2002

TR-2002010: Programming Finite-Domain Constraint Propagators in Action Rules

Neng-Fa Zhou

Programming Finite-Domain Constraint Propagators in Action Rules

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

Abstract

In this paper, we propose a new language construct, called *action rules*, and describe how various propagators for finite-domain constraints can be implemented in it. 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. Action rules combine the goal-oriented execution model of logic programming with the event-driven execution model. This hybrid execution model facilitates programming constraint propagators. A propagator for a constraint is an agent that maintains the consistency of the constraint and is activated by the updates of the domain variables in the constraint. The action rule language has a much stronger description power than *indexicals*, the language widely used in current finite-domain constraint systems, and is flexible for implementing not only the interval-consistency but also the arc-consistency algorithms. As examples, we present the implementation of a weak arc-consistency algorithm for the `all_distinct` constraint and a hybrid algorithm for n-ary linear equality constraints. B-Prolog has been extended to accommodate action rules. Benchmarking shows that the performance of B-Prolog as a CLP(FD) system is comparable with the fastest systems available now.

1 Introduction

CLP(FD), the constraint logic programming language over finite-domains, has been proved effective for solving a large number of real-life optimization problems [8]. The key operation employed in CLP(FD) is called *constraint propagation* [14, 22], which uses constraints actively to prune search spaces as follows: whenever a variable changes, i.e., the variable has been instantiated or its domain has been updated, the domains of all the remaining variables are filtered to contain only those values that are consistent with this variable. There may exist different propagation rules for a constraint depending on the level of consistency to be achieved. Constraint propagation has been employed to solve not only constraints over finite-domains but also constraints over trees, lists, sets, floating-point numbers, and many other domains [13].

In early CLP(FD) systems, such as the CHIP system [7], constraints are interpreted rather than compiled. Constraints are first transformed into canonical-form terms and are then executed by an interpreter that performs, among other things, constraint propagation. The propagation procedure adopted is general enough

for handling all types of constraints. Learning from the experience of compiling Prolog programs into the Warren Abstract Machine (WAM) [2], a former research group at ECRC extended the WAM for compiling CLP(FD) [1]. The CHIP compiler compiles constraints into low level instructions such that different specialized propagation procedures are used for different types of constraints. This black-box approach has proved problematic because it is too complicated and lacks flexibility and extendibility. The extended WAM in the CHIP system has over 100 instructions for compiling finite-domain constraints alone [1]!

A language construct, called *indexicals*, has been quite popular as an intermediate language for compiling finite-domain constraints. The language was first proposed by Hentenryck et al [24] and then popularized by [4]. This language is also adopted by other systems [3, 21]. An indexical is a primitive constraint in the form of X *in* r, where X is a domain variable and r specifies the range for X. For each indexical, a propagation procedure specific to it is used. Indexicals are claimed to be a glass-box approach to compiling constraints in contrast with the black-box CHIP compiler. Nevertheless, as the delaying mechanism is embedded in range expressions, indexicals are not as open as claimed. Indexicals can be used to compile arithmetic constraints, but are too weak to be used to program many other kinds of propagators.

CHR (Constraint Handling Rules) [10] may currently be the most powerful implementation language for constraints. It can be used to program not only constraint propagators but also constraint reasoning rules. CHR has been implemented and integrated with Eclipse, Sicstus, and Oz. CHR resembles a production system. In CHR, the left-hand side of a rule specifies a pattern of constraints in the constraint store and the right-hand side specifies new constraints to replace those on the left-hand side or to be added into the store. The left-hand side of a rule may have multiple constraint patterns. This feature is helpful for reasoning about the constraint store. For example, $A > B \ \& \ B > C \rightarrow A > C + 1$ is a CHR rule that generates the constraint $A > C + 1$, which is helpful albeit redundant. The strong description power, however, is not offered without cost. For CHR, a sophisticated matching algorithm is needed to match constraint patterns against the constraint store. Now, constraint solvers implemented in CHR are still an order of magnitude slower than constraint interpreters implemented in C [12].

This paper proposes a new language construct, called *action rules*, that can be used to program event-handling in general and constraint propagation in particular. 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 subgoal in Prolog but behaves in an event-driven manner. An agent can be suspended when certain conditions on it are satisfied and can be activated when certain events are posted. Action rules are an extension of delay constructs such as delay clauses [15] that allows for the descriptions of not only delay conditions but also activating events and actions. The syntax and semantics of action rules will be described in Section 3.

The focus of this paper is on how to implement various propagators for finite-domain constraints in action rules. A propagator for a constraint is an agent that maintains the consistency of the constraint and is activated by the updates

of the domain variables in the constraint. In Section 4, we present propagators for binary, non-binary, and the global constraint `all_distinct`. Action rules are more expressive than indexicals. Some of the propagators presented such as the one for maintaining arc-consistency for binary equality constraints and the one for maintaining weak arc-consistency for `all_distinct` cannot be implemented in indexicals as efficiently.

B-Prolog has been extended to accommodate action rules and several constraint solvers including the ones over finite-domain, Boolean, trees, and sets have been developed in action rules. Section 5 compares the performance of the finite-domain solver of B-Prolog with GNU-Prolog, a state-of-the-art implementation of CLP(FD). The benchmarking results show that B-Prolog is faster than GP for most of the benchmarks and on average as well.

