Integrating Propagation and Built-in Constraints

Thierry Le Provost
Mark Wallace

January 1992

Abstract

Constraint logic programming is often described as logic programming with unification replaced by constraint solving over a computation domain. There is another, very different, CLP paradigm based on constraint satisfaction, where program-defined goals can be treated as constraints and handled using propagation. This paper proposes a generalisation of propagation, which enables it to be applied on arbitrary computation domains, thereby restoring orthogonality and bridging the gap between two important constraint logic programming paradigms. The main idea behind generalised propagation is to use whatever constraints are available over the computation domain to express restrictions on problem variables. Generalised propagation on a goal G requires that the system extracts a constraint approximating all the answers to G. The paper introduces a generic algorithm for generalised propagation called "topological branch and bound" which avoids enumerating all the answers to G. Generalised propagation over the Herbrand universe has been implemented in a system called *Propia*, and we describe its behaviour on some applications.

1 Background and Motivation

1.1 Motivation

Although classical logic programming is a convenient vehicle for *stating* combinatorial problems, it can be grossly inefficient for *solving* these problems. The essential reason for this inefficiency is the naivety of Prolog's computation procedure, based solely on resolution.

To get an efficient solution with standard Prolog it is therefore often necessary explicitly to program adequate algorithms for the problems at hand. One obvious drawback of such an approach is that the explicit algorithm has little in common with the initial declarative problem specification, and hence requires a major development effort from the programmer. An even more severe drawback is that the resulting program may no longer correctly solve the problem.

To avoid these drawbacks, it is therefore desirable to try and provide built-in control mechanisms that would allow a declarative problem statement to be executed efficiently.

Historically, the first step in that direction was to introduce so-called dynamic computation rules into Prolog. The idea is to allow programmers to alter the standard left-to-right goal selection mechanism of Prolog, so that goals with a beneficial effect on search space size are selected for resolution first. Control declarations typically take the form of metalogical conditions associated to goals (Prolog-II's *freeze*) or to predicate definitions (SEPIA's *delay*, MU-Prolog's *wait*, NU-Prolog's *when*), which prescribe when goals may be selected for resolution.

Unfortunately, dynamic computation rules are limited in scope, as they only affect the order of goal selection; there are many hard search problems for which no good goal ordering exists, as the search space will anyhow remain unacceptably large. Besides, control declarations can become quite unwieldy to design, debug, and reason about. Small alterations of the problem's definition often require an extensive revision of dynamic declarations.

A more recent, orthogonal approach is to build certain goals ("constraints") into the logic programming language, so that the system knows about the satisfiability of such goals without any effort by the programmer. Languages in the CLP Scheme [JL87] and CHIP's finite domain constraints are examples of this approach.

The shortcoming of such built-in approaches to search space reduction is precisely that they are built in. One can of course define new predicates using such hardwired constraints, but these predicates will be processed by the usual blind resolution mechanism. If a particular application needs a particular constraint, then it must be compiled into the system.

CHIP therefore offers a facility to support program-defined constraints. It is termed *propagation* and is provided via two special inference rules called `forward checking` and `lookahead`. Propagation enables program-defined predicates to be used as constraints in a very specific way: for reducing the domains associated with one or two variables.

"Generalised propagation" is a generalisation of CHIP's propagation that overcomes its limited expressive power. The idea is to reduce search space size while preserving the declarativeness of problem statement. Generalised propagation provides a clean and simple means of using declarative predicate definitions as deterministic reasoning agents.

1.2 Declaratively Specified Constraints

The main idea of generalised propagation is that declarative predicate definitions can be given two complementary meanings.

The first meaning is the traditional one, that is, predicate definitions denote relations over objects in some computation domain. In the case of Prolog, the computation domain is the Herbrand Universe (finite trees), but other languages in the CLP Scheme work on less trivial objects (rational trees, lists, integers, reals, Booleans, etc).

The second, new meaning attributed to predicate definitions is that they define deterministic "refinement operators" over variable bindings. Informally, a predicate holds for a (possibly infinite) set of values, and all these values might have some common properties. (For example they might all be integers greater than 20, or they might all be compound terms of the form *book*(_)). One can therefore validly extract these common properties and add them to the current environment, without losing any potential solutions to the problem at hand.

Making common properties explicit helps reduce the search space, as they may preclude some wrong guesses by the resolution procedure. Put another way, a system that features a way of extracting properties common to all solutions to goals turns goals into active constraints.

In contrast to the other approaches to introducing constraints outlined above, generalised propagation constraints are:

- program-defined, in that *any* predicate definition can be declared to be a constraint;
- declaratively specified, in that the programmer need not describe how constraints operate, but only what relation they enforce.

1.3 A few examples

Let us take a "standard" constraint satisfaction problem, which just consists of a conjunction of goals whose predicate definitions are extensional collections of Prolog facts.

```
% Some problem to be solved:
?- p(X,f(Y)), q(Y), r(X,Y,Z), ...

% The declarative definition for p/2:
p(a,f(a)).
p(b,c).
p(b,f(b)).
p(X,g(h(X))).

% Other definitions for q/1, r/3, ...:
...
```

To use `p(X,f(Y))` and `r(X,Y,Z)` as constraints there is literally nothing to do, except annotate the goal as follows:
`?- constraint p(X,f(Y)), q(Y), constraint r(X,Y,Z), ...`

Generalised propagation will take any of the constraints (say, `p(X,f(Y))`), and deterministically extract all common properties of the goal's solutions that are expressible as a substitution. Here, it will infer that $X = Y$ must hold in any solution to the problem. Knowledge that $X = Y$ may in turn allow some further deterministic inferences, using the definition for $r/3$, and so on. When this propagation process stops, control is given back to the Prolog engine to perform a resolution step. The new query will often be more explicit, e.g. more variables in it will have received a value, and some doomed resolution steps may therefore be avoided.

Declaratively specified constraints are not restricted to extensional definitions, as for instance:

```
member(X,[X|_]).
member(X,[_|T]) :-
  member(X,T).
```

is a perfectly acceptable constraint specification, which allows the system to infer $X = a$ from the goal `member(a,[b,f(Y),X,c])`.

Such constraints are not confined to reasoning on terms either, as the extracted common properties are just whatever the language's basic computation domain affords. (Hence the name "generalised propagation", as it generalises over some more restricted, similar notions that are available in the Chip system.)

For instance, if the host language allows finite-domain variables, then common properties include domain reductions in addition to bindings. If hardwired symbolic or arithmetic constraints are available in the language, then they can be combined with declaratively specified constraints.

One demonstration of this is the possibility to employ disjunctively defined constraints. Such constraints typically arise in scheduling problems; due to resource usage exclusion, two given tasks may not be performed at the same time:

```
% Either Task 1 occurs before Task 2:

disj(Start1,Duration1, Start2,Duration2) :-
  Start2 >= Start1 + Duration1.

% Or Task 2 occurs before Task 1:

disj(Start1,Duration1, Start2,Duration2) :-
  Start1 >= Start2 + Duration2.
```

Chip does not allow a $disj/4$ goal to be used as a constraint, i.e. it must make a guess between the two alternatives when a $disj/4$ goal is applied (leading to a potential combinatorial explosion). Generalised propagation allows the programmer to demand that a $disj/4$ goal be used as a constraint; and domain reductions for the four arguments then occur deterministically without committing to either alternative.

1.4 Applications

Within the CHIC project a number of applications for generalised propagation have already emerged. The first, and simplest, was stated by ICL at a CHIC user group meeting at ECRC. ICL needs to test if a constraint is satisfiable, without adding it to the environment. This is a direct application of generalised propagation using the *consistency* propagation language (as described in section 5.1 below).

The second application was stated by OCT at the same CHIC user group meeting. For propagating constraints on traffic flows it is necessary to use the *flow* predicate as a constraint, whose definition is:

```
flow(1,Flow) :- 0<Flow, Flow<0.3
flow(2,Flow) :- 0.3<Flow, Flow<0.7
```

...
Thus given the goal `?- flow(D,F)` and the constraint $D \leq 2$, the system should deduce that $0 \leq F \leq 0.7$. On the other hand, given the constraint $0 \leq F \leq 0.5$ the system should be able to deduce that $D \leq 2$. To obtain this behaviour using generalised propagation over inequalities it suffices simply to annotate the goal as a constraint, viz: `?- constraint flow(D,F)`.

The third application is more than just a single application but a very general problem in scheduling: how to deal with disjunctive constraints. The example of $disj/4$ above shows how disjunctive constraints are treated in a quite natural way using generalised propagation.

The way these problems are currently solved is by using special built-in constraints (like the *extended element* constraint recently added to CHARME), and by encoding the problem using an algorithmic understanding of constraint propagation to obtain roughly the required behaviour (as described by Dassault at the CHIC user group meeting in Valencia).

However each problem is naturally expressed in standard Prolog, and with generalised propagation the required constraint behaviour can be obtained by simply annotating the relevant goal as a constraint. No new built-in constraints need be compiled into the system.

The advocated programming methodology using generalised propagation is first to design a declarative Prolog program *stating* the problem without any concern for efficiency. Then, this probably very inefficient program can be refined by earmarking appropriate goals as declarative constraints. Of course, determining which goals are appropriate for use as constraints is the program designer's responsibility!

Programming with declaratively specified constraints is therefore still more of an art than an exact science ... but in opposition to other approaches to programming with constraints, the required trial-and-error process of identifying beneficial constraints can be performed without altering the program's structure, designing complex metalogical declarations, or implementing new language primitives.

