Constraint Logic Programming on Boolean, Integer and Real Intervals

Frédéric Benhamou[*], William J. Older and André Vellino[†]

May 10, 1994

Abstract

Imperative programming languages for computing on intervals stand in contrast to systems of relational interval arithmetic that are seamlessly integrated into a logic programming language like CLP(BNR). The combined power of a symbolic, logic programming language on the one hand and a mathematically correct, logically sound and computationally efficient programming system (constraint arithmetic on intervals) on the other, is considerably more powerful than either taken separately. Boolean, integer and real-valued simultaneous constraint equations can be mixed freely in CLP(BNR), not only to contain floating point errors and perform sensitivity analysis but also to express a wide variety of linear and non-linear programming problems, scheduling and configuration problems as well as optimization and operations research problems. Some simple examples of how this can be done are given as well as a summary report on experimental applications of this technology to industrial problems.

1 Introduction

Conventional programming systems for computing on intervals, such as enhanced Pascal and Fortran compilers have existed for many years [6]. More

[*]Groupe d'Intelligence Artificielle, Faculté des Sciences de Luminy, case 901,163, avenue de Luminy, 13288 Marseille Cedex 9 FRANCE, benham@gia.univ-mrs.fr

[†]Bell Northern Research, Computing Research Laboratory, PO Box 3511, Station C, K1Y 4H7 Ottawa, Ontario, Canada, {vellino,wolder}@bnr.ca

recently, Cleary suggested that a *relational* form of interval arithmetic could be seamlessly integrated into a logic programming language (Prolog)[3]. This idea is qualitatively different from other forms of automated interval arithmetic because the combined power of a symbolic, logic programming language on the one hand and a mathematically correct, logically sound and computationally efficient programming system (constraint arithmetic on intervals) on the other, is considerably more powerful than either taken separately.

Constraint logic programming originated with the observation by Alain Colmerauer that if one regards the terms in Prolog as forming a certain algebraic structure, then the unification algorithm can be regarded as a kind of equation solving in that algebra. Since this immediately opens the door to substituting alternate algebraic structures with their own equation solving algorithms, while keeping the rest of the language essentially unchanged, it liberates Prolog from its somewhat confining relationship to SDL Resolution. These ideas were carried out first in Prolog II, then later in Prolog III and other systems based on boolean unification, linear programming, and Groebner base technology.

There are a number of motivations, consistent with Colmerauer's view, for integrating relational interval arithmetic into constraint logic programming. In the first place, conventional arithmetic in Prolog is essentially no different from conventional arithmetic in any procedural programming language and suffers from all the same problems (e.g. the mathematical incorrectness of floating point arithmetic, rounding errors etc.). Secondly, Prolog is at heart a declarative, relational language and the presence of conventional, procedural arithmetic in Prolog spoils the "logical" character of the language.

The relational model for computing on real intervals centers around the notion of *narrowing*. The first proposal for a relational arithmetic using intervals by John Cleary in [3] was designed to address the problem that arithmetic in Prolog is evaluated functionally (whereas Prolog is otherwise a relational language). Just as the Prolog unification mechanism "narrows" the space of possible instantiations of a variable as computations proceed so, in this model, the addition of constraints on an interval-bounded variable restricts the range of reals to which it could be bound. Intervals narrow by raising their lower bounds, lowering their upper bounds, or both. Narrowing occurs when additional constraints are applied to an interval.

The relational character of constraints on intervals can be given a clear lattice-theoretical model with a fixed-point semantics as shown in [7]. The logical semantics for interval narrowing and its connections to existing CLP systems such as CLP($\Re$) and CHIP is discussed by J. Lee and M. Van Emden in ([4]).

1.1 Overview

Cleary's ideas were first fully implemented at Bell-Northern Research (BNR) in 1987 on a Macintosh. Since then, the language has evolved into CLP(BNR) (constraint logic programming on Booleans Naturals and Reals), implemented on Unix workstations, which includes constraints on integer-bounded intervals and on boolean intervals as well as constraints on real-valued intervals. A general description of the technique of relational interval arithmetic and its abstract semantics is given in [7] and [2]. A detailed description of how to program in CLP(BNR) can be found in [8].

