IMPLEMENTING CONSTRAINT SOLVERS IN B-PROLOG

Neng-Fa Zhou
Department of Computer and Information Science
CUNY Brooklyn College & Graduate Center
New York, NY 11210-2889, USA
zhou@sci.brooklyn.cuny.edu

Abstract Constraint Logic Programming (CLP) defines a family of programming languages that extend Prolog to support constraint solving over certain domains. Current state-of-the-art CLP systems are based on abstract machines or language constructs that are good for certain domains and propagation algorithms but are either not sufficiently expressive or inefficient for some other domains and propagation algorithms. B-Prolog provides a construct, called *action rules*, for programming interactive agents such as constraint propagators. As far as constraint propagation is concerned, an agent maintains dynamically a certain level of consistency for a constraint. This paper presents constraint solvers implemented in action rules for six domains, namely, finite-domains, Boolean, trees, lists, sets, and floating-point intervals. Some of the solvers such as the finite-domain and set solvers are competitive in performance with the fastest solvers available now.

Keywords: Constraint propagation, constraint solving, constraint logic programming, and action rules.

1. Introduction

Constraint Logic Programming (CLP) defines a family of programming languages that extend Prolog by replacing unification of Herbrand terms with constraint solving over certain domains [11, 26]. Various abstract machines and language constructs have been proposed for implementing constraint solvers. Most languages are designed for a particular domain. For example, there are extended WAMs [2] for constraint solving over finite-domains [1, 9], reals [25], and floating-point intervals [29]. The drawback of these languages is a lack of flexibility and extendibility. CHR (Constraint Handling Rules) [17] is a high-level language for imple-

menting propagation-based constraint solvers. However, a fast compiler for CHR is yet to be implemented. Now, constraint solvers implemented in CHR are an order of magnitude slower than constraint interpreters implemented in low level languages.

B-Prolog [38, 40] provides a powerful language construct, called *action rules*, for programming interactive agents. An action rule specifies a pattern for agents, an action that the agents can carry out, and an event pattern for events that can activate the agents. An agent is like a sub-goal in Prolog that behaves in an event-driven manner. An agent can be suspended when certain conditions on it are satisfied and can be activated to take its actions when certain events are posted. The action rule language combines goal-oriented and event-driven execution models, and is suitable for programming interactive agents such as constraint propagators. As far as constraint propagation is concerned, an agent maintains dynamically a certain level of consistency for a constraint.

The merits of action rules over most other languages are: (1) it is high-level and therefore easy to learn and use; (2) it is flexible and facilitates implementation of various kinds of propagation algorithms; (3) it is efficient because it is compiled; and (4) it provides a convenient platform for end-users to program problem-specific algorithms.

This paper presents constraint solvers implemented in action rules for six domains, namely finite-domains, Boolean, trees, lists, sets, and floating-point intervals. For each domain, the representation scheme of domain variables and sample propagation rules are given.

2. Action Rules in B-Prolog

An *action rule* takes the following form:

```
<Agent> <Condition> {<Event>} '=>' <Action>
```

where `Agent` is an atomic formula that represents a pattern for agents, `Condition` is a sequence of conditions on the agents, `Event` is a set of patterns for events that can activate the agents, and `Action` is a sequence of actions performed by the agents when they are activated.

All conditions in `Condition` must be in-line tests. The event set `Event` together with the enclosing braces is optional. If an action rule does not have any event patterns specified, then the rule is called a *commitment rule*. A set of built-in events is provided for programming constraint propagators and interactive graphical user interfaces. For example, `ins(X)` is an event that is posted when the variable `X` is instantiated and `dom(X,E)` is posted when an inner element `E` is excluded from the domain of the finite-domain variable `X`. A user program can create and post its own events and define agents to handle them. A

user-defined event takes the form of `event(X,T)` where `X` is a variable, called a *suspension variable*, that connects the event with its handling agents, and `T` is a Prolog term that contains the information to be transmitted to the agents. If the event poster does not have any information to be transmitted to the agents, then the second argument `T` can be omitted. The built-in action `post(E)` posts the event `E`.

When an agent is created, the system searches in its definition for a rule whose agent-pattern *matches* the agent and whose conditions are satisfied. This kind of rules is said to be *applicable* to the agent. Notice that since one-directional matching rather than full-unification is used to search for an applicable rule and no variable in the conditions can be instantiated, the agent will remain the same after an applicable rule is found.