2 Introduction

2.1 The CLP Scheme

Constraint logic programming is often described as logic programming with unification replaced by constraint solving over a computation domain. This is captured in a theoretical framework called the CLP scheme [JL87]. A $CLP(X)$ program comprises rules of the form

$$h \leftarrow c_1, \ldots c_n, b_1, \ldots b_m$$

where the c_i are constraints over the domain X and the b_j are (user-defined or built-in) logic programming goals. During computation clauses are unfolded, and the constraints in their bodies are collected up and tested for consistency. In this paper we shall often refer to constraints in the $CLP(X)$ framework as "basic constraints". One point to note is that the basic constraint predicates are built-into the system, and cannot be defined by program clauses. A second point is that the consistency check covers all the basic constraints which have been collected up during the computation (which distinguishes constraints from ordinary built-in predicates [Mah87]). This check must, in theory, be effective.

2.2 CSP in Logic Programming

There is another, very different, CLP paradigm which is based on constraint satisfaction techniques dating back to Golomb [GB65]. In the constraint satisfaction problem (CSP) paradigm the constraints are problem-specific, and defined by sets of tuples. When CSP is embedded into logic programming, the definition of a constraint can be defined in the program as a set of facts, or even as a set of rules [Van89]. We shall often refer to constraints in the CSP framework as "propagation constraints".

One prerequisite for applying CSP techniques is that problem variables should have an associated domain of possible values. Traditionally [Fik70, Mon74, Mac77, HE80] this is a finite domain, though more recently continuous intervals have been studied [Dav87]. Up to now, constraint logic programming systems based on the CSP paradigm (eg CHIP [DVS+88]) have therefore had to include domain declarations for the problem variables. This is one restriction we will relax using generalised propagation.

It should be noted that constraint solving over a computation domain is replaced in this paradigm by constraint propagation over value domains [Fik70, Mon74, Mac77, HE80]. Informally constraint propagation operates by looking ahead at yet unsolved goals to see what locally consistent valuations there remain for individual problem variables. In the CSP framework there is no guarantee that, after a complete propagation sequence, the propagation constraints are globally consistent, by contrast with constraint solving for basic constraints in the CLP scheme. However such propagation techniques can have a dramatic effect in cutting down the size of the search space. Evidence of the practical effectiveness of constraints propagation in logic programming is given in [DSV90].

2.3 Propagation in Finite Domains

To date the technique of propagation in constraint logic programming has only been defined for finite domain variables. Each such variable can only take a finite number of values, and looking ahead is a way of deterministically ruling out certain locally inconsistent values and thus reducing the domains. This restriction has prevented the application of propagation to new computation domains introduced by the CLP scheme and related approaches. In addition propagation as currently defined only exploits a fraction of the power of its native universe of discourse. For instance it cannot reason on compound terms, thereby enforcing an unnatural and potentially inefficient encoding of structured data as collections of constants.

This has meant that the two approaches to integrating constraints into logic programming, as *basic* constraints and as *propagation* constraints, have had to remain quite separate. Even in the CHIP system [DVS$^+$88] which utilises both types of integration, propagation is excluded from those parts of the programs involving new computation domains, such as Boolean algebra or linear rational arithmetic.

2.4 Generalised Propagation

This paper proposes a generalisation of propagation, which enables it to be applied on arbitrary computation domains. Generalised propagation can be applied in $CLP(X)$ programs, whatever the domain X. We shall call $GP(X)$ the system applying generalised propagation in $CLP(X)$. Finite domain propagation in logic programming is just $GP(FD)$.

The basic concepts, theoretical foundations, and abstract operational semantics of $GP(X)$ can be defined independently of the computation domain, X. This allows programmers to reason about the efficiency of $GP(X)$ in an intuitive and uniform way. This generality carries over to the implementation, where algorithms for executing generalised propagation apply across a large range of basic constraint theories. Last but not least, the declarative semantics of $CLP(X)$ programs is preserved in $GP(X)$.

We briefly motivate generalised propagation with a simple example. Constraint logic programming has been applied to the problem of crossword compilation [Van89]. A domain variable was associated with each blank word in the crossword, whose domain was the set of words in the lexicon that could fit there. The correct choice of words was enforced by constraints on their intersections. The finite domains associated with the variables had around 30 possible words. A more natural representation of the problem, where variables denote letters ranging over the alphabet, unfortunately fails to allow sufficient constraint propagation to enable the problem to be solved in any reasonable time.

With generalised propagation this representation of the problem is possible. It is stated as follows:

```
top :-
    constraint word(A1,A2,B1,A4,C1),
    constraint word(B1,B2,B3,B4),
    constraint word(C1,C2,C3),
    ...
```

where the intersections are reflected as shared letters in distinct word goals. Instead of propagating domain reductions, the system propagates equalities. Not only does this allow the original problem, with a lexicon of 150 words, to be solved, it enables the problem to be scaled up to realistic proportions. Using generalised propagation, this program compiles crosswords from a lexicon of 25000 words. A more detailed discussion of this application follows in section

5.1. The main idea behind $GP(X)$ is to use whatever constraints are available over the computation domain X to express restrictions on problem variables. (Associating finite domains with variables is one specific application of this concept.) Goals designated as propagation constraints are repeatedly approximated as closely as possible using these constraints. When no further refinement of the current resolvent's approximation is feasible, a resolution step is performed and propagation starts again.

The practical relevance of generalised propagation has been tested by implementing it in the underlying constraint theory of first-order terms with syntactic equality [Cla79], which is the computation domain of Prolog. This is $GP(HU)$. Programs are just sets of Prolog rules with annotations identifying the goals to be used for propagation. The language has enabled us to write programs which are simple, yet efficient, without the need to resort to constructs without a clear declarative semantics such as demons. The performance results have been very encouraging.

In the next section we shall describe finite domain propagation in logic programming, then in section 3 we shall specify generalised propagation, discussing its logical and operational semantics and introducing a generic algorithm for its implementation over arbitrary computation domains. In section 4 we shall describe two implemented instances of generalised propagation, based on a single system called *Propia*. In section 5 we shall compare generalised propagation with some related approaches, and we shall conclude in section 6.

3 Constraint Propagation

In this section we briefly review the motivation of finite domain propagation in logic programming and describe its behaviour with some examples.

3.1 Passive and Active Constraints

For solving CSP problems in logic programming, backtrack search is used. The aim is to perform relevant "tests" as soon as possible after instantiating a variable. Sophisticated dynamic computation rules, such as *freeze* [Col85] and *delay* [Nai86, MAC+89] can be used to determine which goal to evaluate next. However even such dynamic rules can only postpone evaluation until the constraints are partially or fully instantiated. Evaluating partially instantiated constraints will generate values for variables, usually creating undesirable branches in the search tree. Waiting till the constraint is ground before evaluating, is to use the constraint as an a posteriori test. To summarise, logic programs can only use constraints *passively*.

Our motivation for constraints logic programming is to support the *active* use of constraints [Gal85]. This is provided by techniques developed for solving constraint satisfaction problems (CSP). The study of CSP has a long history, and we mention here just a few important references. The concept of arc consistency was introduced in [Mac77, Fre78]; its combination with backtrack search was described in [HE80]; the notion of value propagation is due to [SS80]; the application of constraint methods to real arithmetic was surveyed in [Dav87]; finally [Van89] extensively motivates and describes in detail the integration of finite-domain propagation methods into logic programming.

3.2 Propagation in Constraint Logic Programming

The idea behind local propagation methods for CSP's is to work on each propagation constraint independently, and deterministically to extract information about locally consistent assignments. This has lead to various consistency algorithms for networks of constraints, the most widely applicable of these being arc-consistency [RHZ75, Mon74]. Consistency can be applied as a preliminary to the search steps or interleaved with them [HE80]. The application of these techniques in logic programming was accomplished through two complementary extensions [VD86, Van89]

- explicit *finite domains* of values to allow the expression of range restrictions, together with the corresponding extension of unification (FD-resolution)
- new *inference rules*, based on looking ahead at "future" computations, to reduce finite domains in a deterministic way

The effect of looking ahead on a goal is to reduce the domains associated with the variables in the goal, so that the resulting domains approximate *as closely as possible* the set of remaining solutions. Application of these inference rules is repeated on all propagation constraint goals until no more domain reductions are possible, forming a *propagation sequence*. Propagation constraint goals that are satisfied by any combination of values in the domains of their arguments can now be dropped.

One algorithm for implementing lookahead is to enumerate all combinations of values for the constraint's arguments and check each combination by calling the goal instantiated with these values. The reduced domains are then formed by projecting successful combinations onto each argument. CHIP in addition implements a variety of predefined constraint predicates, which efficiently perform domain reduction by specialised algorithms. (The drawback of such dedicated algorithms is that they cannot be applied to program-defined predicates.) An example problem encoded in a CHIP-like syntax follows:

```
csp(X1,X2,X3,X4) :-
    [X1,X2,X3,X4] :: [a,b,c],
    constraint p(X3,X1),  /* 1 */
    constraint p(X2,X3),  /* 2 */
    constraint p(X2,X4),  /* 3 */
    constraint p(X3,X4).  /* 4 */

p(a,b).    p(a,c).    p(b,c).
```

The `constraint` annotations identify goals that must be treated by the new inference rule. Annotations can be ignored for a declarative reading.