In general terms, constraint interval arithmetic can be described as a method for taking a set of mathematical relations between certain quantities, and constructing from them a process which maps initial intervals on all quantities to final intervals for these quantities. It does this by constructing a proof (using interval fixed point iteration) that any solutions in the initial intervals must lie in the final intervals. The mapping from initial to final intervals is contracting (the final intervals are subintervals of their initial values), idempotent (repeating the same constraint has no effect on the interval values), and inclusion monotone (smaller initial intervals yield smaller final intervals). Moreover, the mapping from initial to final intervals is independent of the order in which constraints are imposed and of the details of the implementation.

In CLP(BNR) the user can write a set of simultaneous arithmetic equations and inequalities that relate and constrain interval-bounded variables. These equations are then compiled into a constraint network whose nodes are instances of primitive arithmetic relations supported by the system. The variables of the problem are associated with a pair of floating point numbers that determines the interval's upper and lower bounds; these intervals are partially ordered by set inclusion. The constraint network then defines an operator on intervals which is monotone and contracting with respect to the partial order, and which is also idempotent and correct. Correctness here means that no valid solution (e.g., in the theoretical real numbers) is ever eliminated during contraction; as a consequence of correctness the separation of multiple solutions must involve a mechanism such as backtracking or or-parallelism. This technique, since it works by removing non-solutions, can be regarded as a model elimination proof procedure, and this gives it quite different characteristics from constructivist exact arithmetic systems.

Because the boolean, integer and real-valued intervals in CLP(BNR) share the same fundamental implementation and logical framework, it is possible to mix all three types of variables in a single problem. Thus CLP(BNR) can be used, not only to contain floating point errors and perform sensitivity analysis but also to express a wide variety of linear and non-linear programming problems, scheduling and configuration problems as well as optimization problem. In the reals, we have found that it can deal effectively with general non-linear functions, transcendental functions, and non-continuous functions, all on the same footing. The *solutions* to sets of simultaneous non-linear equations can be obtained by systematically narrowing constrained intervals using divide and conquer or branch and bound techniques.

One advantage of the relational character of interval arithmetic is that it encourages an economy of coding: inverse functions, for example, do not need to be defined explicitly. But its most important feature is that, since the same expressions can be used either to evaluate numeric or symbolic values, it is possible to combine symbolic techniques with numerical techniques. Furthermore, we can prove that numerical and symbolic techniques can be mixed with impunity, without concern for generating incorrect results, because interval arithmetic preserves the algebraic properties of expressions and interval operations are declarative, logical and provably correct.

2 Theory of Narrowing Algebras

The particular form that CLP(BNR) has assumed was in a large part dictated by that of its host language BNR Prolog. Whereas most of the CLP languages can be seen as the result of substituting an alternative algebraic structure in place of term unification in Prolog, BNR Prolog began with a lattice-oriented view of unification. If general Prolog terms are interpreted as a representation of their sets of ground instances, partially ordered by inclusion, and if one adds an empty term representing Prolog 'failure', then unification of two terms becomes a matter of replacing each by their common intersection or 'meet'. This leads naturally to the consideration of substituting the standard

Prolog lattice of finite trees with alternate lattices, e.g. the lattice of (freely generated) infinite terms over a finite symbol (atom) space.

In this lattice theoretic framework, a Prolog "call" (predicate call/procedure invocation) can be seen as an operator (self-map/lattice automorphism) which replaces the initial on-entry argument terms with their return values. With respect to the lattice ordering, the results are always as small as, or smaller than (i.e. less general than) the input values, so we are dealing with contraction mappings. In addition, since Prolog's logic variables are "write-once" variables that remain unchanged once instantiated (backtracking, of course, excluded), operators are almost always *persistent*, that is, act as the identity on anything smaller than their return values. (There are exceptions to this such as the meta-logical primitive `var` and any constructs that use it.) A less obvious common property is *monotonicity*, said of maps that preserve the partial order of terms; this captures the intuitive idea that more vague questions get (and deserve) more vague answers.