The rules in the definition are searched sequentially. If there is no rule that is applicable, the agent will fail. After an applicable rule is found, the agent will behave differently depending on the type of the rule.

If the rule found is a commitment rule in which no event pattern is specified, the actions will be executed. The agent will commit to the actions and a failure of the actions will lead to the failure of the agent. A commitment rule is similar to a clause in concurrent logic languages, but an agent can never be blocked while it is being matched against the agent pattern.

If the rule found is an action rule, the agent will be suspended until it is *activated* by one of the events specified in the rule. When the agent is activated, the conditions are tested *again*. If they are met, the actions will be executed. A failure of any action will cause the agent to fail. The agent does not vanish after the actions are executed, but instead turns to wait until it is activated again. So, besides the difference in event-handling, the action rule "`H,C,E => B`" is similar to the guarded clause "`H :- C | B, H`", which creates a clone of the agent after the action `B` is executed.

Let `post(E)` be the selected sub-goal. After `E` is posted, all agents waiting for `E` will be activated. In practice, for the sake of efficiency, events are postponed until before the execution of the next non-inline call. At a point during execution, there may be multiple events posted that are all expected by an agent. If this is the case, then the agent has to be activated once for each of the events.

There is no primitive for killing agents explicitly. As described above, an agent never disappears as long as action rules are applied to it. An agent vanishes only when a commitment rule is applied to it.

Suspension variables

A suspension variable is a variable to which there are suspended agents and some other information attached. Suspension variables are useful for implementing user-defined domains. The call

```
susp_attach_term(X,T)
```

attaches the term `T` to the variable `X`. The formerly attached term to `X`, if any, will be lost after this operation. This operation is undone automatically upon backtracking. In other words, the originally attached term will be restored upon backtracking. The call

```
susp_attached_term(X,T)
```

gets the current term `T` attached to the variable `X`. In this paper, we use the notation `X^attached` for the term attached to `X`.

Suspension variables are similar to attribute variables [24], but do not rely on goal expansion to define the behaviors associated with them. Whenever a suspension variable `X` is bound to another term, which may be another variable, the event `ins(X)` will be posted. The user can specify the action to be taken after a suspension variable is bound, but not the action to be taken before unification takes place.

The following example illustrates the use of suspension variables:

```
create_fd_variable(X,D) =>
     susp_attach_term(X,D),
     check_member(X,D).

check_member(X,D),var(X),{ins(X)} => true.
check_member(X,D) => member(X,D).
```

This is a simple implementation of finite-domain variables. The agent `check_member(X,D)` is suspended when `X` is a variable. When `X` is instantiated, the agent is activated to check whether the value assigned to `X` is a member of `D`. In a real implementation, unification of two finite-domain variables should be considered as well.

3. Implementing Constraint Solvers in Action Rules

In the CLP family, CLP(X) is a language that supports constraint solving over the domain X. Most CLP systems introduce new operators for expressing constraints rather than extending the unification operator. In this paper, we use self-explainary mathematical symbols for constraints. Operators are usually generic and their interpretation depends

on the types of the constraint expressions. For this reason, the users are required to provide the information about the types of variables.

The type of each variable can be known from its domain declaration or can be inferred from its context. The domain of a variable is declared by a call as follows:

```
V :: D
```

where `V` is a variable and `D` is a range `L..U` of values, where `L` is the lower bound and `U` is the upper bound. One of the bounds can be omitted if it is unknown, but not both. The type of `V` is determined by the bounds. For example, `1..3` denotes a set of integers from 1 to 3, `1..` a set of positive integers, `[]..[_,_,_]` a set of lists of up to three elements, `{}..{a,b,c}` a set of all subsets of `{a,b,c}`, and `0.0..` a set of non-negative floating-point numbers. For finite-domain variables, `D` can be a list of ground terms. So, `V :: [a,b,c]` says that `X` can be `a`, `b`, or `c`.

We only consider propagation-based solvers. Constraint propagation, which is a technique originated in Artificial Intelligence [28, 36] for solving constraint satisfaction problems, works as follows: Whenever the constraint store is changed, e.g., new constraints are added or the domain of a variable in some constraint is updated, it propagates the change to other constraints to attempt to exclude those no-good values from the domains of variables that can never be a part of a solution. Constraint propagation is an iterative procedure that continues until no further change can be made to the constraint store.