For our example problem, the initial propagation sequence is sufficient to produce the only solution. A possible computation sequence is as follows (though the ordering is immaterial for the final result):

```
constraint p(X3,X1)        [1]  produces  X3::{a,b}, X1::{b,c}
constraint p(X2,X3::{a,b}) [2]  produces  X2=a, X3=b
constraint p(a,X4)         [3]  produces  X4::{b,c}
constraint p(b,X4::{b,c})  [4]  produces  X4=c
constraint p(b,X1::{b,c})  [1]  produces  X1=c
constraint p(a,b)          [2]  succeeds
constraint p(a,c)          [3]  succeeds
constraint p(b,c)          [4]  succeeds
```

Note that the propagation constraint [1] takes part in two propagation steps before it is solved. In general constraints may be involved in any number (> 0) of propagation steps.

This example is deliberately very simple. Normally an answer is not obtained by propagation alone. If a propagation sequence terminates without producing an answer, then variables are instantiated non-deterministically to values in their domains: this can be achieved by an explicit "labelling" routine (as in CHIP) or by an implicit labelling performed automatically by the system.

3.3 Clauses Defining Propagation Constraints

A nice property of constraint logic programming is the fine level of control it offers over problem solving. A propagation constraint goal can be defined by rules and therefore can be arbitrarily complex. As an example consider a CSP problem comprising propagation constraints $C_1, \ldots, C_n$. As a constraint logic program this could be expressed as

$top \leftarrow$
    /* domain declarations */
    constraint C_1,
    ...
    constraint C_n

Assuming that the propagation constraints are defined by immediately accessible facts, independent propagation on the sequence of constraints $C_1, \ldots C_n$ can be relatively efficient. CHIP for example uses the arc consistency algorithm of [MF85], which is $o(n.d^3)$ where d is the size of the largest variable domain. In fact this complexity result holds for constraints on two variables. When more variables are linked in a constraint, different algorithms may be more efficient.

Another way to encode the same problem in CLP is as follows:

```
bigConstraint ←
                $C_1$,
                ...
                $C_n$
top ←
    /* domain declarations */
    constraint bigConstraint
```

In this case, by contrast, to find a single tuple for *bigConstraint* is to solve the whole CSP problem. Its worst case complexity is therefore exponential in the size of the largest domain.

Clearly it brings nothing to define the *whole* problem as a single propagation constraint. However the facility to combine clusters of constraints into a single larger constraint means that propagation can be used to enforce consistency just as local or global as necessary for the problem at hand. The only practical necessity is to treat efficiently constraints involving a number of variables. Indeed there is no reason why a predicate cannot include constraints in its definition, whilst itself appearing in a constraint goal in a larger program. Generalised propagation provides a framework where local and global propagation are practical alternatives.

4 A Specification of $GP(X)$

4.1 Definitions

The language syntax and semantics used in this paper are based on first order logic. Atomic formulae are built from variables, predicate symbols, function symbols and constant symbols. If Φ is any open formula, then $\exists\Phi$ and $\forall\Phi$ denote respectively the existential and universal closure of Φ as usual.

The predicate symbols are divided into *interpreted* predicates and *user* predicates (often called "uninterpreted" predicates). The function and constant symbols are divided similarly.

For a given computation domain X, the interpreted symbols have a predefined interpretation, independent of the programs in which they appear. The $=$ predicate symbol is always an interpreted predicate, interpreted as equality in the underlying domain. Two further predicates which are always interpreted are $true$ and $false$. Typically interpreted function symbols include $+$ and $-$ with their usual interpretation.

By contrast the interpretation of the user predicates is dictated by the program semantics. User functions and constants have the free interpretation in the underlying domain.

An atom containing only interpreted predicates, functions and constants is termed an *interpreted constraint*. An atom containing only user predicates, functions and constants is termed a *user* atom. An atom cannot contain both interpreted and user symbols.[1]

We admit an additional syntax for atoms *constraint* A where A is a user atom. This syntax yields another kind of constraints called *propagation* constraints. Unlike interpreted constraints, propagation constraints have user predicates whose interpretation is dictated by the program semantics.

We now further distinguish two classes of interpreted constraints. These are the *basic* constraints and *approximation* constraints. The conjunction of a set of basic or approximation constraints is also termed an basic or approximation constraint respectively. Approximation constraints "approximate" basic constraints in the sense that for any approximation constraint AC there are basic constraints C such that $X \models (C \rightarrow AC)$. An example of an approximation constraint is $X :: \{1, 2, 3\}$ which states that either $X = 1$ or $X = 2$ or $X = 3$. It approximates each of the basic equality constraints $X = 1$, $X = 2$ and $X = 3$. Approximation constraints are a generalisation of Davis' *labels* [Dav87]. The approximation constraints and

[1] In practise one can admit such atoms (e.g. $p(1 + X)$) and view them as abbreviations for a conjunction where the equalities are made explicit (e.g. $p(Y) \wedge Y = (1 + X)$).

the basic constraints need not be disjoint: in other words basic constraints could approximate themselves.

A $GP(X)$ program is a set of clauses of the form $Head \leftarrow Body_1, \ldots, Body_s$. The head $Head$ is a user atom. The body is a set of atoms, which may include user atoms and constraints. A clause with an empty head $\leftarrow Body_1, \ldots, Body_s$ is termed a goal.

An example clause is

$$\begin{aligned}
&profit(Company, P) \leftarrow \\
&\qquad\qquad \text{constraint } public(Company), \\
&\qquad\qquad income(Company, I), \\
&\qquad\qquad expenditure(Company, E), \\
&\qquad\qquad P = I - E
\end{aligned}$$

4.2 Logical and Operational Semantics for $GP(X)$

A Framework for Evaluation in $GP(X)$ The framework for evaluation in $GP(X)$ is based on that of Jaffar and Lassez [JL87], extended with the concept of a constraint store and the *tell* operation by Saraswat [Sar89].

An evaluation in $GP(X)$ involves at any time a current goal and a current constraint store. The constraint store contains a conjunction of interpreted constraints.[2]

Declarative Semantics for $GP(X)$ Logically a propagation constraint *constraint* A is equivalent to the user atom A. A clause
$Head \leftarrow Body_1, \ldots, Body_s$ with free variables $X_1, \ldots X_t$ as usual denotes the formula
$\forall X_1, \ldots X_t.(Head \vee \neg Body_1 \vee \ldots \vee \neg Body_s)$.
The meaning of a program is given by the conjunction of its clauses. The denotation of a goal $\leftarrow Body_1, \ldots, Body_s$, is the positive formula $Body_1 \vee \ldots \vee Body_s$.

An answer is a basic constraint which, when added to the constraint store, implies the truth of the goal:

Definition 1 *For a given $GP(X)$ program P, the evaluation of a goal G with current constraint store S is a basic constraint C such that $X \models \exists.(S \wedge C)$ and $X \models P \rightarrow \forall.((S \wedge C) \rightarrow G)$*

When it is clear from the context, instead of $X \models P \rightarrow \Phi$ we shall write $\models \Phi$. Thus we can write that every answer C to a goal G with store S satisfies $\models \forall.((S \wedge C) \rightarrow G)$.

Operational Semantics for $GP(X)$ A query Q is evaluated against a $GP(X)$ program by initialising the goal to $\leftarrow Q$ and starting with an empty constraint store. In general a goal may contain both atomic constraints and user atoms.

As in $CLP(X)$, a goal is evaluated by reducing user atoms to constraints and adding the constraints to the constraint store. Each atom in the goal is selected and processed in turn. However a user atom can only be selected if there are no constraints left in the goal.

As in $CLP(X)$, a user atom A is processed by non-determistically selecting a clause $H \leftarrow B_1, \ldots B_n$ whose head matches A. The atom A is then replaced in the goal by the set $\{H = A, B_1 \ldots B_n\}$. Each alternative clause selection defines an alternative branch in the evaluation.

When a constraint is selected it is first removed from the goal. Then information is extracted from it and "told" to the constraint store. The *tell* operation checks that the new information is consistent with the store, and if so the information is added to the store. If the new information is inconsistent with the store, the *tell* operation fails.

As in $CLP(X)$, interpreted constraints are treated by "telling" them immediately to the constraint store.

The special feature of $GP(X)$ is the handling of propagation constraints using generalised propagation. Generalised propagation can be seen as an example of the *relaxed tell* operation of [HD91] which is discussed in more detail in section 6.3, below. Approximation constraints are extracted from the propagation constraint and they are "told" to the store. However

[2] In [SKL90] each store is viewed as a set. However this second view can only treat distinct tokens of information which are logically independent.

propagation is a continuing process which goes on until an answer to the propagation constraint has been found. In general propagation will continue concurrently with the handling of other constraints and user atoms in the goal.

We now define the computed answer returned by an evaluation:

Definition 2 *For a given $GP(X)$ program P, the evaluation of a goal G with current constraint store S terminates when the goal is finally empty and the propagations are all complete. The conjunction of basic constraints added to the constraint store during the evaluation is termed a "computed answer" to G with S.*

An answer to a query is a special case of an answer to a goal with a constraint store:

Definition 3 *An answer, or computed answer, to a query Q is an answer, or respectively a computed answer, to $\leftarrow Q$ with the empty constraint store.*

For soundness we require that all the computed answers are answers. For completeness we cannot require that all answers are also computed answers, but rather we require the computed answers to represent all the answers in the sense that they generate the same set of solutions.

This requirement is expressed as follows (see [Smo91]):

Definition 4 *Over the computation domain X, a set of computed answers R represents the set of answers S if for every answer $s \in S$, and every variable valuation α such that $X \models_\alpha s$, there is a computed answer $r \in R$ such that $X \models_\alpha r$.*

If there are no propagation constraints, then our framework reduces to $CLP(X)$. In this case, as long as the choice of goals is fair, i.e. in any infinite branch every goal is selected sooner or later, then sound and complete sets of answers are returned. The only requirement is that the test for satisfiability of basic constraints be effective (correct and terminating). The special requirements on the domain of computation introduced in [JL87] are not necessary here since we are only interested in positive queries and programs. The proof of soundness and completeness is easily adapted from [Llo84]. A related proof is in [Smo91].