Readers are assumed to be familiar with logic programming and constraint satisfaction, but the knowledge about abstract machines is not a prerequisite. In Section 2, we define some preliminary terms and concepts about CLP(FD) and constraint propagation. Readers are referred to [23] and [14] for the details. Some remarks on the implementation of action rules are given in Section 5. A detailed description of the implementation techniques is beyond the scope of this paper. Interested readers are referred to [25] for the abstract machine adopted in B-Prolog and to [28] for the extended memory architecture for supporting agents.

2 Preliminaries

2.1 CLP(FD)

CLP(FD) [23] is an extension of Prolog that supports built-ins for specifying domain variables, constraints, and strategies for instantiating variables.

The domains of variables are declared as follows:

```
Vars in D
```

where `Vars` is a variable or a list of variables, and `D` is a list of ground terms or a range between two integers $l..u$. A domain variable is normally represented as a Prolog variable with attributes. A CLP(FD) system provides primitives for accessing and updating attribute values.

A CLP(FD) system provides *equality* ($=$), *disequality* ($\neq$), and *inequality* constraints. In addition, a CLP(FD) system also provides some other constraints such as global constraints. The global constraint `all_distinct(L)` ensures that the elements in the list `L` must be all different.

2.2 Constraint propagation

Constraint Propagation [23] is a key operation employed in CLP(FD) systems for maintaining the consistency of constraints. The basic idea of constraint propagation is to activate the propagators of constraints whenever some updates occur to the domain variables in the constraints. Propagating the updates to other variables may result in the shrink of the domains of the variables or the instantiation of the variables.

There are different levels of consistency for constraints [14]. We define below three levels of consistency needed in this paper, namely *node*, *interval* and *arc*, and define the propagators that maintain them.

A unary constraint *p(X)* where X has the the domain D is said to be *node-consistent* if for any element x in D *p(x)* is satisfied.

$$\forall_{x \in D} p(x)$$

For example, for the equality constraint $X = Y + 1$, when X is instantiated to 3, Y must be instantiated to 2 to make the constraint node-consistent. As another example, for the disequality constraint $X \neq Y$, when X is instantiated to 3, 3 must be excluded from the domain of Y to make the constraint node-consistent. The propagation rule that maintains node-consistency is called *forward checking*. A propagator for a constraint that performs forward checking is activated whenever the constraint becomes unary.

Let C be an equality constraint $f(X_1, X_2, ..., X_n) = 0$ where X_i is defined over the domain D_i $(i = 1, ..., n)$. Suppose $X_i = g_i(X_1, ..., X_{i-1}, X_{i+1}, ..., X_n)$ and the functions *min* and *max* are defined as follows:

$$min(g_i(X_1, ..., X_{i-1}, X_{i+1}, ..., X_n)) = min\{g_i(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n) | x_i \in D_i\}$$

$$max(g_i(X_1, ..., X_{i-1}, X_{i+1}, ..., X_n)) = max\{g_i(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n) | x_i \in D_i\}$$

The constraint C is interval consistent w.r.t. X_i if:

$$\forall_{x \in D_i}(min(g_i(X_1, ..., X_{i-1}, X_{i+1}, ..., X_n)) \leq x \leq max(g_i(X_1, ..., X_{i-1}, X_{i+1}, ..., X_n)))$$

To make the constraint interval consistent w.r.t. X_i, we have to exclude all the elements from D_i that are not in the range. The constraint C is interval consistent if C is interval consistent w.r.t. all the variables. For example, the constraint $X = Y + 1$, where X and Y have the domain 1..5, is not interval-consistent. To make the constraint interval-consistent, we have to exclude 1 from the domain of X and 5 from the domain of Y. Propagators for maintaining interval consistency are activated whenever a bound of a variable is updated or whenever a variable is instantiated. The definition can be easily extended to inequality constraint $f(X_1, X_2, ..., X_n) \geq 0$.

Consider a binary constraint $p(X, Y)$ where X and Y are defined over the domains D_x and D_y, respectively. The constraint is said to be *arc-consistent* w.r.t. X if for any element in D_x there exists a supporting element in D_y such that the constraint is satisfied:

$$\forall_{x \in D_x} \exists_{y \in D_y} p(x, y)$$

Similarly, the constraint is arc-consistent w.r.t. Y if for any element in D_y there exists a supporting element in D_x such that the constraint is satisfied:

$$\forall_{y \in D_y} \exists_{x \in D_x} p(x, y)$$

The constraint is arc-consistent if it is arc-consistent w.r.t. both X and Y. For example, the equality constraint $X = Y + 1$ ($X \in \{2, 4, 5\}$, $Y \in \{1..4\}$) is not

arc-consistent since there is no element in the domain of X that supports 2 in the domain of Y. To make the constraint arc-consistent, we must exclude 2 from the domain of Y. Propagators for maintaining arc-consistency are triggered whenever changes occur to the domain of a variable. It is very costly to maintain arc-consistency for arbitrary constraints. For this reason, some CLP(FD) systems maintain arc-consistency only for binary constraints and many others do not consider arc-consistency at all.

3 Action Rules

The action rule language is designed for programming active agents. An agent is like a subgoal in Prolog but behaves in an event-driven manner. In this section, we describe the syntax and operational semantics of action rules and also give the set of events offered for programming constraint propagators.

3.1 Syntax

An *action rule* takes the following form:

```
<Agent> <ConditionSeq> {<EventSet>} '=>' <ActionSeq>
```

where `Agent` is an atomic formula that represents a pattern for agents, `ConditionSeq` is a sequence of conditions on the agents, `EventSet` is a set of patterns for events that can activate the agents, and `ActionSeq` is a sequence of actions performed by the agents when they are activated. Conditions, event patterns, and actions are all atomic formulas. In a sequence, the delimiter ',' is used to separate the elements.