For most problems, propagation alone is inadequate for finding a solution, and the *divide-and-conquer* or *relaxation* method is usually necessary for finding a solution. The call

```
indomain(V)
```

finds a value for `V` either by enumerating the values in `V`'s domain or by splitting the domain. After a variable is instantiated, the propagation procedure will be invoked again.

In this section, we describe how to implement in action rules constraint solvers over six different domains, namely finite domain, Boolean, trees, lists, sets, and floating-point intervals.

3.1. CLP(FD)

CLP(FD), the member of the CLP family that supports finite-domain constraints, may be the most successful member in the CLP family. A large number of applications ranging from design, scheduling, to configuration have been developed [15, 37], and many implementation methods have been explored [1, 7, 9, 19, 33, 39].

A finite domain variable is represented as a suspension variable with an attached term of the following form:

```
fd(First,Last,Size,Elms)
```

The arguments refers to, respectively, the *first* element, the *last* element, the number of remaining elements, and a data structure that represents the elements in the domain. The last argument may be a bit vector or a hashtable that tells the status of each element in the domain.

An event will be posted whenever the domain of a variable is updated. For a domain variable `X`, instantiating `X` posts the event `ins(X)`, updating the bound of the domain posts `bound(X)` if the domain contains only integers, and excluding any inner element `E` from the domain posts the event `dom(X,E)`.

Action rules extend delay clauses [39] and can be used to implement various kinds of propagation algorithms for finite-domain constraints. The following shows the implementation of the arc consistency rule for the constraint `X = Y+C` where `X` and `Y` are integer domain variables and `C` is an integer:

```
'X=Y+C_arc'(X,Y,C):-
    'X in Y+C_arc'(X,Y,C),
     C1 is -C,
    'X in Y+C_arc'(Y,X,C1).

'X in Y+C_arc'(X,Y,C),var(X),var(Y),{dom(Y,Ey)} =>
     Ex is Ey+C,
     fd_exclude(X,Ex).
'X in Y+C_arc'(X,Y,C) => true.
```

The propagator `'X in Y+C_arc'(X,Y,C)` maintains arc consistency for `X` in the constraint. Whenever an element `Ey` is excluded from the domain of `Y`, it excludes `Ex`, the counterpart of `Ey`, from the domain of `X`.

3.2. CLP(Boolean)

CLP(Boolean) can be considered as a special case of CLP(FD) [10] where each variable has a domain of two values. We use 0 to denote *false*, and 1 to denote *true*. A Boolean expression is composed of constants (0 or 1), Boolean domain variables, basic relational constraints, and the operators. Since constraints can be operands in a Boolean expression, it is possible to use a Boolean variable to indicate the satisfibility of a constraint. For example, the constraint `(X = Y) <=> B` says that `X` and `Y` are equal iff `B` is equal to 1. This technique, called *reification*, is useful for implementing global constraints such as cardinality constraints.

It is possible to implement various kinds of propagators with different powers in action rules. For example, for the constraint `(X = Y) <=> B`, one simple propagator is to delay the evaluation until either B is ground or both X and Y become ground. A more powerful propagator would add the constraint `B = 0` once the domains of `X` and `Y` are known to be disjoint.

3.3. CLP(Tree)

Prolog can be considered as a CLP language over trees. The unification `T1=T2` finds a valuation, called *unifier*, for the variables in the terms such that `T1` and `T2` become identical after each variable is replaced with its substitute. Prolog does not support disequality constraints over trees. The built-in `T1\=T2` is equivalent to not(T1=T2), which may fail even if `T1` and `T2` represent two different terms. For instance, the query

```
f(X) \= f(Y), X=a, Y=b
```

fails in Prolog. Prolog-II [12] and many others support disequality constraints over trees. These systems delay the evaluation of disequality constraints until the variables in the terms are instantiated sufficiently [6]. Since action rules extend the delay constructs, it is possible to implement disequality constraints over trees in a similar way.

3.4. CLP(List)

CLP(List) hasn't received as much attention as CLP(FD) and now few systems support constraints over lists. Nevertheless, CLP(List) is becoming popular as it is found useful in string processing such as the analysis of bio-sequences [16] and the processing of XML documents. Propagation rules for list constraints are yet to be explored. In Prolog-III, which may be the only CLP language that supports list constraints now, a concatenation constraint is delayed until the lengths of the participating lists are fixed. In [16], an implementation of a pattern language is given that adopts the backtracking algorithm. We propose a solver for CLP(List) that integrates propagation with string pattern matching.