In such an operational framework, computation can be avoided by adding extra constraints to the constraint store so that unsatisfiability is detected earlier. This is the idea behind propagation. For example in finite domain propagation a domain constraint like $X = 1 \vee X = 2$ when added to the constraint store can cut short the evaluation of the goal $p(X, Y)$ using the clause $p(3, Y) \leftarrow body(Y)$, since the constraint $X = 3$ is inconsistent with it.

4.3 Propagation on a Single Constraint

Some Conditions on Approximation Constraints The information extracted from a single propagation constraint is informally the best approximation to all its answers. To make this notion formal we first introduce a partial ordering on interpreted constraints by logical implication; that is if A implies B we write $A \leq B$. Thus logically stronger formulae are below logically weaker formulae in our ordering.

We shall also impose a few conditions on the approximation constraints.

- They should include $true$ and $false$.
- The ordering on approximation constraints should be well-founded: i.e. any set of approximation constraints should have a least element. If the set is consistent, the least element should be different from $false$.
- Every consistent decreasing sequence of approximation constraints should be finite: i.e. the ordering restricted to the approximation constraints is well-founded.

The first condition will merely ensure that every propagation constraint has an approximation. The second condition will ensure that every propagation constraint has a best approximation, which is not $false$ if the propagation constraint is satisfiable. The third condition will ensure that successful propagation sequences terminate. The third condition, together with the fact that approximation constraints are closed under finite conjunction, implies the second condition.

Information Extracted by a Single Propagation Step We are now in a position to specify precisely the result of a single propagation step on a constraint.

Definition 5 *The result of a single propagation step on a propagation constraint PC with constraint store S is the smallest approximation constraint AC which is an upper bound on the set of answers to PC with S.*

The intuition behind this definition is to extract as much information as possible from the propagation constraint without excluding any answers. For example, the result of finite domain propagation on the constraint $p(X,Y)$ defined by the facts
$p(1,2)$
$p(2,1)$
$p(3,1)$
with store $X \in \{2,3\}$ is the smallest domain constraint which is implied by both $X = 2 \wedge Y = 1$ and $X = 3 \wedge Y = 1$. This is indeed $(X = 1 \vee X = 2) \wedge Y = 1$ as expected.

We now introduce a function $prop(S,C)$ which captures the effect of propagation on a constraint C with constraint store S. Specifically $prop(S,C)$ is the new constraint store which results from adding to S the result of propagation on C with S.

Definition 6 *If the result of propagation on constraint C with store S is AC, then $prop(S,C)$ is defined to be $S \wedge AC$.*

4.4 Propagation Sequences

After propagation on a number of constraints, the constraint store contains new basic constraints. Since the new store is an argument to the propagation function, yet more information can often be extracted by propagating again on the same constraints:

$$
\begin{array}{ll}
S_1 & = prop(S_0, C_1) \\
\ldots & \\
S_n & = prop(S_{n-1}, C_n) \\
S_{n+1} & = prop(S_n, C_1) \\
\ldots & \\
S_{2n} & = prop(S_{2n-1}, C_{2n})
\end{array}
$$

In order to extract as much information as possible, propagation is attempted repeatedly on all constraints until there is no more information to be extracted. Such a complete sequence of propagations is termed a "propagation sequence".

The propagation function $prop(S,C)$ adds new information to the constraint store, and thus under our ordering $prop(S,C) \leq S$. In other words the propagation function is decreasing. We require one more property of the propagation function and that is monotonicity. If there is more information in the constraint store then propagation on a given constraint should not yield less additional information.

Lemma 1 *If $S1 \leq S2$, then $prop(S1,C) \leq prop(S2,C)$.*

Proof 1 *The condition $S1 \leq S2$ implies that $(C \wedge S1) \rightarrow (C \wedge S2)$. From the definition of an answer it follows that if A is an answer to C with $S1$ then A is an answer to C with $S2$. Thus the set of answers to A in $S2$ contains the set of answers to A in $S1$. Therefore any upper bound on the answers to A with $S2$ is an upper bound on the answers to A in $S1$.*

Consequently, if $AC1$ and $AC2$ are the results of propagation on C with stores $S1$ and $S2$ respectively, $AC1 \leq AC2$. Finally $prop(S1,C) \equiv (S1 \wedge AC1) \leq (S2 \wedge AC2) \equiv prop(S2,C)$.

In our definition of $GP(X)$ clauses and goals above, we said that the ordering of atoms is immaterial. We now demonstrate that the information extracted as the result of a propagation sequence is independent of the order of constraint propagations, thus it is not necessary to reintroduce any ordering to obtain a well-defined semantics for constraint propagation.

Theorem 2 *The fixpoint reached by a propagation sequence is independent of the order in which propagation steps are performed.*

Proof 2 *Since propagation steps are monotonic and decreasing, they yield closure operators, and the theorem is a standard lattice theoretic result.*

In an implementation the system must be able to detect when a fixed point has been reached. This requires an effective decision procedure to determine if an approximation constraint is a logical consequence of the constraint store.

4.5 Propagation in a Program

Unfolding Propagation Constraints It was stated in section 4.1 above that logically a propagation constraint *constraint* A is equivalent to the user atom A. However a single propagation step or propagation sequence is not guaranteed to extract an approximation constraint equivalent to A.

As much information is extracted as can be expressed using approximation constraints, but in general there may remain further information not expressible as approximation constraints. For finite domain propagation an example is the propagation constraint $p(X, Y)$, defined by the facts
$p(1, 2)$
$p(2, 1)$
The information extracted from the constraint is that X has domain $\{1, 2\}$ and Y has domain $\{1, 2\}$. Consequently if p is defined as above and q is defined by the clauses
$q(1, 1)$
$q(2, 2)$
then the query *constraint* $p(X, Y)$, *constraint* $q(X, Y)$ will yield $X \in \{1, 2\}, Y \in \{1, 2\}$ after propagation. However the result of the query $p(X, Y), q(X, Y)$ is, of course, failure.

This example shows that for soundness of $GP(X)$ it is necessary that evaluation should not terminate until the basic constraints in the constraint store imply the truth of the user atom in the constraint.[3] We therefore say that propagation is not complete until this state is reached. In the following definition we denote by $b(S)$ the conjunction of basic constraints in the constraint store S.

Definition 7 *Propagation on a constraint* PG *is complete when the constraint store S licenses the implication:* $\models b(S) \rightarrow G$.

In the CHIP system this condition is not enforced by the system, but left to the programmer who generally adds a "labelling" routine to instantiate problem variables non-determistically to values in their current domains. The labellings adds the required basic constraints (equalities) to the constraint store.

In $GP(X)$ the soundness is enforced automatically by the system. When the goal is empty, before termination, propagation constraints for which propagation is not complete are selected to be evaluated like user atoms by unfolding them. Consequently sound and complete answers are computed as in the $CLP(X)$ framework. This is an appropriate (and automatic) generalisation of CHIP's labelling routines.

Soundness of $GP(X)$ In the following we shall view propagation constraints as "annotated" atoms, and we shall call interpreted constraints and user atoms "un-annotated" atoms. We shall call the $CLP(X)$ program that results from a $GP(X)$ program by replacing all propagation constraints *constraint* A by A, the "un-annotated" program. An un-annotated goal is defined similarly. The soundness and completeness of $GP(X)$ for an arbitrary goal and program will be proved by using the soundness and completeness of the un-annotated goal and program.

For the purposes of these proofs, any unfolding of propagation constraints in $GP(X)$ will not be distinguished from the unfolding of un-annotated user atoms.

The constraints added to the constraint store during the processing of un-annotated atoms in a $GP(X)$ goal and program are precisely those added in a $CLP(X)$ evaluation of the un-annotated goal and program. Thus the constraint store contains computed answers to all un-annotated atoms in the goal. For the propagation constraints, propagation is not complete until the constraint store implies their truth. Since the whole constraint store is consistent, the computed answers are indeed answers to the complete goal.

Completeness of $GP(X)$ If we assume that generalised propagation introduces no extra non-termination, then the completeness of generalised propagation is a consequence of the completenes of $CLP(X)$. We first prove that every success branch in the un-annotated program remains successful in the $GP(X)$ program. Then we show that the final constraint store

[3] The restriction to basic constraints is required if answers are expressed as basic constraints. Alternatively the framework could allow approximation constraints to appear in answers.

for the un-annotated program on any branch is logically stronger than that for the $GP(X)$ program on the same branch. The completeness of the $GP(X)$ program is then a direct consequence of the completeness of the un-annotated program.

The proof that every success branch for the un-annotated program remains successful in the $GP(X)$ program depends on a few simple observations. Let S be the final constraint store, PA and PB be arbitrary propagation constraints, CA and CB the complete conjunction of approximation constraints extracted from PA and PB respectively during propagation, and A and B arbitrary answers to PA and PB. Finally let C be an arbitrary constraint added to the constraint store during evaluation of the un-annotated program.

1. $\models A \rightarrow CA$ and $\models B \rightarrow CB$
2. If $S \wedge A$ is consistent then so is $S \wedge CA$
3. If $S \wedge A \wedge C$ is consistent, then so is $S \wedge CA \wedge C$
4. If $S \wedge A \wedge B$ is consistent, then so is $S \wedge CA \wedge CB$.

