[Vod88] P. Voda. The constraint language trilogy: Semantics and computations. Technical report, Complete Logic Systems, North Vancouver, BC, Canada, 1988.

[Wal72] D. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI271, MIT, Massachusetts, November 1972.

Contents

Other Reports Available from ECRC

[ECRC–TR–LP-60] Mireille Ducasse and Anna-Maria Emde. *Opium 3.1 - User Manual A High-level Debugging Environment for Prolog*. 1991.

[ECRC–TR–LP-61] E. Yardeni, T. Frühwirth, and E. Shapiro. *Polymorphically Typed Logic Programs*. 1991.

[ECRC–TR–DPS-81] U. Baron, S. Bescos, and S. Delgado. *The ElipSys Logic Programming Language*. 17. 01. 1991.

[ECRC–TR–DPS-82] Sergio Delgado, Michel Dorochevsky, and Kees Schuerman. *A Shared Environment Parallel Logic Programming System On Distributed Memory Architectures*. 18. 01. 1991.

[ECRC–TR–DPS-83] Andre Veron, Jiyang Xu, and Kees Schuerman. *Virtual Memory Support for OR-Parallel Logic Programming Systems*. 05. 03. 1991.

[ECRC–TR–DPS-85] Michel Dorochevsky. *Garbage Collection in the OR-Parallel Logic Programming*. 15. 03. 1991.

[ECRC–TR–DPS-100] Alan Sexton. *KCM Kernel Implementation Report*. 22. 05. 1991.

[ECRC–TR–DPS-103] Michel Dorochevsky. *Key Features of a Prolog Module System*. 08. 03. 1991.

[ECRC–TR–DPS-104] Michel Dorochevsky, Kees Schuerman, and Andre Veron. *ElipSys: An Integrated Platform for Building Large Decision Support Systems*. 29. 01. 1991.

[ECRC–TR–DPS-105] Jiyang Xu and Andre Veron. *Types and Constraints in the Parallel Logic Programming System ElipSys*. 15. 03. 1991.

[ECRC–TR–DPS-107] Olivier Thibault. *Design and Evaluation of a Symbolic Processor*. 13. 06. 1991.

[ECRC–TR–DPS-112] Michel Dorochevsky, Jacques Noyé, and Olivier Thibault. *Has Dedicated Hardware for Prolog a Future ?* 14. 09. 1991.

[ECRC–91–1] Norbert Eisinger and Hans Jürgen Ohlbach. *Deduction Systems Based on Resolution*. 29. 10. 1991.

[ECRC–91–2] Michel Kuntz. *The Gist of GIUKU: Graphical Interactive Intelligent Utilities for Knowledgeable Users of Data Base Systems*. 4. 11. 1991.

[ECRC–91–3] Michel Kuntz. *An Introduction to GIUKU: Graphical Interactive Intelligent Utilities for Knowledgeable Users of Data Base Systems*. 4. 11. 1991.

[ECRC–91–4] Michel Kuntz. *Enhanced Graphical Browsing Techniques for Collections of Structured Data*. 4. 11. 1991.

[ECRC–91–5] Michel Kuntz. *A Graphical Syntax Facility for Knowledge Base Languages*. 4. 11. 1991.

[ECRC–91–6] Michel Kuntz. *A Versatile Browser-Editor for NF2 Relations*. 4. 11. 1991.

[ECRC–91–7] Norbert Eisinger, Nabiel Elshiewy, and Remo Pareschi. *Distributed Artificial Intelligence - An Overview*. 4. 11. 1991.

[ECRC–91–8] Norbert Eisinger. *An Approach to Multi-Agent Problem-Solving*. 11. 11. 1991.

[ECRC–91–9] Klaus H. Ahlers, Michael Fendt, Marc Herrmann, Isabelle Hounieu, and Philippe Marchal. *TUBE Implementor's Manual*. 21. 11. 1991.

[ECRC–91–10] Klaus H. Ahlers, Michael Fendt, Marc Herrmann, Isabelle Hounieu, and Philippe Marchal. *TUBE Programmer's Manual*. 21. 11. 1991.

[ECRC–91–11] Michael Dahmen. *A Debugger for Constraints in Prolog*. 26. 11. 1991.

[ECRC–91–12] Jean-Marc Andreoli and Remo Pareschi. *Communication as Fair Distribution of Knowledge*. 26. 11. 1991.

[ECRC–91–13] Jean-Marc Andreoli, Remo Pareschi, and Marc Bourgois. *Dynamic Programming as Multiagent Programming.* 26. 11. 1991.

[ECRC–91–14] Volker Küchenhoff. *On the Efficient Computation of the Difference Between Consecutive Database States.* 5. 12. 1991.

[ECRC–91–15] Sylvie Bescos and Michael Ratcliffe. *Secondary Structure Prediction of rRNA Molecules Using ElipSys.* 16. 12. 1991.

[ECRC–91–16] Michael Dahmen. *Abstract Debugging of Coroutines and Constraints in Prolog.* 30. 12. 1991.

[ECRC–92–1] Thierry Le Provost and Mark Wallace. *Constraint Satisfaction Over the CLP Scheme.* 30. 1. 1992.

[ECRC–92–2] Gérard Comyn, M. Jarke, and Suryanarayana M. Sripada. *Proceedings of the 1st Compulog Net meeting on Knowledge Bases (CNKBS'92).* 30. 1. 1992.

[ECRC–92–3] Jesper Larsson Traeff and Steven David Prestwich. *Meta-programming for reordering Literals in Deductive Databases.* 30. 1. 1992.