For list domains, the following notations are used. `E1+E2` denotes the concatenation of `E1` and `E2`, `V^[I..J]` the sublist of `V` from the `I`th to the `J`th positions where `I` and `J` can be variables, `V^[I]` the `I`th element, and `|V|` the length of `V`.

A list domain variable `V` can be represented as a suspension variable with an attached term of the following form:

```
list(Length,Value,IndexTable,SuperLists)
```

where the arguments have the following meanings: `Length` is an integer domain variable that indicates the length of `V`. `Value` the prefix of `V` that is already known. `Value` evolves from a variable to a complete list with a fixed length while information about `V` is accumulated. `IndexTable` is a hashtable that facilitates the access of particular list elements. For each element `E` at the `I`th position in `Value`, there is a pair `(I,E)` in `IndexTable`. `SuperLists` represents a list of super lists of which `V` is a sublist. For each super list `SL` of `V`, there is an element `(Start,SL)` in `SuperLists` where `Start` indicates the starting position of `V` in `SL`. This representation of list domains facilitates translating list constraints into finite-domain constraints.

The sublist constraint `A^[I..J] = B` is interpreted as follows: `B` becomes a sublist of `A` whose starting position in `A` is `I` and whose length is equal to `J-I+1`. Whenever an element of `B` becomes sufficiently instantiated, the matching algorithm is invoked to reduce the possible values for `I` and `J`. For example, suppose `A` is the list `[b,c,a,a,d,a]` and `B`'s current value is the incomplete list `[B1,_|_]`. Once `B1` is bound to `a`, the set of possible starting positions for `B` in `A` is narrowed to `[3,4]`.

The concatenation constraint `A+B = C` entails the following: `A` is a sublist of `C` starting at `1`, `B` is a sublist of `C` starting at `|A|+1`, and `|A|+|B| = |C|`.

The call `indomain(V)` searches for a value that is a sublist of all the super lists of `V`. The accumulated constraints on `V` are used to guide the string pattern matching algorithm.

3.5. CLP(Set)

CLP(Set) is a member in the CLP family where each variable can have a set as its value. We consider only finite sets of ground terms. CLP(Set) is well suited for some optimization problems that are hard to model in CLP(FD) [18, 3]. CLP(Set) is also found useful in some other application areas such as program analysis [22] and computational linguistics [32]. Systems that support set constraints include Eclipse [18], Mozart Oz [31] and the ILOG solver [34].

One of the key issues in implementing set constraints is how to represent set domains. Let `N` be the size of the universal set. Then the domain of a set variable has 2^N sets. Because the domain size is exponential in the size of the universal set, it is unrealistic to enumerate all the values in a domain and represent them explicitly. One method is to use intervals to represent set domains [18, 34]. We adopt the same method, but instead of using constant sets to represent the bounds we use finite-domain variables [41].

A set-domain variable `V` is represented as a suspension variable with an attached term of the following form:

```
set(Low,Up,Card,Univ)
```

where `Low` and `Up` are two finite-domain variables that represent respectively the lower and upper bounds, `Card` is another finite-domain variable that represents the cardinality, and `Univ` is a term that represents the universal set.

The representation scheme for set domains facilitates the manipulation of bounds. It takes constant time to add an element to or remove an element from a set domain. Constraint propagators update the bounds when related domains are updated. For example, the following shows one of the rules for dynamically maintaining the interval consistency for the subset constraint `A` $\subseteq$ `B`:

```
propagate_inclusion_low(A,B),
      A^attached = set(ALow,AUp,ACard,AUniv),
      {dom(ALow,E)} =>
      add(E,B).
```

Whenever an element is added to the lower bound of `A`, it adds the element to `B`.

3.6. CLP(F-Interval)

Interval arithmetic, which is an arithmetic defined on sets of intervals, has become a rich source of methods for scientific computing [27]. Cleary first introduced interval arithmetic into Prolog [8]. Since then, several systems have been implemented (e.g., [4, 5, 23, 29]). In BNR-Prolog [5], a propagation method similar to the one used in CLP(FD) is used to reduce the set of values for variables. In Newton [4], a propagation method inspired by the Newton's root finding method is used to speed-up the convergence process. All the systems require the modification of the underlying abstract machines. In this subsection, we illustrate how to implement interval arithmetic with action rules.