From 2 we conclude that the extraction of approximation constraints from a propagation constraint cannot fail if there are any consistent answers. Thus no extra failure is introduced.
From 3 we conclude that the approximation constraints extracted and held in the constraint store cannot prevent any other finally consistent constraints being added. Thus no extra failure can be caused.
From 4 we conclude that the approximation constraints held in the store cannot cause any further propagation to fail, if there are any consistent answers. Thus no extra failure is introduced.
Consequently the *tell* operation which adds constraints to the constraint store never fails on a particular branch in a $GP(X)$ program if it did not already fail on the same branch in the un-annotated $CLP(X)$ program.

To prove that the final constraint store in the un-annotated program is logically stronger than that in the $GP(X)$ program we examine the added constraints.

Firstly all the same constraints are added during the processing on un-annotated atoms. Secondly the approximation constraints extracted from the propagation constraints are logically weaker than all answers, and in particular weaker than answers computed by the un-annotated program. Therefore for every computed answer obtained from the un-annotated program there is a logically weaker computed answer obtained from the $GP(X)$ program. The completeness of the $GP(X)$ program is immediate from the completeness of $CLP(X)$.

4.6 Termination in $GP(X)$

Termination of the search for answers to a propagation constraint is not guaranteed. Non-termination due to unfolding is inherited from $CLP(X)$: in practise the programmer is responsible for ensuring that unfolding should terminate. Just as any user goal in $CLP(X)$, a propagation constraint in $GP(X)$ can only be evaluated after the clause in whose body it appears has been unfolded. In this sense $GP(X)$ is no different from $CLP(X)$.

There are two differences. Firstly all answers to a propagation constraint are generally required instead of just one as in $CLP(X)$. Of course backtracking will generally imply that many answers to a goal must be found in $CLP(X)$ as well. The theoretical problem remains that in $CLP(X)$ every answer lies at the end of a terminating success branch, whilst the requirement during propagation for *all* answers to a propagation constraint implies that *any* infinite branch in the search tree can cause non-termination of a propagation step.[4]

Secondly, a propagation constraint may be evaluated and re-evaluated many times in $GP(X)$. Luckily this does not alter the termination behaviour of the program. The reason is that on later evaluations the constraint store is logically at least as strong as before. Consequently the later evaluations may benefit from extra pruning of some branches, but no new infinite branches can arise.

Of particular concern in $GP(X)$ is the potential for propagation sequences not to terminate, even though the propagation constraints are satisfiable and unfolding would terminate in every case. A propagation sequence defines a sequence of constraint stores
$S \wedge AC_1$

[4]But it frequently does not, as we show below in section 4.7.

$S \wedge AC_1 \wedge AC_2$
$\ldots$
where no AC_n is a logical consequence of $S \wedge AC_1 \wedge \ldots \wedge AC_{n-1}$. However in this case no AC_n is a logical consequence of $AC_1 \wedge \ldots \wedge AC_{n-1}$ either, and therefore the conjunctions of approximation constraints define a descending sequence. If the constraints are satisfiable, so are all the approximations. Such a satisfiable sequence cannot be infinite due to our well-foundedness assumption (section 4.3 above). Therefore every satisfiable propagation sequence must terminate.

4.7 Topological Branch and Bound

Evaluating Propagation Constraints Conceptually the evaluation of a propagation constraint PC requires

- finding all the answers to the goal PC
- finding the smallest approximation constraint which is an upper bound for the set of answers

Lemma 3 *The upper bounds of the set of answers S are precisely the upper bounds of the set of computed answers R*

Proof 3 *If u is an upper bound of S, then since $R \subset S$, u is also an upper bound of R.*

Show that if u is an upper bound of R it is an upper bound of S. If u is not an upper bound of S, then for some answer $s \in S$ and some variable valuation α, $\models_\alpha s$ but $\models_\alpha \neg u$. However by our definition of completeness there is an $r \in R$ such that $\models_\alpha r$ and therefore u is not an upper bound for R.

Using this result, when finding the smallest approximation constraint, the system can use the set of computed answers instead of the whole set of answers.

We shall start by assuming that the computation of the computed answers terminates, and therefore the set of computed answers is finite. Notice that a finite number of computed answers in a $CLP(X)$ program may denote an infinite number of solutions. For example the propagation constraint $p(X)$, defined by
$p(X) \leftarrow X > 1$
$p(X) \leftarrow X < 1$
has a finite number (two) of computed answers, though it has an infinite number of solutions over the domain of integers.

For $GP(X)$ three built-in procedures are required.

- For finding answers, the system must support an effective decision procedure for basic constraints over X (the same procedure is required for $CLP(X)$)
- For extracting approximations, the system must additionally support an effective procedure for producing the smallest approximation constraint which is an upper bound for a finite set of basic constraints.
- In section 4.4 above another effective decision procedure was mentioned, to determine if an approximation constraint is a logical consequence of the current store. This is necessary to enable propagation sequences to terminate.

Interleaving Answering and Approximation Practically the evaluation of propagation constraints interleaves the finding of individual answers and their generalisation. To make this possible we assume that our procedure for extracting approximations can return the smallest approximation constraint which is an upper bound for a basic constraint and an approximation constraint. To approximate a finite set of computed answers it is now possible perform the approximations pairwise.

Lemma 4 *If $A2$ is the best approximation of $\{D1, D2\}$ and $A3$ is the best approximation of $\{A2, D3\}$, then $A3$ is the best approximation of $\{D1, D2, D3\}$.*

Proof 4 *Suppose AC is the best approximation of $\{D1, D2, D3\}$, then AC is an upper bound for $\{D1, D2\}$.*

Consequently $A2 \rightarrow AC$. However we also know that $D3 \rightarrow AC$. and therefore AC is an upper bound for $\{A2, D3\}$. However we assumed $A3$ was the best approximation for $\{A2, D3\}$, therefore $AC \equiv A3$.

This lemma generalises to finite sets of answers by induction.

Cutting All Remaining Branches We now describe two optimisations which fit naturally into the evaluation of propagation constraints. Both optimisations depend upon the interleaving of answering and approximation. At any point in the evaluation of a propagation constraint the system has available

- the constraint store S
- the current approximation constraint AC which is the smallest approximation constraint which is an upper bound for the answers found so far

The current approximation constraint can be used just like the current best cost in a branch and bound search. However it can also be used, in a way not available in branch and bound, to prune off all the remaining branches of the search tree.

Using the procedure which decides if an approximation constraint is implied by the constraint store, it is possible to prune the evaluation of a propagation constraint by

- interleaving the finding of answers and generating new approximation constraints AC
- terminating the computation as soon as the current approximation constraint is implied by the constraint store S, i.e. $S \rightarrow AC$

This optimisation is very important for propagation constraints defined by large numbers of clauses. For such constraints it is often easy to find a few solutions, but very expensive to find them all. Its significance is illustrated by the crossword compilation application described below 5.1.

Cutting off the Current Branch When exploring a single branch the system collects locally a set of basic constraints extracted during the unfolding of clauses. The conjunction of all the basic constraints extracted along a branch goes to make up a single answer to the propagation constraint. If this answer is logically stronger than the current approximation constraint (which approximates all the answers found so far), then it cannot affect the final result.

Branch and bound search benefits from the observation that there is no need to explore to the end a branch that is already more expensive than the current best branch. In evaluating a propagation constraint the same observation applies: there is no need to explore further if the local constraints gathered on a branch are already logically stronger than the current approximation constraint.

The required decision procedure is the same as before: we need to determine if the current approximation constraint is implied by a set of basic constraints.

This optimisation proves to be very valuable for propagation constraints defined by recursive clauses. This will be illustrated using the *member* predicate in section 5.2 below.

We can summarise the procedure for evaluating a propagation constraint *constraint* G with constraint store S as follows:

> After each answer A to the goal G is retrieved it is first checked for consistency with the constraint store S. If $S \wedge A$ is unsatisfiable, then the answer is thrown away. If no consistent answers are found, then constraint propagation has detected an inconsistency, and the propagation sequence terminates signalling inconsistency.
>
> The initial approximation constraint AC is set to *false*. When a consistent answer A is found it is added to the current best approximation, and the pair $\{AC, A\}$ is approximated yielding a new approximation constraint. AC is set to the new constraint.
>
> During the search for an answer, basic constraints are added to a local constraint store, LS. If at any stage $LS \rightarrow AC$, then the local search is abandoned. Search for new answers continues by choosing other clauses to unfold.
>
> Propagation terminates as soon as the approximation constraint is implied by the constraint store, $S \rightarrow AC$. In this case no new information could be extracted,

and so $prop(S, G) = S$. Otherwise propagation terminates when there are no further alternative clauses to unfold. Then the current approximation constraint is added to the constraint store, and so $prop(S, G) = (S \wedge AC)$.

Evaluating Propagation Sequences In the case of finite domain propagation, the procedure for performing propagation on a single constraint is called $REVISE$ [MF85]. Essentially the evaluation of a propagation sequence for generalised propagation can be obtained from the AC-3 algorithm by replacing $REVISE$ with topological branch and bound.

A feature of AC-3 is that after propagating on a constraint C, C is removed from the list of constraints to be dealt with in the current propagation sequence. C is only added to the list again if some of its variables are affected by propagation on other constraints. For the correctness of AC-3 it is therefore necessary that propagation on a single constraint is itself a fixpoint operation. This can be stated as a simple lemma:

Lemma 5 *For any constraint store S and propagation constraint C, $prop(S, C) = prop(prop(S, C), C)$*

Proof 5 *Let AC be the result of propagation on constraint C with store S. By definition, $prop(S, C) \equiv S \wedge AC$. AC approximates every answer A to C with S, so $A \rightarrow AC$. Since every A is consistent with S, it follows that A is consistent with $S \wedge AC$. Moreover if $(S \wedge A) \rightarrow C$, then a fortiori $(S \wedge AC \wedge A) \rightarrow C$. Consequently every answer to C with S is also an answer to C with $prop(S, C)$. Therefore the result of propagation on C with $prop(S, C)$ remains $prop(S, C)$.*

This condition is not satisfied by *relaxed tell* [HD91], which is an abstraction of generalised propagation (see below 6.3).