All conditions in `ConditionSeq` must be in-line tests. The event set `EventSet` 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. 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`.

An *agent definition* consists of a sequence of rules, and a *program* consists of a sequence of *agent definitions*. Agents correspond to subgoals in Prolog, and agent definitions correspond to predicates. In this paper, we use the term *agents* for subgoals that can be suspended and activated.

3.2 Examples

The following defines an agent that echoes the messages sent to it by event posters.

```
echo_agent(X), {event(X,Message)} => write(Message).
```

The following query,

```
echo_agent(Ping), echo_agent(Pong),
post(event(Ping,ping)), post(event(Pong,pong))
```

creates two echo agents named `echo_agent(Ping)` and `echo_agent(Pong)`, and activate them by posting two events. The event `event(Ping,ping)` activates the agent `echo_agent(Ping)`, and the event `event(Pong,pong)` activates the agent `echo_agent(Pong)`.

The following defines the freeze predicate in Prolog-II [5].

```
freeze(X,G), var(X), {ins(X)} => true.
freeze(X,G) => call(G).
```

The primitive `freeze(X,G)` is logically equivalent to `call(G)` but the execution of `G` is delayed until `X` is instantiated to a non-variable term. The agent `freeze(X,G)` is suspended waiting for an event `ins(X)` when `X` is a variable. When an event `ins(X)` is posted, the condition `var(X)` is tested *again*. If it succeeds, then the action `true` is executed and the agent becomes suspended again. As long as `X` is a free variable, the agent `freeze(X,G)` will be suspended. Only when `X` becomes a non-variable term, can the second rule be applied.

3.3 Operational Semantics

The operational semantics of action rules is better formulated as a system of transition states. Each state is a pair $<A, S>$, where A is a *sequence* of actions and S is a *set* of pairs (α, r) where α is a suspended agent and r is an action rule to be tried when the agent is activated. An initial state is $<A_0,\phi>$ where A_0 is a sequence of actions given by the user and an ending state is $<\phi,S>$ where the sequence of actions is empty.

When an agent is created, the system searches in its definition for a rule whose agent-pattern *matches* the agent and whose condition is satisfied. This kind of rule is said to be *applicable* to the agent. Formally, an action rule "$H, C, E => B$" or a commitment rule "$H, C => B$" is applicable to an agent α, if "$H\theta = \alpha \wedge C\theta$" is satisfied. Notice that since one-directional matching rather than full-unification is used to search for an applicable rule and in the condition no variable 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.

Application of commitment rules

If the rule found is a commitment rule "$H, C => B$" in which no event pattern is specified, then the action B will be added to the action sequence.

$\frac{<\alpha.A,S>}{<B\theta+A,S>}$ (application of commitment rule)

The agent will commit to the action and a failure of the action will lead to the

failure of the agent.

A commitment rule is similar to a guarded clause in concurrent logic languages [19], but an agent can never be blocked while it is being matched against an agent pattern.

Application of action rules

If the rule found is an action rule "$r : H, C, \{E\} => B$", the agent will be suspended until it is *activated* by one of the events in E.

$\frac{<\alpha.A,S>}{<A,\{(\alpha,r\theta)\}\cup S>}$ (suspension rule)

Notice that the instance of the rule $r\theta$ rather than the original rule is memorized. So when α is activated, it is unnecessary to match α against the head of the rule again.

Activating agents

Let $<post(E).A, S>$ be the current state. After E is posted, all the agents waiting for E in S will be activated. For each agent (α, r) in S that is waiting for E, the condition in the rule r is tested *again*. If it succeeds, the action B will be added to the action sequence.

$\frac{<(A,\{(\alpha,r)\}\cup S>}{<B\theta+A,\{(\alpha,r)\}\cup S>}$ (activation rule 1)

The agent does not vanish after the activation, but instead turns to wait until it is activated again. So, aside from the difference in event-handling, the action rule "$H, C, \{E\} => B$" is similar to the guarded clause "$H{:-}C \mid B, H$", which creates a clone of the agent after the action B is executed. Failure of B will cause the agent α to fail.

If the condition of r is not satisfied, then the element (a, r) will be deleted from the suspension set and the agent a will be added to the front of the action sequence.

$\frac{<(A,\{(\alpha,r)\}\cup S>}{<\alpha.A,S>}$ (activation rule 2)

α becomes an action now. When it is selected, an applicable rule will be searched for it. Since once a rule becomes inapplicable to an agent, it can never become applicable again, we only need to search for applicable rules for α from the subsequent rules of r.

Remarks

There is no primitive for killing agents explicitly. 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.

At a certain time, an event may activate several agents. It is up to the implementers of the language to use a strategy to schedule them. Whatever scheduling

strategy is adopted, the user should not rely on the strategy to guarantee the correctness of programs.

In practice, for the sake of efficiency, events are postponed until before the execution of the next non-inline subgoal. 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. In our implementation, the first-come-first-considered strategy is used to schedule events and agents.