Contraction operators are called *pure* when they are both monotone and persistent. Unification, for example, is a pure operator, and indeed the quintessential pure operator, for every pure operator is equivalent to a unification. Pure operators commute, so their order of application is immaterial, and for such operators the "," connective is therefore commutative. Hence, non-commutativity of "," can always be traced to a failure of either monotonicity or persistence of one of the built-in "extra-logical" primitives.

The standard Prolog built-in arithmetic, although persistent, is not monotone, since it depends on the instantiation of its variables; it is perhaps the most important source of impurity in most Prolog programs. In addition to this formal failing, floating point arithmetic has the serious problem of being simply incorrect because of the usual rounding errors.

Cleary's original proposal for a "logical arithmetic" (based on primitive arithmetic relations computed using functional interval arithmetic) had the virtues of being relational (and hence compatible with a relational host language), strictly correct (because rounding errors were contained within interval bounds by outward rounding), and monotone (because of the "inclusion isotone" property of functional interval arithmetic). Persistence, however, was lacking, as was commutativity. But since the weaker property of idempotence was present, it appeared plausible that persistence could be maintained artificially by retaining every operation in a "constraint store" or "constraint network" and reapplying the operator as often as necessary using a kind of arc-consistency algorithm. This raised a number of theoretical issues, however, which were not fully resolved until later and which are discussed below.

Another problem with Cleary's original formulation concerned operations (such as multiplication/division of sign-indefinite intervals) in which the needed function interval operation did not exist. For various reasons, Cleary's solution to this problem which involved the introduction of choice points, was not followed in BNR Prolog, which chose in effect to postpone such choices, as suggested in the "Aurora Principle." As more complex primitives (e.g. trigonometric relations) were added to the system, this problem arose with increasing frequency, and Cleary's recipe for the construction of primitives was eventually replaced by a more general relational algebra prescription no longer dependent on this 'interval convexity' property (see [4], [2]). These issues became of crucial significance when the system was extended in 1992 to handle boolean and discrete variables.

In analogy with Prolog terms, the internal states of the relational interval arithmetic constraint system were represented as vectors of (closed) intervals, conceptually regarded as a Cartesian product of intervals *qua* sets, partially ordered by set inclusion, and with the null set representing Prolog failure. The meet operation is defined by set intersection. By adding a largest possible state one gets a complete lattice L in which the join of states is defined as the smallest state larger than all of them; note that this is not set union, and the lattice that results is non-distributive.

A constraint network over this lattice L is a finite collection of instances of primitive narrowing operators: idempotent monotone contractions. Thus one is concerned theoretically with the subalgebras of the space of monotone contractions $CM(L)$ which are finitely generated by idempotents. Conveniently, $CM(L)$ is a lattice with respect to the induced partial order, meet, and join operations, and has top 1 (the identity map) and bottom 0 (fail map). As a collection of maps under composition as product, it is a monoid with identity 1 and two-sided zero 0. Furthermore, these two algebraic structures are nicely intertwined since the monoid product is jointly monotone with respect to the partial order to form a lattice-ordered monoid.

The first fundamental fact about this algebraic structure is that the absorption relations (defined solely in terms of the product) that express the relative strengths of idempotents coincide with both the induced partial order on idempotents and with the set inclusion order on their respective families of fixed points. That is the idempotent operator p is smaller than the idempotent $q \leftrightarrow pq = p \leftrightarrow qp = p \leftrightarrow$ every fixed point of p is also a fixed point of q. The second fundamental fact is that the join of fixed points of an idempotent is also a fixed point.

If two idempotents commute, then their product is also an idempotent. But if they do not commute, then one can iterate their product and it will converge to an idempotent. This defines an "iterated product" construction for generating a new idempotent $p * q$ from a given ordered pair of idempotents. Given a constraint network regarded as a set of idempotents, there are then possibly many different ways to form an iterated product of them, but if this operation can be shown to be both commutative and associative then all these different implementations will produce the same result, the unique operator defined by the network.