[ECRC–92–4] Beat Wüthrich. *Update Realizations Drawn from Knowledge Base Schemas and Executed by Dialog.* 4. 2. 1992.

[ECRC–92–5] Lone Leth. *A New Direction in Functions as Processes.* 25. 2. 1992.

[ECRC–92–6] Steven David Prestwich. *The PADDY Partial Deduction System.* 23. 3. 1992.

[ECRC–92–7] Andrei Voronkov. *Extracting Higher Order Functions from First Order Proofs.* 23. 3. 1992.

[ECRC–92–8] Andrei Voronkov. *On Computability by Logic Programs.* 23. 3. 1992.

[ECRC–92–9] Beat Wüthrich. *Towards Probabilistic Knowledge Bases.* 02. 4. 1992.

[ECRC–92–10] Petra Bayer. *Update Propagation for Integrity Checking, Materialized View Maintenance and Production Rule Triggering.* 08. 4. 1992.

[ECRC–92–11] Mireille Ducassé. *Abstract views of Prolog executions in Opium.* 15. 4. 1992.

[ECRC–92–12] Alexandre Lefebvre. *Towards an Efficient Evaluation of Recursive Aggregates in Deductive Databases.* 30. 4. 1992.

[ECRC–92–13] Udo W. Lipeck and Rainer Manthey (Hrsg.). *Kurzfassungen des 4. GI-Workshops "Grundlagen von Datenbanken", Barsinghausen, 9.-12.6.1992.* 12. 05. 1992.

[ECRC–92–14] Lone Leth and Bent Thomsen. *Some Facile Chemistry.* 26. 05. 1992.

[ECRC–92–15] Jacques Noyé (Ed.). *Proceedings of the International KCM User Group Meeting,Munich, 7 and 8 October 1991.* 03. 06. 1992.

[ECRC–92–16] Frederick Knabe. *A Distributed Protocol for Channel-Based Communication with Choice.* 10. 06. 1992.

[ECRC–92-17] Benoit Baurens, Petra Bayer, Luis Hermosilla, and Andrea Sikeler. *Publication Management: A Requirements Analysis.* 03. 07. 1992.

[ECRC–92-18] Thom Frühwirth. *Constraint Simplification Rules.* 28. 07. 1992.

[ECRC–92-19] Mark Wallace. *Compiling Integrity Checking into Update Procedures.* 29. 07. 1992.

[ECRC–92-20] Petra Bayer. *Data and Knowledge for Medical Applications: A Case Study.* 30. 07. 1992.

[ECRC–92-21] Michel Dorochevsky and André Véron. *Binding Techniques and Garbage Collection for OR-Parallel CLP Systems.* 11. 08. 1992.

[ECRC–92-22] Shan-Wen Yan. *Efficiently Estimating Relative Grain Size for Logic Programs on Basis of Abstract Interpretation.* 25. 08. 1992.

[ECRC–92-23] Jean-Marc Andreoli, Paolo Ciancarini, and Remo Pareschi. *Interaction Abstract Machines.* 25. 08. 1992.

[ECRC–92-24] Jean-Marc Andreoli and Remo Pareschi. *Associative Communication and its Optimization via Abstract Interpretation.* 25. 08. 1992.

[ECRC–92-25] Jean-Marc Andreoli, Lone Leth, Remo Pareschi, and Bent Thomsen. *On the Chemistry of Broadcasting.* 25. 08. 1992.

[ECRC–92-26] Marc Bourgois, Jean-Marc Andreoli, and Remo Pareschi. *Extending Objects with Rules, Composition and Concurrency : the LO Experience.* 25. 08. 1992.

[ECRC–92-27] Benoit Dageville and Kam-Fai Wong. *SIM: A C-based SIMulation Package.* 28. 09. 1992.

[ECRC–92-28] Beat Wüthrich. *On the Efficient Distribution-free Learning of Rule Uncertainties and their Integration into Probabilistic Knowledge Bases.* 29. 09. 1992.

[ECRC–92-29] Andrei Voronkov. *Logic Programming with Bounded Quantifiers.* 29. 09. 1992.

[ECRC–92-30] Eric Monfroy. *Gröbner Bases: Strategies and Applications.* 30. 09. 1992.

[ECRC–92-31] Eric Monfroy. *Specification of Geometrical Constraints.* 30. 09. 1992.

[ECRC–92-32] Bent Thomsen, Lone Leth, and Alessandro Giacalone. *Some Issues in the Semantics of Facile Distributed Programming.* 22. 10. 1992.

[ECRC–92-33] Mireille Ducassé. *An Extendable Trace Analyser to Support Automated Debugging.* 04. 12. 1992.

[ECRC–92-34] Jorge Bocca and Luis Hermosilla. *A Preliminary Study of the Performance of MegaLog.* 20. 12. 1992.

[ECRC–93–1] Benoit Dageville and Kam-Fai Wong. *Supporting Thousands of Threads Using a Hybrid Stack Sharing Scheme.* 18. 01. 1993.

[ECRC–93–2] Steven Prestwich. *ElipSys Programming Tutorial.* 18. 01. 1993.

[ECRC–93–3] Beat Wüthrich. *Learning Probabilistic Rules.* 28. 01. 1993.

[ECRC–93–4] Eric Monfroy. *A Survey of Non-Linear Solvers.* 02. 02. 1993.

[ECRC–93–5] Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. *Constraint Logic Programming - An Informal Introduction.* 02. 02. 1993.