3.4 Domain variables and built-in events

A *domain variable* variable is a variable to which there are suspended propagators and some other information attached. A domain variable is represented in B-Prolog as a record that has the following fields:

`ref`	reference to the value
`type`	type of the domain
`min`	minimum element in the domain
`max`	maximum element in the domain
`size`	number of elements that remain in the domain
`ins_cs`	list of propagators to be executed when the variable is instantiated
`minmax_cs`	list of propagators to be executed when a bound is updated
`dom_cs`	list of propagators to be executed when an inner element is excluded
`elms`	pointer to the bit vector representation of the elements

where `ref` refers to the variable itself if the variable is not instantiated, and `elms` is a pointer to a bit vector that represents the elements. When the domain is an interval without holes, then no bit vector is necessary and `elms` is a null pointer.

An event is posted whenever the domain of a variable is updated. For a domain variable `X`, instantiating `X` posts the event `ins(X)`, updating the low or upper bound of the domain posts the event `minmax(X)`, and excluding an inner element `E` from the domain posts the event `dom(X,E)`. Notice that the event `dom(X,E)` is not posted when the bounds of the domain of `X` are updated. This implies that a propagator that maintains arc-consistency has to handle not only `dom(X,E)` events but also `minmax(X)` and `ins(X)` events.

Each event on a domain variable activates its corresponding list of propagators. The event `ins(X)` activates the propagator list `ins_cs` of `X`, `minmax(X)` activates the list `minmax_cs`, and `dom(X,E)` activates the list `dom_cs`.

Consider, as an example, how to implement the following indexical:

```
X in min(Y)+min(Z)..max(Y)+max(Z).
```

which ensures that the constraint `X = Y+Z` is interval-consistent w.r.t. `X`.

```
'V in V+V'(X,Y,Z),{ins(Y),minmax(Y),ins(Z),minmax(Z)} =>
      reduce_domain(X,Y,Z).

reduce_domain(X,Y,Z) =>
      L is min(Y)+min(Z), U is max(Y)+max(Z),
      X in L..U.
```

The propagator is activated whenever a bound of `Y` or `Z` is updated or either one is instantiated. The action `reduce_domain(X,Y,Z)` enforces that the domain of `X` be in the range `min(Y)+min(Z)..max(Y)+max(Z)`. The original indexical is equivalent to the following two subgoals:

```
'V in V+V'(X,Y,Z),reduce_domain(X,Y,Z)
```

where `reduce_domain(X,Y,Z)` enforces interval consistency w.r.t. `X` when the constraint is generated.

4 Programming Constraint Propagators in Action Rules

The high description power of action rules opens new ways to implementing constraint propagators. In this section, we implement propagators that maintain node, interval, and arc consistency for binary constraints, a hybrid algorithm for non-binary constraints, and a week arc-consistency algorithm for the global constraint `all_distinct`.

4.1 Binary constraints

We consider how to implement propagators for the binary constraint `A*X = B*Y+C`, where `X` and `Y` are domain variables, `A` and `B` are positive integers, and `C` is an integer of any kind. Similar propagators can be implemented for other types of binary constraints.

4.1.1 Forward checking

Recall that forward checking enforces node-consistency. The following shows a propagator that performs forward checking for the binary constraint.

```
'aX=bY+c'(A,X,B,Y,C) =>
      'aX=bY+c_forward'(A,X,B,Y,C).

'aX=bY+c_forward'(A,X,B,Y,C),var(X),var(Y),{ins(X),ins(Y)} => true.
'aX=bY+c_forward'(A,X,B,Y,C),var(X) =>
       T is B*Y+C, X is T//A, A*X=:=T.
'aX=bY+c_forward'(A,X,B,Y,C) =>
       T is A*X-C, Y is T//B, B*Y=:=T.
```

The operation `op1//op2`, which is equivalent to `truncate(op1/op2)`, gives the integer quotent of the division. When both `X` and `Y` are variables, the propagator is suspended. When either variable is instantiated, the propagator computes the value for the other variable.

4.1.2 Interval-consistency

The following propagator, which extends the forward-checking propagator, maintains interval-consistency for the constraint.

```
'aX=bY+c'(A,X,B,Y,C) =>
      'aX=bY+c_reduce_domain'(A,X,B,Y,C),
      'aX=bY+c_forward'(A,X,B,Y,C),
      'aX=bY+c_interval'(A,X,B,Y,C).
```

The subgoal `'aX=bY+c_reduce_domain'(A,X,B,Y,C)` preprocess the constraint to make it interval-consistent when the constraint is generated.

```
'aX=bY+c_reduce_domain'(A,X,B,Y,C) =>
      'aX in bY+c_reduce_domain'(A,X,B,Y,C),
      MC is -C,
      'aX in bY+c_reduce_domain'(B,Y,A,X,MC).

'aX in bY+c_reduce_domain'(A,X,B,Y,C) =>
      L is (B*min(Y)+C) /> A,
      U is (B*max(Y)+C) /< A,
      X in L..U.
```

The operation `op1 /> op2` returns the lowest integer that is greater than or equal to the quotient of `op1` by `op2` and the operation `op1 /< op2` returns the greatest integer that is less than or equal to the quotient. The arithmetic operations must be sound to make sure that no solution is lost. For example, the minimum times any positive integer remains the minimum.

The subgoal `'aX=bY+c_interval'(A,X,B,Y,C)` maintains interval-consistency for the constraint.

```
'aX=bY+c_interval'(A,X,B,Y,C) =>
      'aX in bY+c_interval'(A,X,B,Y,C),  % reduce X when Y changes
      MC is -C,
      'aX in bY+c_interval'(B,Y,A,X,MC). % reduce Y when X changes

'aX in bY+c_interval'(A,X,B,Y,C),var(X),var(Y),{minmax(Y)} =>
      'aX in bY+c_reduce_domain'(A,X,B,Y,C).
'aX in bY+c_interval'(A,X,B,Y,C) => true.
```

Notice that the action `'aX=bY+c_reduce_domain'(A,X,B,Y,C)` is executed only when both variables are free. If either one turns to be instantiated, then the forward-checking rule will take care of that situation.