To show this, one first uses the closure of fixed points under join to show that there is a bijective correspondence between narrowing operators and join-closed families of fixed points. This, together with the observation that a state is a fixed point of an iterated product $p{*}q$ iff it is both a fixed point of p and of q, implies that the correspondence can be extended to meet: r is the largest idempotent smaller than both p and $q \leftrightarrow r = p{*}q \leftrightarrow r = q{*}p \leftrightarrow$ every fixed point of r is a common fixed point of p and q. Hence, the idempotents themselves form a lattice with respect to the underlying partial order, with the iterated product providing the meet and (it turns out) the underlying join as join, and this implies (among other things) the uniqueness of the operator associated with a constraint network, however it may have been implemented among the many choices available.[1]

It is worth noting that this theory applies to the constraint network in its entirety, but as a user one can usually only observe the action on a relatively few "visible" variables and the vast array of intermediate variables remain hidden. It is therefore significant that such a projection of the lattice of narrowing operators also appears as a lattice of narrowing operators over the visible variables, the projection mapping being monotone, while, in general, the underlying monoid structure and CM-structure is not preserved under such projections.

[1]Regarded just as an abstract piece of mathematics, this structure is fascinating partly because of its several connections with classically important constructs. It can be regarded, for example, as a generalization (because it lacks an involution) of the lattice dual of the theory of reflexive binary relations, or as a generalization of the interior operators of general topology.

3 Relational vs. Functional Interval Arithmetic

Relational interval arithmetic in CLP(BNR) has many connections with the mathematical and computational disciplines of interval analysis and functional interval arithmetic founded by Moore in 1966 [5]. The basic concept of using intervals with floating-point representable bounds to contain roundoff error in floating point computations and the importance of the monotonicity property are common to both. However, this overlap of concepts and vocabulary can obscure the significant differences between the two paradigms.

We believe that, at bottom, there is only one fundamental difference between the paradigms: interval analysis is a language for talking *about* intervals, while CLP(BNR) is a language for *using* intervals to talk about reals. This distinction has so many ramifications and consequences that it can be most easily described through these diverse manifestations.

3.1 Functional Interval Arithmetic

Functional interval arithmetic has been codified as a mathematical theory of intervals, where an interval object can be represented as a pair `[L,U]` of reals with `L =< U`. Point intervals, those for which `L=U`, are then isomorphic to the reals and is convenient to think of the reals as being embedded in the intervals in this way. Intervals are partially ordered by set inclusion. For each of the primitive mathematical functions on the reals there is a "lifted" or extended version for intervals (which however may be only a partial function, as for example, division by intervals containing `0` is undefined.) The extended function is uniquely determined by the requirements that it produces intervals that agree with the usual functions on point arguments, and that it be monotonic with respect to interval inclusion, and that it contain no points that are not actually necessary.

Intervals are considered to be equal if and only if they have the same endpoints, or equivalently, if each includes the other. With this notion of equality and primitive functions, some of the fundamental axioms of the reals, such as the distributive law, no longer hold and consequently the algebraic structure is quite different from that of the reals (or any other standard structure). Mathematically equivalent expressions over the reals thus become in general inequivalent when extended, and this captures the fact that some may be better approximations or better behaved than others.

Implementations of this mathematical model naturally provide an interval type or data structure and the library of implemented interval functions (ideally as an Abstract Data Type) within either a conventional procedural or functional programming language. The entire package is quite consistent, and apart from relatively minor details, there are no basic decisions to make nor alternatives to consider.

In terms of usage, much (but not all) of the literature of interval analysis gives the impression that the intent of the usual application is to gain a greater control over the sources of error in conventional deterministic algorithms.