An interval domain variable `V` is represented as a suspension variable with the following attached term `float(Low,Up)`, where `Low` and `Up` are floating-point numbers that denote respectively the lower and upper bounds of the domain. Whenever a bound is updated, the event `bound(V)` is posted.

Just as for integer constraints, there are many different ways of implementing propagation rules. For instance, the following propagator maintains interval consistency on `X` for the constraint `X=Y+Z`:

```
x_is_y_plus_z(X,Y,Z),{bound(Y),bound(Z)} =>
```

```
Low is min(Y)+min(Z),
Up is max(Y)+max(Z),
X in Low..Up.
```

The call `X in Low..Up` narrows the bounds of `X` if the current lower bound is less than `Low` or the current upper bound is greater than `Up`.

The interval arithmetic is a rigorous theory that provides the definitions of all the computable functions in the floating-point arithmetic. All the definitions can be translated into action rules.

4. Concluding Remarks

This paper presents six constraint solvers in action rules, a language construct available in B-Prolog. These solvers illustrate the power of the language. The solvers for finite-domains, Boolean, trees and sets have been incorporated into B-Prolog, and the solvers for lists and floating-point intervals are to be included in B-Prolog in the future. The results are very encouraging. The finite-domain solver is one of the fastest. It is about four times as fast as the solver in Sicstus Prolog and is even faster than GNU-Prolog, a native compiler that has the reputation as the fastest CLP(FD) system. The reader is referred to *www.probp.com/fd_evaluation.htm* for the comparison results. The set constraint solver is significantly faster than Conjunto in Eclipse. The high performance is attributed not only to the fast finite-domain constraint solver but also to the new representation scheme for domains that facilitates updates of bounds. Further work needs to be done on improving the compiler and deploying domain-specific optimization techniques in the solvers.

References

[1] A. Aggoun and N. Beldiceanu: Overview of the CHIP Compiler System, In *Proc. of the 8th International Conference on Logic Programming*, pp.775-789, MIT Press, 1991.

[2] H. Ait-Kaci : *Warren's Abstract Machine*, The MIT Press, 1991.

[3] F. Azevedo and P. Barahona: Modeling Digital CIrcuits Problems with Set Constraints, *Proc. Computational Logic - CL 2000*, LNAI 1861, 2000.

[4] F. Benhamou, D. McAllester, P. van Hentenryck, CLP(Intervals) Revisited, *Proc. International Symposium on Logic Programming*, pp.124-138, 1994.

[5] F. Benhamou and W.J. Older, Applying Interval Arithmetic to Real, Integer, and Boolean Constraints, *Journal of Logic Programming*, 1996.

[6] M. Carlsson: Freeze, Indexing, and other Implementation Issues in the WAM, *Proc. 4th International Conference on Logic Programming*, 40-58,1987.

[7] M. Carlsson, G. Ottosson, and B. Carlson: An Open-ended Finite Domain Constraint Solver, *Proc. Programming Languages and Logic Programming*, pp.191-206, 1997.

[8] J.G. Cleary, Logical Arithmetic, *Future Generation Computing*, Vol. 2, pp.125-149, 1987.

[9] P. Codognet and D. Diaz: Compiling Constraints in clp(FD), *Journal of Logic Programming*, 27(3), pp.185-226, 1996.

[10] P. Codognet and D. Diaz: Boolean Constraints Solving Using clp(FD), *Journal of Automatic Theorem Proving*, 1996.

[11] J. Cohen: Constraint Logic Programming Languages, *Communications of ACM*, Vol.33, No.7, pp.52-68, 1990.

[12] A. Colmerauer.: Equations and In-equations on Finite and Infinite Trees, *Proc. of the International Conference on Fifth Generation Computer Systems (FGCS'84)*, ICOT, 85-99, 1984.

[13] A. Colmerauer.: An Introduction to Prolog-III, *Communications of ACM*, Vol.33, No.7, 1990.

[14] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier: The Constraint Logic Programming Language CHIP, In *Proceedings of the Fifth Generation Computer Systems*, pp.693-702, ICOT, 1988.