5 Some Instances of GP(X)

Two implementations of generalised propagation over the Herbrand universe have been completed. In the two following sections the examples we describe have all been run on a GP(HU) implementation called "Propia". Propia extracts information about equalities from propagation constraints, and it offers a number of approximation languages some of which will be described below. Propia is implemented in Sepia [MAC$^+$89] with the help of some special added built-ins.

5.1 GP(Datalog)

Datalog is logic programming without functions. The basic constraints in Datalog are equalities, $X = c$ or $X = Y$ where c is a constant and X and Y are variables. This means that the number of ground answers to an n-ary query $top(X_1, \ldots, X_n)$ is finite (at most d^n where d is the number of constants appearing in the program). There are also finitely many non-ground answers. Either $X_i = c$, for some constant c, or $X_i = X_j$ for one or more other query variables X_j, or X_i is unconstrained.

Crossword Compilation Crossword compilation is an application of $GP(Datalog)$. Each word in the lexicon is recorded as a fact for the user-predicate $word$, thus:
$word(a, r, k)$
$word(a, r, m)$
$word(a, r, r, a, y)$
...
As explained in the introduction, the problem is expressed as a set of propagation constraints, each one representing a blank word in the crossword

$top \leftarrow$
constraint $word(A1, A2, B1, A4, C1)$,
constraint $word(B1, B2, B3, B4)$,
constraint $word(C1, C2, C3)$,
...

The program is evaluated by performing generalised propagation on all the individual words, until the propagation sequence reaches a fixpoint. On most real crosswords, the first fixpoint is reached without extracting any information whatsoever out of the propagation constraints. Next one of the constraints is selected for unfolding. For example the lexically first constraint $word(A1, A2, B1, A4, C1)$ might be chosen. The result of unfolding is the addition to the constraint store of a conjunction of equalities, say $(A1 = a) \wedge (A2 = r) \wedge (B1 = r) \wedge (A4 = a) \wedge (C1 = y)$. Now propagation on the other constraints is attempted. For example the system will search for all answers to the goal $word(b, B2, B3, B4)$ and attempt to extract an approximation of these answers. Propagation continues until the second propagation sequence is complete (in a typical crossword no information will normally be extracted after the second propagation sequence either).

In our implementation the cost of these fruitless propagation sequences is kept low by

- only attempting propagation on constraints affected by the last unfolding or the current propagation sequence
- ceasing the search for answers after finding only a few, since the approximation of only a few answers soon becomes general enough to allow the search to be terminated (see above section 4.7).

As the crossword fills up, the propagation begins to produce information which ensures no bad choices can be made later. At this point propagation sequences begin to grow in length, as information extracted from one constraint enables further information to be extracted from others.

To sum up, little work is invested in generalised propagation by the system until it actually starts to be useful. Evidence for the naturally good behaviour of generalised propagation on crossword compilation is this. The crossword program sketched above is perfectly naive. In fact a meta-program has been written which takes any crossword drawn as a grid and generates such a program automatically. Yet generalised propagation applied to the resulting program happens to yield a crossword compilation algorithm very similar to one developed specially for crosswords and described in [Ber87]. On a Sun3 workstation, with a 25000 word lexicon, the Herald Tribune crossword can be compiled by Propia in a few minutes.

Propagation as Consistency Checking Various alternative approximation languages can be used for generalised propagation. The more expressive the approximation language the more information is extracted, but the costlier the propagation.

One very simple approximation language has just two approximation constraints: $true$ and $false$. We call this the *consistency* approximation language. With this language the result of propagation on a constraint is either nothing (in case an answer was found) or failure (in case none could be found). The behaviour of the crossword program with this language is to use each constraint as a continual check on the choices made so far. This ensures that no inconsistent choices are made, but that no "active" constraint propagation is done.

The advantage of using such a trivial approximation language is that in this case topological branch and bound is very effective in optimising the evaluation of propagation constraints. Suppose a certain constraint is being evaluated for propagation. As soon as a single answer is found, the current approximation constraint (approximating the answers found so far) becomes $true$. Since $true$ is implied by the current constraint store (since it is implied by any constraint store) propagation terminates immediately.

Clearly with the trivial approximation language generalised propagation is very cheap to implement. It offers an alternative to intelligent backtracking, in this sense. If every user goal is annotated as a propagation constraint, as in the current example, then the propagation prevents any further (irrelevant) choices being made if any other goal is already unsatisfiable. This is because the failure is detected immediately when attempting propagation.

Equalities Between Variables For the crossword application above, the only useful information concerns values for variables (expressed as an equality between a variable and a constant). In this section we shortly demonstrate the usefulness of extracting information about equalities between variables. Applications where such information is important include those involving boolean variables, such as circuit design, analysis and testing, and propositional satisfiablity problems.

Such applications involve complex boolean functions describing the behaviour of, for example, circuit components which are already analysed. Each such function can be used immediately as a propagation constraint. Let us choose a very simple "and-gate" to illustrate the following discussion. Its behaviour can be described using four clauses:
$and(true, true, true)$.
$and(true, fals, false)$.
$and(false, true, false)$.
$and(false, false, false)$.
The approximation language admits any equality as an atomic approximation constraint. In a program where `constraint and(X,Y,Z)` appears as a goal, the following information can be extracted:

Constraint store	Information extracted
$Empty$	$Nothing$
$X = false$	$Z = false$
$X = true$	$Z = Y$
$Y = false$	$Z = false$
$Y = true$	$Z = X$
$Z = true$	$X = true \wedge Y = true$
$X = Y$	$Z = X$

Even though boolean variables have finite (2-element) domains, finite domain propagation cannot elicit any information in case, for example, the constraint store has $X = true$. In this case both Y and Z could take either value $true$ or $false$. For real problems in the applications listed above, the extraction of information of the form $Z = Y$ is essential for performance reasons.

To obtain such a behaviour on these applications in CHIP [SD90, SD87b, SD87a, Sim88, SP89] it was necessary to use a form of guarded clause called "demons". Demons are predicates whose goals behave as follows. Each goal delays until one of the clauses in the predicate definition has a head which matches it. The matching clause is then exclusively chosen for evaluating this goal, and if it fails none of the other clauses are tried.

The demon clauses defining the *and* predicate explicitly use the constraints in the "Constraint Store" column above as guards. Expressed using the syntax of Andorra [HJ90] the *and* demons are:
$and(X, Y, Z) \leftarrow X = false | Z = false$
$and(X, Y, Z) \leftarrow X = true | Z = Y$
...
Whilst the demons for *and* are built-in in CHIP, for complex boolean functions the CHIP programmer is required to generate a set of demons for himself. Propagation constraints like *and* can often be encoded into demons. However, experiments have shown that the number of distinct demons required for even moderately complex boolean functions can often be over ten thousand.

To encode a set of demons for a propagation constraint the programmer must consider all cases and generate each demon body by, effectively, performing the propagation in their head. Since the resulting demons need do no propagation at runtime, they are more efficient to execute. It is therefore interesting to record that Propia when applied to a benchmark of propositional satisfiability problems [MR91], had execution times on the same hardware broadly comparable with that obtained using CHIP's demons.

The relationship between generalised propagation and committed choice languages will be discussed in more detail below.

5.2 GP(HU)

In the last section we examined applications run using Propia which did not use functions. We now consider what happens when functions are used in propagation constraints. The information extracted remains information about equalities between terms. The answers to a query are always expressed as conjunctions of equalities of the form $X = T$, where X is a variable in the original query, and T is a term (possibly involving non-query variables). In the Herbrand universe there are arbitrarily long finite sequences of monotonically stronger such equalities, but no infinite sequences which are satisfiable. Consequently the noetherian requirement is still satisfied.

The presence of function symbols enables generalised propagation to be performed over lists. In particular we shall examine propagation on the *member* predicate defined as follows:
$member(X, [X|\_])$
$member(X, [\_|T]) \leftarrow member(X, T)$
In many applications it is of interest to detect the success or failure of membership as soon as possible, just using member as a check. Yet even this is a serious problem (see for example [Nai86]). For example even if the tail of the list is known most control regimes require the check to delay until the head of the list either equals or fails to unify with the first argument.

Generalised propagation can be applied to any *member* propagation constraint without fear of non-termination. The information extracted from `constraint member(M,[E1,...En|Tail])` can be summarised as follows.

- If $Tail$ is empty, then
  - M becomes equal to the most specific generalisation of $M1, \ldots, Mn$ where Mi is the most general unifier of M and Ei. If none of the Ei unify with M, the result is $false$.
  - Ei becomes equal to the most general unifier of Ei and M if Ei is the only element that unifies with M. Otherwise there are no resulting constraints on Ei.
- If $Tail$ is a variable, then
  - There are no resulting constraints on M
  - There are no resulting constraints on any Ei
  - If none of the Ei unify with M, then $Tail = [\_|\_]$

The effect of the topological branch and bound in pruning the search for the infinite set of answers which return bindings for the tail is essential to ensure termination.

It is very instructive to try and construct ways of expressing the same propagation using guarded clauses!