3.2 Relational Arithmetic in CLP(BNR)

The CLP(BNR) world, on the other hand, begins with the notion of (logic) variables constrained to only take values of a specific type (boolean, integer, or real) and the corresponding sets of constant values. The system will maintain estimates of the bounds of each integer and real variable (as an interval), and as constraints are added to the system the bounds of each variable generally approach one another and may eventually result in the instantiation of the variable to a point. Since these bounds contain information about the variable and can be queried using a meta-predicate, the constrained logic variable becomes in practice a somewhat different thing, perhaps something like the intuitive Newtonian notion of a "quantity" or physical variable that we all understood once upon a time, before becoming mathematicians and computer scientists.

Relational expressions using the primitive relations $\{*, +, \div, -, exp, sin\}$ in these entities can be used to impose constraints between variables.[2] For convenience, the syntax of the expression language is the usual functional expression language, but the semantics are fully relational. Thus the ternary product relation is actually written as:

[2]It is important to note that the primitive interval relations of the language denote the corresponding mathematical relations on the reals in the strict sense that in the point limit in infinite precision the interval relations converge to the mathematical relation. (Away from this asymptotic limit the representation may become very approximate.)

```
Z == X * Y
```

(or, alternatively, as `X == Z/Y` or `Y == Z/X`) rather than as in the predicate form `product(X,Y,Z)`, but the latter is what is meant.

Furthermore, with relations (unlike functions) it is always permissible to write such expressions, even if `X` or `Y` contain `0`, since what would be domain violations in a functional world are handled just by appropriately narrowing the arguments in the relational world. For example, the statement:

```
R**2 == X**2 + Y**2
```

can compute `R` from known `X` and `Y`, `X` from known `R` and `Y`, and `Y` from known `R` and `X`, among many other things.

In particular, the evaluation of a relational constraint equation can narrow the bounds on *each* of the variables. For example evaluating the equation `X + Y == Z`, for initial values `X`=[3,7], `Y`=[2,8] and `Z`=[4,6] narrows all three intervals: `X` to [3,4], `Y` to [2,3] and `Z` to [5,6][3].

Just as $*$ denotes the usual multiplication relation and $+$ the usual addition relation, `==` denotes the usual equality relation.[4] As a consequence of this, the usual algebraic laws (universally quantified axioms and theorems) are preserved in CLP(BNR), which is, in this respect, quite different from traditional interval arithmetic. Thus instances of the distributive law

```
X * (Y + Z) == X * Y + X * Z
```

are tautologies (equivalent to `true`). Furthermore, any symbolic transformation of equations which can be justified by such laws preserves their denotation.

This mathematical constraint language lives in a logic programming world (very similar to Prolog) which provides the necessary programming constructs. This language, however, should be construed as an implementation of an intuitionist predicate calculus. An intuitionist 'or' is provided

[3]In this example the declarative reading of `X + Y == Z`, in the numeric interpretation is "the sum of some point in [3,4] and some point in [2,3] is equal to some point in [5,6]".

[4]The implementation of `==` is radically different from the equality used in standard interval analysis, as it involves narrowing both of its arguments to their common intersection, in exact analogy with unification.

through backtracking as in Prolog, and negation is formally Brouwerian, i.e. `not(not(P))` is not equivalent to `P`.[5]

The entire package (constraint equational language plus Prolog) is quite natural, and apart from relatively minor details, there are no basic decisions to make nor alternatives to consider. (However, the implementation has many decisions to make and there are many alternatives to consider, although they affect only performance and not meaning.)

CLP(BNR) has been applied in a variety of experimental applications (see section 7 below) and we conclude from our experience that the successful application of this technology requires the relinquishing of all attempts to micromanage the algorithmic details of computation in favour of concentrating on finding an adequate and practically useful formulation of a usually nondeterministic problem.

4 Symbolic Methods

Even though interval arithmetic is a numeric technique it enhances any symbolic methods with which it is made to cooperate. Unlike conventional floating point, interval arithmetic formally honors the algebraic laws of real arithmetic, in the sense that symbolic transformations based on those axioms can never lead to bogus contradictions due to rounding problems. Thus "redundant" relations, which conventionally can cause problems because of such bogus contradictions, become a positive enhancement instead of a problem. For example, in a linear system problem in constraint interval arithmetic, a pivoting operation produces an additional redundant equation rather than a replacement for one of the original relations.