4.1.3 Arc-consistency

The following propagator, which extends the one shown above, maintains arc-consistency for the constraint.

```
'aX=bY+c'(A,X,B,Y,C) =>
      'aX=bY+c_reduce_domain'(A,X,B,Y,C),
      'aX=bY+c_forward'(A,X,B,Y,C),
      'aX=bY+c_interval'(A,X,B,Y,C),
      'aX=bY+c_arc'(A,X,B,Y,C).
```

```
'aX=bY+c_arc'(A,X,B,Y,C) =>
     'aX in bY+c_arc'(A,X,B,Y,C),  % reduce X when Y changes
     MC is -C,
     'aX in bY+c_arc'(B,Y,A,X,MC). % reduce Y when X changes

'aX in bY+c_arc'(A,X,B,Y,C),var(X),var(Y),{dom(Y,Ey)} =>
     T is B*Ey+C,
     Ex is T//A,
     (A*Ex=:=T -> exclude(X,Ex);true).
'aX in bY+c_arc'(A,X,B,Y,C) => true.
```

Whenever an element `Ey` is excluded from the domain of `Y`, the propagator `'aX in bY+c_arc'(A,X,B,Y,C)` is activated. If both `X` and `Y` are variables, the propagator will exclude `Ex`, the counterpart of `Ey`, from the domain of `X`. Again, if either `X` or `Y` becomes an integer, the propagator does nothing. The forward checking rule will take care of that situation.

4.2 Non-binary Constraints

In indexical-based CLP(FD) systems, constraints are split into indexicals that contain no more than three variables. This algorithm has several advantages. First, it generates linear-size code. Second, indexicals can be implemented in a low-level language to achieve better performance. Third, information propagation can be restricted to only those constraints for which the domains have the possibility to be reduced [4]. For example, consider the two tenary constraints `T1 = X1+X2` and `T1+X3+X4 = 0`. If `T1 = X1+X2` is activated by an update of `X1`, as long as the shared variable `T1` does not change the other constraint needs not be activated. The disadvantages of this algorithm are that new domain variables have to be introduced and the granularity of constraints becomes smaller and thus context-switching becomes more costly. In B-Prolog, each domain variable takes at least 10 words, letting alone the space for the constraints and data structures for the elements. The space overhead cannot be neglected when the number of variables is large.

The high description power of action rules opens new ways to compiling non-binary constraints. We present two algorithms. One is called *unite*, which adopts one propagator for each constraint to maintain the interval-consistency. The other one, called *hybrid*, maintains interval-consistency when the constraint contains more than two variables and maintains arc-consistency when the constraint turns into binary.

4.2.1 Unite: use one propagator for each constraint

Let $A_1 * X_1 + \ldots + A_n * X_n + C = 0$ be an n-ary constraint where each A_i(i=1,...,n) is a non-zero integer and each X_i is a domain variable or an integer. The propagator for the constraint takes the following form:

```
'A1*X1+...+An*Xn+C=0'(C,A1,A2,...,An,X1,X2,..,Xn),
      {ins(X1),minmax(X1),...,ins(Xn),minmax(Xn)}
=>
      reduce the domains of X1,..,Xn.
```

In the action, attempts are made to reduce the low and upper bounds of the domain of every variable.

To facilitate the generation of the code for reducing domains, the compiler splits the expression $A_1 * X_1 + \ldots + A_n * X_n + C$ into the following list of sub-expressions each of which contains at most three variables:

$T_0 = C$,
$T_1 = T_0 + A_1 * X_1$,
$T_2 = T_1 + A_2 * X_2$,
...
$T_n = T_{n-1} + A_n * X_n$

The generated reducer first computes the low and upper bounds of the temporary variables by propagating information forward from T_0 to T_n. The low and upper bounds of T_i are computed from those of T_{i-1} and $A_i * X_i$ $(i = 1, \ldots, n)$. After that, the reducer propagates information backward from T_n to T_1. For each tuple $T_i = T_{i-1} + A_i * X_i$, the new bounds of T_{i-1} and X_i are computed from those of T_i.

For example, the following shows the propagator generated for the constraint `X1+X2+X3+C = 0`.

```
'X1+X2+X3+C=0'(C,X1,X2,X3)
      {ins(X1),minmax(X1),ins(X2),minmax(X2),
       ins(X3),minmax(X3)}
=>
      'X1+X2+X3+C=0_reducer'(C,X1,X2,X3).

'X1+X2+X3+C=0_reducer'(C,X1,X2,X3) =>
      Lt1 is C+min(X1), Ut1 is C+max(X1),        % T1 = C+X1
      Lt2 is Lt1+min(X2), Ut2 is Ut1+max(X2),    % T2 = T1+X2
      Lt3 is Lt2+min(X3), Ut3 is Ut2+max(X3),    % T3 = T2+X3
      Lt3 =< 0, Ut3 >= 0,
      %
      NewLx3 is 0-Ut2, NewUx3 is 0-Lt2,      % T3 = T2+X3
      X3 in NewLx3..NewUx3,
      NewLt2 is 0-max(X3), NewUt2 is 0-min(X3),
      %
      NewLx2 is NewLt2-Ut1, NewUx2 is NewUt2-Lt1,% T2 = T1+X2
      X2 in NewLx2..NewUx2,
      NewLt1 is NewLt2-max(X2), NewUt1 is NewUt2-min(X2),
      %
      NewLx1 is NewLt1-C, NewUx1 is NewUt1-C,    % T1 = C+X1
      X1 in NewLx1..NewUx1,
      NewLt1-max(X1) =< C, NewUt1-min(X1) >= C.
```