So far $GP(HU)$ has only been applied to a few puzzles, and for encoding alternatives within a single term. This has been used to achieve an approximation to finite domain propagation in a simple way.

6 Generalised Propagation and Other Approaches

There are many overlaps with other work and in this paper it is not possible to include a full comparison with all of it. We have tried to consider more closely related research which is particularly interesting and influential. However even in the short list considered here, there are many points on which our comparison could be greatly expanded.

6.1 Propagation in CHIP

Generalised propagation is descended from propagation in CHIP. CHIP provides two propagation inference rules, lookahead and forward checking. In this section we shall briefly study how generalised propagation relates to the two different rules.

Firstly, of course, propagation in CHIP can only produce reductions in finite domains. Generalised propagation is domain independent and can be used to extract interpreted constraints over the domain of computation whatever it is. Within the framework of generalised propagation, systems have been implemented to perform propagation yielding equality constraints and yielding finite domain constraints.

When we compare the implementation of generalised propagation yielding finite domain constraints with CHIP's lookahead inference rule, there remain significant differences.

The algorithm underlying CHIP's lookahead iterates over each finite domain of each free (unlabelled) variable to determine if there is a labelling of the remaining variables consistent with it. Because of the high cost of the algorithm (worst case $n * d^n$ where n is the number of free variables and d the size of the largest domain), lookahead in CHIP is delayed until there are maximally two free variables.

Generalised propagation uses a topological branch and bound procedure which iterates over the answers to the goal rather than the variable domains. Consequently there is no penalty associated with constraints involving more than two free variables.

It is possible to find examples where topological branch and bound is more efficient than CHIP's algorithm and vice versa. However the flexibility offered by the choice of approximation constraints makes it possible to perform efficient generalised propagation on a large range of examples.

CHIP's forward checking rule offers a restricted form of propagation in which the result of propagation must be logically equivalent to the constraint itself in the current state. For example if the current state has $X = 2$ and p is defined by
$p(1,2)$
$p(2,1)$
$p(2,2)$
then $p(X,Y)$ is logically equivalent to $Y \in \{1,2\}$. In CHIP forward checking is simply delayed until there is at most one free variable. The set of values in the domain of this variable consistent with the constraint are all the answers to the constraint, and they are its only answers. The required logical equivalence is therefore always achieved between the constraint and the reduced domain (as illustrated by the above example).

A generalisation of forward checking can be achieved within the framework of generalised propagation by specifying a rather special approximation language. The propagation language *self* is defined to include all (conjunctions of) answers to the current goal. Each such answer is, in fact, a conjunction of interpreted constraints in the domain of computation. Generalised propagation using this language has precisely the effect of forward checking.

This language (or strictly this class of languages) has been implemented in Propia. In fact its implementation turned out to be very simple (requiring only three lines of code!). Just as for other language-based approximations of propagation, Propia uses topological branch and bound for computing the best approximation in the language *self*. Consequently generalised forward checking is, as expected, quicker to evaluate than generalised propagation on the same constraint using basic constraints as the approximation language.

Notice that there is no longer any need to delay forward checking, as in CHIP. Notice also that generalised forward checking is indeed a generalisation of forward checking in finite domains. It is applicable to arbitrary computation domains, and returns as a result interpreted constraints in the current domain of computation, whatever it is.

6.2 Most Specific Logic Programs

The instance $GP(HU)$ of generalised propagation extracts information from propagation constraints which is precisely the most specific generalisation described in [MNL88]. In this earlier work, the most specific generalisation of a set of possible solutions was calculated in advance of execution, so as to transform a program statically into one which was more efficient and had other better properties. Various algorithms have been proposed for calculating most specific logic programs, some using bottom-up evaluation and others breadth-first.

Generalised propagation is, by contrast, performed at runtime, repeatedly as more information becomes available and more information can be extracted. The topological branch and bound procedure, based on a pruned top-down evaluation, is much more efficient to implement and makes practicable the extraction of most specific generalisations at runtime.

6.3 Relaxed Tell

In [HD91] an operational semantics for constraint logic programming is introduced which offers an operation called *relaxed tell*. The relaxed tell operation extracts from a non-basic constraint an approximation. The operation requires two functions, a *relaxation* function and an approximation function which depends on the relaxation function.

A relaxation function r maps the constraint store S to an approximation $r(S)$ satisfying $\models S \rightarrow r(S)$. CHIP uses such a relaxation function in its treatment of arithmetic constraints over finite domains. A finite domain for a variable V, such as $\{1,2,4\}$ can be approximated by its end points, $1 \leq V \leq 4$.

An approximation function ap (given a relaxation function r) maps a non-basic constraint C and a store S to an approximation $ap(S,C)$ satisfying $(r(S) \wedge C) \rightarrow ap(S,C)$. CHIP also uses approximation functions in its treatment of arithmetic constraints over finite domains. For example the linear constraint $1+V1=V2$ is handled by using the equations to reduce the upper bounds and increase the lower bounds of the variable domains so that the equation is satisfied by the new bounds. Thus if $V1 \in \{1,3\}$ and $V2 \in \{2,3\}$, the result of approximation on the above equation is $1 \leq V1 \leq 2$ and $2 \leq V2 \leq 3$.

The requirement on the approximation function in the relaxed tell framework is that it must approximate the constraint C, whereas in the framework of generalised propagation the result approximates all the answers to the constraint. This difference arises because relaxed tell is designed for non-basic *built-in* constraints such as arithmetic ones. For generalised propagation *any user goal* can be annotated as a constraint. In this case there is a clear definition of an answer to the constraint, but the logical semantics of the constraint itself is more difficult to pin down. The logical semantics for program clauses does not license any negative consequences. However in this case no pruning information could be extracted from propagation constraints! For our purposes it would therefore be necessary to use some form of minimal model semantics for constraint logic programs, with all the restrictions this entails [JL87].

Apart from the restriction to built-in constraints, relaxed tell is an abstraction of generalised propagation. The inclusion of a relaxation function makes it strictly more powerful than generalised propagation, whose "relaxation function" is just the identity function. The disadvantage of using a relaxation function is that propagation on a single constraint cannot be guaranteed to yield a fixpoint. In fact the example of approximation above has this property. If the result of propagation is added to the constraint store the resulting store now has a different relaxation $1 \leq V1 \leq 1$, which enables further useful propagation to be performed *on the same constraint.* This means that the efficient AC-3 algorithm no longer produces complete propagation sequences.[5]

6.4 Guarded Clauses and Concurrent Constraint Logic Programming

It is not possible in this paper to make a comparison of generalised propagation with the different languages in these frameworks. At an abstract level propagation constraints can be seen as deterministic processing agents which communicate with the constraint store using *relaxed tell.* More concretely it is interesting to specify precisely what communications take place in terms of *ask* and *tell*, and how this behaviour reflects the declarative semantics of the constraint.

We can therefore attempt to encode the behaviour of a propagation constraint as a set of definitions using committed choice, guarded clauses. Let us take finite domain propagation as an example and use $ask\ X \in \{C_1,\ldots,C_n\}$ to ask if the current constraint store implies that $(X=C_1) \vee \ldots \vee (X=C_n)$, and $tell\ X \in \{C_1,\ldots,C_n\}$ to tell this formula to the constraint store. For $constraint\, p(X,Y)$, where p is defined as
$p(1,2)$
$p(2,1)$
$p(3,1)$
we could express finite domain propagation thus:

$$
\begin{array}{lll}
constraint\ p(X,Y) & \leftarrow true & \mid tell\ X \in \{1,2,3\}, tell\ Y \in \{1,2\}, constraint\ p1(X,Y) \\
constraint\ p1(X,Y) & \leftarrow ask\ X \in \{2,3\} & \mid tell\ Y = 1 \\
constraint\ p1(X,Y) & \leftarrow ask\ X = 1 & \mid tell\ Y = 2 \\
constraint\ p1(X,Y) & \leftarrow ask\ X = 2 & \mid tell\ Y = 1 \\
constraint\ p1(X,Y) & \leftarrow ask\ X = 3 & \mid tell\ Y = 1 \\
constraint\ p1(X,Y) & \leftarrow ask\ Y = 1 & \mid tell\ X \in \{2,3\} \\
constraint\ p1(X,Y) & \leftarrow ask\ Y = 2 & \mid tell\ X = 1
\end{array}
$$

This encoding is similar to that used for the *and* demons (see section 5.1 above).

The main drawback of using such an encoding is the huge number of clauses necessary to capture each interesting propagation. We hypothesise that if conjunctions of basic constraints are admitted in the guard, the number of guarded clauses can rise exponentially with the number of clauses needed to express the propagation constraint.

[5] In CHIP, which uses AC-3, it is therefore sometimes necessary to state constraints twice!

A second drawback of guarded clauses is, paradoxically, their great expressive power. For example it is possible to express the *merge* operation using guarded clauses, although this operation has no logical semantics. In general it is not possible to give a declarative semantics for a set of guarded clauses, and thus it is not possible to state the effect of a program except in terms of the operational behaviour of its clauses.

There is a "logical subset" of guarded clause programs that have a logical semantics. It is possible to state when a set of "logical" guarded clauses is sound with respect to a logic program specification as in [Smo91]. However even for such logically sound guarded clauses there remains the question of completeness. There seems no simple way to determine when the behaviour of a set of clauses is equivalent to the behaviour of generalised propagation. For example it is only possible to confirm that the encoding of *constraint* $p(X, Y)$ using guarded clauses above really does extract all possible propagations in all possible constraint stores by performing an exhaustive analysis on constraint stores. The set of interesting constraint stores to be analysed soon grows prohibitively large for non-trivial constraints (see also above section 5.1).