Symbolic methods have, of course, the great advantage of generality, when they can be applied. But a numeric approach is often necessary because there is no symbolic solution, such as roots of polynomials of degree greater than 4, or because the symbolic solution is too unwieldy. Furthermore, numeric methods can take advantage of the specific quantitative situation in ways that are not available to general symbolic techniques. For example, constraint re-

[5]When `P` is a constraint expression,the success of not(P) indicates that P is provably inconsistent with the current state (other existing constraints), and the success of not(not(P)) indicates that it is *possible* that P may be consistent with the current state (without actually imposing the constraint P).

lations which are topologically close (in the sense of having "nearby" graphs) will generally behave in a qualitatively similar fashion, although one may be symbolically tractable and the other symbolically intractable.

However, the numeric nature of this technique implies that any system of finite precision floating point arithmetic with which it is implemented renders the precise bounds of computed intervals sensitive to the specific formulation of the problem. For example, `Y1 is (A + B) + (C + D)` and `Y2 is (((A + B) + C) + D)` may result in different computed bounds for `Y` because of the non-associativity of floating point arithmetic. However, in interval arithmetic, since both `Y1` and `Y2` are supersets of the answer `Y`, $\emptyset \neq$ `Y` $\subseteq$ `Y1` $\cap$ `Y2` is always true. Given a particular one of the many mathematically equivalent formulations (e.g. a specific set of parentheses for an expression) the theory developed in [7] then implies that final bounds are independent of the order in which the constraints are processed.

5 Computations as Proofs

The other aspect of constraint interval arithmetic, and one which makes it very different from traditional numerical techniques, is that its computations represent *proofs*, and these are always proofs of the *non-existence* of solutions. As proofs, they carry a degree of logical force generally absent from traditional floating point numerical computing, which is, by comparison, concerned only with heuristics. The fact that its proofs are always non-existence proofs also makes it very different from traditional exact (rational) arithmetic, in which computations can be thought of as constructive proofs of existence.

It is precisely because interval proofs are non-existence proofs that they can be interpreted as referring to the mathematical reals, even though only bounded precision constants actually appear in the proofs themselves and the constraint system itself has no notion of "real number" in the full mathematical sense. (Of course, these proofs refer as well to the rationals, computable reals, and non-standard reals.)

To take full advantage of this proof aspect of the technique, it is useful to formulate problems negatively, so that a "failure" indicates a successful proof. Thus in the problem of formal system verification mentioned above, one asks questions of the form: if components all lie within their specified tolerance intervals, can a system parameter lie outside its specification? A "no" answer then indicates that a successful proof of compliance has been constructed, thus achieving a formal verification for all systems characterized by the model and initial intervals. If, however, a contradiction is not found, then the final intervals indicate conditions in which the specifications might not be met, and thus provide a direct indication of where the design may be marginal.

The art of computation-as-proof is particularly striking in interval search techniques, as in problems of global non-linear optimization, where it is used to eliminate subregions where the optimum cannot occur. These algorithms, essentially of the branch-and-bound type, were originally developed in the context of functional interval arithmetic, but become much more elegant in the context provided by Prolog and relational interval arithmetic where Prolog backtracking automatically manages the housekeeping chores associated with branching, while the interval machinery provides the necessary bounds automatically.

6 CLP(BNR)

This section is intended to whet the reader's appetite with a taste of what it is like to program with the CLP(BNR) system (code excerpts from [8]).

An interval can be created using syntax of the form:

```
V:real(LB,UB)  V:integer(LB,UB)  V:boolean
```

where `LB` and `UB` are either expressions which evaluate to numeric values or they are unbound variables, representing $\pm\infty$. Arithmetic relations on intervals, i.e., constraints, will either fail (in the usual Prolog sense), or succeed, in which case the bounds of the intervals may have been changed (narrowed).