The advantage of this algorithm is that only one propagator is used for each constraint whose code size is linear in the number of variables in the constraint. The weakness of this algorithm is that the reducer is not fast. Whenever a variable is instantiated or a variable's bound is updated, the reducer tries to reduce the domains of all the variables including the seed variable that triggers the propagator.

4.2.2 Hybrid: combining interval and arc consistency algorithms

For a non-binary constraint, it is too expensive to maintain arc-consistency. One practical strategy is to maintain interval-consistency while there are multiple variables in the constraint and to maintain arc-consistency when the constraint turns into binary. The following shows the propagator for the linear non-binary constraint `A1*X1+...+An*Xn+C = 0`.

```
'A1*X1+...+An*Xn+C=0'(C,A1,A2,...,An,X1,X2,..,Xn),
      n_vars_gt([X1,...,Xn],2),
      {ins(X1),minmax(X1),...,ins(Xn),minmax(Xn)}
=>
      reduce domains of X1,..,Xn to achieve interval-consistency.
'A1*X1+...+An*Xn+C=0'(C,A1,A2,...,An,X1,X2,..,Xn) =>
      nary_to_binary([C,A1,X2,A2,X2,...,An,Xn],NewC,B1,B2,Y1,Y2),
      call_binary_constraint_propagator(NewC,B1,Y1,B2,Y2).
```

The propagator is activated whenever any variable is instantiated or its bound is updated. When `n_vars_gt([X1,...,Xn],2)` succeeds, i.e. when there are multiple variables in the constraint, the domains are reduced to make the constraint interval-consistent. When the constraint becomes binary, the condition `n_vars_gt([X1,...,Xn],2)` fails and the second rule will be tried. The subgoal `nary_to_binary` transforms the constraint into the binary constraint `B1*Y1+B2*Y2+NewC = 0`, and the next subgoal invokes an appropriate propagator for the binary constraint. In the real implementation, the two built-ins `n_vars_gt` and `nary_to_binary` do not take the constraint as an argument but instead access the constraint parent subgoal.

4.3 Propagators for `all_distinct(L)`

The constraint `all_distinct(L)` holds if the variables in `L` are pair-wisely different. One naive implementation method for this constraint is to generate binary disequality constraints between all pairs of variables in `L`. This implementation has two problems: First, the space required to store the constraints is quadratic in the number of variables in `L`; Second, splitting the constraint into small granularity ones may lose possible propagation opportunities [18]. The goal of this subsection is to illustrate the expressive power of action rules.

4.3.1 A linear-space propagator

To solve the space problem, we define `all_distinct(L)` in the following way:

```
all_distinct(L) => all_distinct(L,[]).

all_distinct([],Left) => true.
all_distinct([X|Right],Left) =>
      outof(X,Left,Right),
      all_distinct(Right,[X|Left]).

outof(X,Left,Right), var(X), {ins(X)} => true.
outof(X,Left,Right) => exclude_list(X,Left),exclude_list(X,Right).

exclude_list(X,[]).
exclude_list(X,[Y|Ys]):- exclude(Y,X),exclude_list(X,Ys).
```

For each variable `X`, let `Left` be the list of variables to the left of `X` and `Right` be the list of variables to the right of `X`. The subgoal `outof(X,Left,Right)` holds if `X` appears in neither `Left` nor `Right`. Instead of generating disequality constraints between `X` and all the variables in `Left` and `Right`, the subgoal `outof(X,Left,Right)` suspends until `X` is instantiated. After `X` becomes an integer, the subgoals `exclude_list(X,Left)` and `exclude_list(X,Right)` exclude `X` from the domains of the variables in `Left` and `Right`, respectively.

There is a propagator `outof(X,Left,Right)` for each element `X` in the list, which takes constant space. Therefore, `all_distinct(L)` takes linear space in the size of `L`. Notice that the two lists `Left` and `Right` are not merged into one bigger list. Or, the constraint still takes quadratic space.

4.3.2 A weak arc-consistency algorithm

In terms of pruning ability, the linear-space propagator is the same as the naive one that splits `all_distinct(L)` into binary disequality constraints. In this subsection, we present a weak arc-consistency algorithm for `all_distinct(L)` and show how to implement it.

For each variable `X` in `L`, let `L-{X}` be the list of elements in `L` but `X`, n be the size of the domain of `X`, and m be the number of variables in `L-{X}` whose domains are subsets of that of `X`. If $m + 1 > n$, then the constraint is unsatisfiable since it is impossible to assign n values to more than n variables such that each variable gets a different value. If $m + 1 = n$, then for each value v in `X`'s domain, exclude v from the domains of all the variables whose domains are not subsets of that of `X`.

Consider the following query,

```
X in {1,2}, Y in {1,2}, Z in {1,2}, all_distinct([X,Y,Z]).
```

the weak arc-consistency algorithm detects the inconsistency of the constraint without labeling any variables. For the following query,

```
X in {1,2}, Y in {1,2}, Z in {1,2,3}, all_distinct([X,Y,Z]).
```

the algorithm assigns `3` to `Z` without labeling any variables.