A form of guarded rules with multiple heads is being investigated at ECRC [Fru91], which provides a language for expressing constraint simplification. The rules are called *simplification rules*. In many cases it would be practical to express certain interesting propagations as simplification rules. The integration of these *simplification rules* into our framework would make it possible to encode the results of static analysis and partial evaluation of generalised propagation. Consequently the whole range of possibilities on the continuum between compilation and interpretation of generalised propagation would be available in one system.

6.5 Andorra

A relationship has been often pointed out between David Warren's Andorra principle [War88] and the preference for deterministic computation which underlies constraint propagation. Whilst the two principles cannot be clearly distinguished, their embodiment in Andorra [HJ90] and in generalised propagation can usefully be compared.

Andorra promotes deterministic computations. The control of how hard to work to find subcomputations that yield determistic results has reached a considerable degree of sophistication. However the basic idea is to perform parts of the computation locally and if the result is deterministic to make it available globally, adding the resulting constraints to the constraint store. This is similar to extracting results from propagation constraints.

In a local computation in Andorra, nothing is thrown away. This is quite different from constraint propagation which finds many answers, extracts "common" information from them all, and then throws the answers away again. This can in practise make constraint propagation more expensive than Andorra's deterministic promotion, but it also makes it possible to extract more information deterministically than can be done in Andorra. For example generalised propagation extracts $X = f(\_)$ from the propagation constraint *constraint* $p(X)$ defined by

$p(f(a))$
$p(f(b))$

However the evaluation of $p(X)$ is not determistic so no information can be extracted in Andorra.

A second difference has to do with the dependence of information extracted on the precise syntax of the program. In Andorra the information that can be extracted from a local computation depends on the precise clausal definitions of the goal predicates involved. For example we could recode $p(X)$ above as

$p(f(Y)) \leftarrow q(Y)$
$q(a)$
$q(b)$

to get more information extracted by Andorra from the goal $p(X)$. In constraint propagation the information extracted is independent of the program syntax. It depends only on the semantics of the program predicates. Therefore constraint propagation has a more abstract behavioural semantics than deterministic promotion in Andorra.

7 Conclusion

The same word "constraint" has been used to describe two rather different extensions of logic programming. In one extension ($CLP(X)$) "constraints" involve interpreted predicates whose interpretation on the underlying domain is predefined. In the other extension (based on CSP) "constraints" are goals which are used not for search but for deterministic reduction of the search space. This paper has extracted a more abstract concept which includes both uses of the word *constraint*.

The abstract concept is useful for clarifying our understanding of CLP, but this paper has shown that it also yields immediate practical benefits. A generalisation of propagation has been introduced which integrates the constraint behaviour of both extensions. This enables techniques of local consistency enforcement from CSP to be applied to arbitrary goals in arbitrary $CLP(X)$ programs. The result is called $GP(X)$, for "generalised propagation parameterised on the computation domain X".

Propagation on a goal G in $GP(X)$ requires that the system extracts a constraint approximating all the answers to G. The paper has introduced a generic algorithm for generalised propagation which avoids enumerating all the answers to a propagation constraint. Instead the retrieval of answers is interleaved with approximation steps, so that an approximation to the answers found so far is always maintained. This approximation is used to cut branches in the search for answer, in a way similar to branch and bound. Additionally it is used to cut all the remaining branches in the search tree, when the approximation becomes too general to be useful. The algorithm has been called *topological branch and bound*.

Generalised propagation offers very flexible control via the choice of approximation constraints. If only a coarse approximation is offered the topological branch and bound drastically prunes the search tree, thus making generalised propagation relatively cheap. If a finer approximation is offered, more information is extracted from each propagation constraint, enabling the global search to be more reduced.

An implementation (called *Propia*) of generalised propagation over the Herbrand universe has been described. Experiments with Propia have shown that generalised propagation enables problems to be simply stated and efficiently solved in a way not possible using either $CLP(X)$ or propagation based on CSP. It has been very rewarding to take pure logic programs as specifications and, by simply annotating certain goals as propagation constraints, to achieve an efficient implementation. A very important feature of the resulting programs is their guaranteed correctness with respect to their specification. This can be contrasted with the encoding of the same problems using demons (a special form of guarded clause), which cannot be validated against the specification since they have no declarative semantics.

As to the future, further implementations of generalised propagation are being developed for new computation domains, thus expanding the range of problems that can be naturally expressed as $GP(X)$ programs. We are also investigating the notion of propagation constraints as concurrent processing agents. In this view generalised propagation is an interesting special case of concurrent constraint logic programming, in which the operational semantics can be dramatically simplified (and for which there is always an equivalent declarative semantics). Finally partial evaluation of $GP(HU)$ is already under investigation at ECRC, with the results expressed in the form of demons. With the integration of simplification rules into our system (see section 6.4 above), the potential for optimisation of $GP(X)$ programs can be fully explored.

8 Acknowledgements

We are grateful to our partners in the CHIC project, in particular the CHIC user group, for posing problems which have helped to motivate generalised propagation and providing test cases to try it out. This work could never have started without the ground-breaking work of former members of the CHIP team. Our thanks to them as well. We thank the CORE team for valuable discussions, and Alexander Herold for his support and careful criticism of earlier versions of this report.

References

[Ber87] H. Berghel. Crossword compilation with Horn clauses. *The Computer Journal*, 30(2):183–188, 1987.

[Cla79] K.L. Clark. Predicate logic as a computational formalism. Technical Report 79/59, Imperial College, London, 1979.

[Col85] A. Colmerauer. *Theoretical Model of Prolog II*, pages 3–31. Ablex Publishing Corporation, 1985.

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

[DSV90] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8:74–94, 1990.

[DVS$^+$88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, November 1988.

[Fik70] R.E. Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

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

[Fru91] T. Fruehwirth. Introducing simplification rules. Technical Report LP, ECRC, 1991.

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

[GB65] S.W. Golomb and L.D. Baumert. Backtrack programming. *Journal of the ACM*, 12:516–524, 1965.

[HD91] P. Van Hentenryck and Y. Deville. Operational semantics of constraint logic programming over finite domains. In *Proc. PLILP'91*, Passau, Germany, Aug 1991.

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

[HJ90] Seif Haridi and Sverker Janson. Kernel andorra Prolog and its computation model. In *Proc. of the* 7^{th} *Int. Conf. on Logic Programming* [ICL90], pages 31–46.

[ICL87] *Proceedings of the* 4^{th} *International Conference on Logic Programming.* MIT Press, 1987.

[ICL88] *Proceedings of the* 5^{th} *International Conference and Symposium on Logic Programming*, Seattle, 1988. MIT Press.

[ICL90] *Proceedings of the* 7^{th} *International Conference on Logic Programming*, Jerusalem, Israel, 1990. MIT Press.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages (POPL'87)*, Munich, FRG, January 1987.

[Llo84] J.W. Lloyd. *Foundations Of Logic Programming.* Springer-Verlag, 1984.

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

[MAC$^+$89] M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D. De Villeneuve, A. Herold, P. Kay, B. Perez, E.Van Rossum, and J. Schimpf. Sepia - an extendible prolog system. In G. X. Ritter, editor, *Information Processing 89*, San Francisco, September 1989. Elsevier Science Publisher B.V.

[Mah87] M. J. Maher. Logic semantics for a class of committed-choice programs. In *Proc. of the* 4^{th} *Int. Conf. on Logic Programming* [ICL87], pages 858–876.

[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.

[MNL88] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In *Proc. of the* 5^{th} *Int. Conf. and Symp. on Logic Programming* [ICL88], pages 909–923.

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

[MR91] I. Mitterreiter and F. J. Radermacher. Experiments on the running time behaviour of some algorithms solving propositional calculus problems. Technical Report Draft, FAW, Ulm, 1991.

[NAC90] *Proceedings of the 1990 North American Conference on Logic Programming.* MIT Press, 1990.

[Nai86] L. Naish. *Negation and Control in Prolog*, volume 238 of *Lecture Notes in Computer Science.* Springer, 1986. PhD. Thesis, Melbourne Univ.

[RHZ75] A. Rosenfeld, A. Hummel, and S.W. Zucker. Scene labelling by relaxation operations. Technical Report TR-379, Computer Science Department, University of Maryland, 1975.

[Sar89] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Pittsburgh, Pa, January 1989.

[SD87a] H. Simonis and M. Dincbas. Using an extended prolog for digital circuit design. In *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, pages 165–188, Munich, W.Germany, October 1987.

[SD87b] 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. (to appear).

[Sim88] H. Simonis. Test pattern generation with logic programming. In *New Aspects of Research for Testing of VLSI Circuits*, Ising, W. Germany, March 1988.

[SKL90] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. Janus: A step towards distributed constraint programming. In *Proceedings of the 1990 North American Conference on Logic Programming* [NAC90], pages 431–446.

[Smo91] G. Smolka. Residuation and guarded rules for constraint logic programming. Technical Report 12, Digital PRL, June 1991.

[SP89] H. Simonis and T. Le Provost. Circuit verification in chip: Benchmark results. In L.J.M. Claesen, editor, *Proceedings of the IFIP TC10/WG10.2/WG10.5 Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, Belgium, November 1989. IFIP, North Holland, Elsevier Science Publishers.

[SS80] G.J. Sussman and G.L. Steele. CONSTRAINTS: A language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, January 1980.

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

[VD86] P. Van Hentenryck and M. Dincbas. Domains in logic programming. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI'86)*, Philadelphia, PA, August 1986.

[War88] D.H.D. Warren. The andorra model. Presented at the Gigalips Workshop, Univ. of Manchester, 1988.