Establishing a constraint propagates information from the known to the less known

```
?- X:real(1,3),   Y**2==X.
   ==> X : real(1.0, 3.0),
       Y : real(-1.73205080756888, 1.73205080756888)
```

Here is the same relation with both integer arguments; note that `X` becomes bound:

```
?- X:integer(1,3), Y:integer, Y**2==X.
   ==> X : 1
       Y : integer(-1, 1)
```

If `Y` is constrained to be positive, then `Y` also becomes bound to its unique answer:

```
?- X:integer(1,3), Y:integer, Y**2==X, Y>0.
   ==> X : 1
       Y : 1
```

Similar rules apply to general boolean relations:

```
?- B:boolean, 1 == B and (C or ~D) .
   ==> B : 1
       C : boolean
       D : boolean

?- B:boolean,   1 == B and (C or ~D),   0 == B and C .
   ==> B : 1
       C : 0
```

In some cases where an equation has a unique solution, equation solving is automatic:

```
?- [X,Y]:real, 1 == X + 2*Y,  Y - 3*X == 0.  % pair of linear eqns.
   ==> X : real(0.142857142857143, 0.142857142857143)
       Y : real(0.428571428571429, 0.428571428571429)
```

Here is a more interesting example, but with non-linear (including transcendental) equation solving:

```
?- [X,Y]:real,X>=0,Y>=0,  tan(X) == Y, X**2+Y**2 == 5.
   ==>  X : real(1.09666812870547, 1.09666812870547)
        Y : real(1.94867108960995, 1.94867108960995)
```

Note that although the upper and lower bounds in these answers print the same at this printing precision, the internal binary forms must differ by at least one bit in the last place, or else the variables would have been bound to the exact answer.

For more complex problems, which may have multiple solutions, there is a "solve" predicate, written entirely in CLP(BNR) itself, which separates the solutions (by backtracking) and forces convergence. For example, to find roots of polynomials:

```
?- X:real(0,1), 0== 35*X**256 - 14*X**17 + X, solve(X).
   ==>  X : 0.0                                        % 1st sol.
   ==>  X : real(0.847943660827315, 0.847943660827315) % 2nd sol.
   ==>  X : real(0.995842494200498, 0.995842494200498) % last sol.
```

It is easy to see how one could go about implementing different searching algorithms (such as branch-and-bound) to explore the space of possible solutions to a set of equations.

The application of this technology to really complex problems is not always straightforward, because different formulations of the same problem can sometimes have quite different performance characteristics. Finding "good" formulations is therefore still a process of discovery, guided by experience, intuition, and a handful of basic principles, such as the deferring of choices, the first fail principle, and the controlled use of redundancy. Once found, however, such formulations are usually transparently clear (but possibly subtle) statements of the problem.

7 Applications

The prototype implementation of CLP(BNR) became available for experimental purposes in late 1992, and in the subsequent months it was used in a number of small (mostly a few days to a few weeks) exploratory projects dealing with a variety of topics of real interest. A workshop (ARIA'93) was held in Ottawa in August of 1993 to report on the results of these projects and the following is a brief summary of these workshop presentations.

One of the areas of computer engineering that requires solutions to simultaneous non-linear equations is the *Performance Analysis* of Client Server Tasking Systems. The application is a straightforward narrowing application on continuous variables, but it is interesting because of the particular kinds of its nonlinearity and the large size (aprox. 30MBytes) of some of the constraint networks (This work is being done by Prof. Shikharesh Majumdar and others at Carleton University.)

In the area of discrete combinatorial problems, of course, the problem of *System Configuration* looms large. Constraints that specify which pieces of equipment require others, how many objects can fit in a container and so forth are typical in this area and CLP(BNR) can be used to tackle the problem. One of the authors (Vellino) has explored a simple, declarative language for describing a class of discrete bin-packing problems, a procedure to compile it into a CLP(BNR) constraint generator, and the associated enumeration algorithms for solving specific packing problems. Experimentation with different problems illustrated the sensitivity of solution times, not only to enumeration algorithms, but also to problem formulation.