The weak arc-consistency algorithm is not as powerful as the algorithm proposed by Regin [18] in terms of pruning ability but is much easier to implement.

To incorporate the weak arc-consistency checking algorithm into the linear-space propagator, we only need to redefine `outof(X,Left,Right)` as follows:

```
outof(X,Left,Right), var(X), {ins(X),minmax(X),dom(X)} =>
      outof_reducer(X,Left,Right).
outof(X,Left,Right) => exclude_list(X,Left),exclude_list(X,Right).
```

where `outof_reducer(X,Left,Right)` first counts the number of variables in `Left` and `Right` whose domains are subsets of the domain of `X` and then decides what action to take depending on the number and the size of the domain of `X`.

The key operation is to decide whether a domain is a subset of another domain. In the worst case, the two domains have to be scanned. There are several facts that can be used to avoid scanning domain elements. A domain `D1` cannot be a subset of another domain `D2` if `D1` has a larger size or has a larger interval. Also if two domains are intervals without holes, then scanning the elements is unnecessary. Another fact that can be used in the detection is that if the event is `dom(X,E)` meaning that `E` has been excluded from `X`'s domain, then another domain `Y` cannot be a subset of `X` if `E` is included in `Y`. To take advantage for this fact, the propagator can be rewritten into the following:

```
all_distinct(L) => all_distinct(L,[]).

all_distinct([],Left) => true.
all_distinct([X|Right],Left) =>
      outof(X,Left,Right),
      outof_dom(X,Left,Right),
      all_distinct(Right,[X|Left]).

outof(X,Left,Right), var(X), {ins(X),minmax(X)} =>
      outof_reducer(X,Left,Right).
outof(X,Left,Right) => exclude_list(X,Left),exclude_list(X,Right).

outof_dom(X,Left,Right),var(X), {dom(X,E)} =>
      outof_reducer(X,E,Left,Right).
outof_dom(X,Left,Right) => true.
```

The subgoal `outof_reducer(X,E,Left,Right)` takes `E` into account when detecting whether a domain is a subset of that of `X`.

5 Implementation and Performance Evaluation

B-Prolog has been extended to accommodate action rules and the finite-domain constraint solver described in this paper has been developed in action rules. In this section, we first briefly describe the implementation of action rules and then evaluate the performance of the finite-domain constraint solver. The description of the implementation serves as the grounds for the explanations of the experimental results. A detailed description of the implementation is beyond the scope of this paper.

5.1 Implementation

The abstract machine of B-Prolog, called ATOAM [25], is extended to support agents. The ATOAM is a variant of the Warren Abstract Machine (WAM) [2]. Unlike in the WAM where arguments are passed through argument registers, arguments in the ATOAM are passed through stack frames and only one frame is used for each subgoal. Each time when a predicate is invoked by a subgoal, a frame is placed on top of the control stack unless the frame currently at the top can be reused. Frames for different types of predicates have different structures. Agents are stored as frames on the control stack. The frame for an agent has the following slots in addition to those included in a normal frame: the status of the agent, the activating event, the program pointer to move to when the agent is activated, and the pointer to the previous agent's frame in the chain of agents. The frames on the control stack comprise three chains, namely the chain of *active subgoals* that are being executed, the chain of *choice points*, i.e., subgoals that have alternative clauses to be tried when the execution backtracks to them, and the chain of *agents*.

Storing agents on the stack facilitates context-switching [26] for agents but complicates memory management. With frames of agents on the stack, the chronological order of frames is no longer preserved and therefore run-time checking is needed to determine whether the frame at the top can be reused and a garbage collector is needed to collect useless frames on the control stack [28].

Action rules are compiled into matching trees [25] such that shared tests among different rules do not need to be executed multiple times. This technique is useful for speeding-up constraint propagators.

The actions of constraint propagators are to reduce the domains of variables. This characteristic is used to improve the performance of constraint propagators. Some events that cannot lead to the shrinking of any domains are ignored. For example, if multiple events of `minmax(X)` are posted at the same time, then only one of them needs to be handled, and if `minmax(X)` and `ins(X)` are posted at the same time, then the `minmax(X)` event is ignored. In this way, many redundant activations of rules that do not lead to the reduction of any domains can be suppressed.

5.2 Performance Evaluation

We compared the performance of B-Prolog version 6.1 (BP)[1] with GNU-Prolog (GP), one of the fastest CLP(FD) systems available now. In BP, the hybrid algorithm presented in this paper is employed as the default algorithm for non-binary equality constraints and the linear-space propagator is used for `all_distinct`. In order to evaluate the effectiveness of the hybrid algorithm, we compared the default solver with another solver in B-Prolog, called BPIC, that uses the interval consistency algorithm.

Table 1 shows the CPU times taken by three solvers for a set of benchmarks[2], assuming the time taken by BP is 1. Most of the benchmarks have been widely used

[1] Available from www.probp.com.

[2] Available from `www.probp.com/bench/clpfd.tar.gz`.

Table 1: Comparison of CPU times (Linux).