[15] M. Dincbas, H. Simonis, P. van Hentenryck: Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1), pp.75-93, 1990.

[16] I. Eidhammer, D. Gilbert, I. Jonassen, and M. Ratnayake: A Constraint Based Structure Description Language for Biosequences, *Constraints, An International Journal*, 2001.

[17] T.W. Fruhwirth: Theory and Practice of Constraint Handling Rules, *Journal of Logic Programming*, Vol.37, pp.95-138, 1998.

[18] C. Gervet: Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language, *Constraints, An International Journal*, vol.1, pp.191-246, 1997.

[19] W. Harvey and P.J. Stuckey: Improving Propagation by Changing Constraint Representation, *Constraints, An International Journal*, to appear.

[20] P. van Hentenryck: *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[21] P. van Hentenryck and V. Saraswat (eds.), Strategic Directions in Constraint Programming, *ACM Computing Survey*, Vol.28, No.4, pp.701-728, 1996.

[22] N. Hentze and J. Jaffar: Set Constraints and Set-Based Analysis, *Proc. Principle and Practice of Constraint Programming*, A. Borning, ed., LNCS, pp.281-298, Springer-Verlag, 1994.

[23] T. Hickey, Q. Ju, and M.H. Van Emden, Interval arithmetic: From principles to implementation, *Journal of the ACM*, Vol. 48 , No. 5, pp.1038-1068, 2001.

[24] C. Holzbaur: Meta-structures Vs. Attribute Variables in the Context of Extensible Unification, *Proc. PLLP'92*, LNCS 631, pp.260-268, 1992.

[25] J. Jaffar, S. Michaylov, P. Stuckey and R. Yap, An Abstract Machine for CLP(R), *Proc. SIGPLAN'92 Conf. on Programming Language Design & Implementation (PLDI)*, San Francisco, 1992.

[26] J. Jaffar and M. J. Maher, Constraint Logic Programming: A Survey, *Journal of Logic Programming*, 1994.

[27] R.B. Kearfott and V. Kreinovich: *Applications of Interval Computations*, Kluwer, 1996.

[28] V. Kumar: Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*, 13(1), pp.32-44, 1992.

[29] J.H.M Lee and T.W. Lee: A WAM-based Abstract Machine for Interval Constraint Logic Programming, *Proc. IEEE International Conference on Tools with Artificial Intelligence*, pp.122-128, 1994.

[30] A.K. Mackworth: Constraint Satisfaction, In *Encyclopedia of Artificial Intelligence*, John Wiley & Sons, 205–211, 1986.

[31] T. Muller: *Constraint Propagation in Mozart*, PhD Thesis, Programming Systems Lab, Universit Saarlandes, *http://www.ps.uni-sb.de/~ tmueller/thesis/*, 2001.

[32] L. Pacholski and A.Podelski: Set Constraints: A Pearl in Research on Constraints, In *Proc. 3rd International Conference on Constraint Programming*, 1997.

[33] J.F. Puget and M. Leconte: Beyond the Glass Box: Constraints as Objects, *Proc. International Logic Programming Symposium*, pp.513-527, 1995.

[34] J.F. Puget: Finite Set Intervals, in *Proc. Workshop on Set Constraints*, CP'96, 1996.

[35] E. Shapiro: The Family of Concurrent Logic Programming Languages, *ACM Comput. Surveys*, vol.21, no.3 pp.412-510, 1989.

[36] E. Tsang: *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[37] M.G. Wallace, Practical Applications of Constraint Programming, *Constraints Journal*, vol.1 no.1, Kluwer, 1996.

[38] N.F. Zhou: Parameter Passing and Control Stack Management in Prolog Implementation Revisited, *ACM Transactions on Programming Languages and Systems*, 18(6), 752-779, 1996.

[39] N.F. Zhou: A High-Level Intermediate Language and the Algorithms for Compiling Finite-Domain Constraints, *Proc. Joint International Conference and Symposium on Logic Programming*, 70-84, MIT Press, 1998. A revised version is available from: *http://www.sci.brooklyn.cuny/~ zhou/pappers/arule.pdf*.

[40] N.F. Zhou: *B-Prolog User's Manual (Version 6.1)*, *http://www.probp.com*, 2001.

[41] N.F. Zhou and J.Schimpf: Implementation of Propagation Rules for Set Constraints Revisited, submitted for publication, 2002.