Another kind of configuration problem that has been tackled with CLP(BNR) is the problem of *Resource Allocation* in Field Programmable Gate Arrays (FPGAs). Rick Workman and Mike Kelly at BNR showed that in compiling code to small Field Programmable Gate Arrays (FPGAs) there is a combinatorial problem involved in assigning functions to different parts of the FPGA in such a way as to minimize the resources being used. A rapid prototype FPGA compiler was developed that used CLP techniques to prune a huge space (aprox. 10^{10}) of possible assignments and produced solutions in just a few seconds.

A different kind of combinatorial problem, described by Prof. Bernard Nadel from Wayne State University, is the problem of the *Mechanical Design of Gear Trains*. The problem is that of searching the design space for 5-speed transmissions (aprox. 10^8 qualitatively different designs) for a design that meets specific performance requirements. It illustrates quite elegantly the value of having mixed boolean, integer and continuous constraints.

An interesting and complicated problem in the continuous domain is the problem of *Timing Requirement Verification* in digital circuits. Dave Brown, Tammer Kamel and William Older from BNR designed an efficient prototype verification system in CLP(BNR) which could either prove that a possible timing violation could never occur (given certain assumptions about delays and external event timings) or else provide a detailed description of the event timing circumstances under which a violation might occur, including the effects of nonlinear correlations between various delays.

The other problems discussed at this conference included applying interval constraints to solve *Geometric Combinatorial Problems* (Gilles Pesant, University of Montreal); using boolean satisfiability to analyse the structural properties (e.g. possible deadlocks and livelocks) of *Petri Nets* (Angelo Bean, BNR, Montreal); using branch-and-bound techniques to compute exact solutions to small ($N < 20$) *Traveling Salesman* problems and experimentation with the enumeration of Kuhn-Tucker points within a branch-and-bound framework to solve small but classically difficult *Nonlinear Constrained Optimization* problems (both by William Older).

In all of these cases the final formulations are remarkably short and deceptively simple, although the process of arriving at them was occasionally not a direct one, and in the most interesting cases involved major conceptual changes to how to think about the problem.

8 Conclusions

CLP(BNR) combines the strengths of symbolic computation in a declarative language and the mathematical correctness of interval arithmetic. A large variety of problems in system design and analysis that involve simultaneous non-linear equations or scheduling or configuration or optimization can be expressed effectively in CLP(BNR) and have produced encouraging preliminary results. However different formulations of the same problem often have quite different performance characteristics which can make the process of finding efficient solutions quite difficult for inherently hard problems (such as travelling salesman). Further research deserves to be done in, for example, exploring the synergy between this technology and known techniques in numerical analysis and operations research.

References

[1] Benhamou F. and Colmerauer A. (eds), *Constraint Logic Programming: Selected Research*, MIT Press, 1993.

[2] Benhamou F. and Older W., "Applying Interval Arithmetic to Integer and Boolean Constraints" in *Journal of Logic Programming* (to appear).

[3] Cleary, J. C. "Logical Arithmetic", Future Computing Systems,**2** (2), pp.125–149, 1987.

[4] J.H.M. Lee and M.H. van Emden, "Adapting CLP($\Re$) to Floating Point Arithmetic", in *Proceedings of the Fifth Generation Computer Systems Conference*, Tokyo, Japan, 1992

[5] Moore, R. E. (Ed.) *Interval Analysis*, Prentice Hall, New Jersey, 1966.

[6] Moore, R. E. (Ed.) *Reliability in Computing (The role of Interval Methods in Scientific Computing)*, Perspectives in Computing, **19**, Academic Press 1988.

[7] Older W. and Vellino A., "Constraint Arithmetic on Real Intervals" in *Constraint Logic Programming: Selected Research*, Benhamou F. and Colmerauer A. (eds), MIT Press, 1993.

[8] Older W. and Benhamou F., *Programming in CLP(BNR)*, BNR Research report, 1993.