Program	BP	BP-IC	GP
alpha	1	1.24	0.96
bridge	1	0.98	0.51
crypta	1	0.90	1.72
cars	1	0.96	1.07
color	1	1.00	1.03
eq10	1	0.94	2.78
eq20	1	0.96	1.68
magic3	1	0.74	0.87
magic4	1	0.82	1.23
olympic	1	0.92	1.49
queens(25)	1	0.99	0.34
sendmoney	1	0.90	1.80
sudoku81	1	1.01	1.23
zebra	1	0.92	1.28
Arithmetic mean	1	0.95	1.29
Geometric mean	1	0.94	1.15

by other authors to compare CLP(FD) systems [3, 4, 23]. Three new programs were added by the author into the set: `color` is a program that colors a map with 110 regions; `olympic` is a puzzle taken from a Mathematics Olympic game for elementary students; and `sudoku81` is a program for solving a puzzle. The left-to-right labeling strategy is used to instantiate variables in all the benchmarks. The CPU times were measured on a 400 MHz Pentium PC running Linux. Each program was run at least 10 times and the average was taken. For some programs, execution was repeated up to 1000 times to obtain a stable average. BP supports garbage collection but not GP. To be fair, the garbage collector was disabled in this comparison.

On overage, the default solver of BP is 15 percent faster and BPIC is 20 percent faster than GP. BP outperforms GP remarkably for programs that contain non-binary equality constraints, such as `crypta`, `eq10`, `eq20`, and `sendmoney`. This result reveals that the disadvantages of splitting n-ary constraints into indexicals outweigh the advantages. On the other hand, GP is three times as fast as BP for `queens(25)` and twice as fast for `bridge`. The fast speed for `queens` may be attributed to an optimization technique adopted in GP that combines indexicals. If the action rules for the disequality constraints are combined for the program, the speed will be doubled. For `bridge`, BP is slower than GP because in BP updates of lower and upper bounds are not treated as different events. Therefore, for the constraint $X \leq Y$, the propagator is activated even if the upper bound of X or the lower bound of Y is updated.

Comparing BP and BPIC reveals that the hybrid algorithm is effective for

Table 2: Comparison of numbers of backtracks.

Program	BP	BP-IC	GP
alpha	*4605*	8440	8440
bridge	0	0	0
crypta	52	52	52
cars	53	53	*34*
color	560	560	560
eq10	49	49	49
eq20	49	49	49
magic3	2	2	2
magic4	18	18	18
olympic	*36*	50	50
queens(25)	7255	7255	7255
sendmoney	2	2	2
sudoku81	0	0	0
zebra	2	2	2

`alpha` only. The hybrid algorithm is adopted as the default one since for some programs, such as the queens program given in [17][3], BP is exponentially faster than BPIC. BP is slightly slower than BPIC for most of the programs. In general, this happens for programs for which the efforts to reduce domains do not pay off.

Table 2 gives the numbers of backtracks performed by the programs. `BP` makes the same number of backtracks as `BPIC` except for `alpha` and `olympic`, and `GP` makes fewer backtracks than `BP` for `cars`. Basically, the three solvers explore the same search tree for most of the programs. Therefore, the comparison results shown in Table 1 reflect the real performance of the solvers.

GP and BP are quite different. In GP constraints are compiled into indexicals defined in C while in BP constraints are compiled into propagators defined in action rules. In GP programs are compiled into native code using the WAM as the intermediate language, while in BP programs are compiled into the ATOAM virtual machine code which is interpreted by an emulator. Although the GP Prolog engine may not have much impact on the performance of constraint programs, the BP Prolog engine does have a great impact since all the propagators are defined in action rules. One evidence for this observation is that the BP constraint solver becomes 20-30 percent faster after the main switch statement in the emulator is changed to a jump table. A further speed-up is expected if a native code compiler is employed.

There are other factors that affect the performance of a solver, such as domain representation, interaction with other solvers, and garbage collection. GP supports only finite domains of positive integers, while BP supports not only finite integer

[3]This program is not included in the benchmark set since it requires support of negative integer domains and thus cannot run on GP.

domains but also trees and finite domains of ground terms and sets [29]. In BP, integer domains are represented as described in Subsection 3.4. BP adopts a sound arithmetic that guarantees that no solution is lost. For example, when excluding an inner element from a large interval domain, the system generates a disequality constraint rather than brutally changes the interval into a bit vector. BP has a garbage collector that collects garbage on the heap and the control stack, but GP does not support garbage collection yet. It is likely that garbage collection will suppress some optimization techniques.

Finite-domain constraint solving has been a very popular research area and many constraint solvers have been developed over the last decade. In addition to GNU-Prolog and B-Prolog, the Prolog systems such as CHIP, CLP(BNR), Eclipse, IF/Prolog, Nicolog, and Sicstus all support finite-domain constraint solving. Mozart Oz can be classified as a CLP system although it adopts different syntax from Prolog. In addition to CLP(FD) systems, there are constraint solvers such as Ilog solver and Claire that are housed in C++. Comparison results with Eclipse and Sicstus are available at `www.probp.com/fd_evaluation.htm`. B-Prolog is over four times as fast as Eclipse and Sicstus for the same set of benchmarks. No comparison has been done with other systems. According to [6], GNU-Prolog is as fast as the ILOG solver and twice as fast as CHIP from Cosytec. According to [20], Mozart Oz has comparable performance with Eclipse and Sicstus. There are also different implementations of CHR and different finite-domain constraint solvers implemented in CHR. No comparison was conducted with CHR either. According to [12], the fastest constraint solver implemented in CHR is still over 10 times slower than Sicstus.

6 Related Work

CLP(FD) systems have undergone an evolution process, from closed to open and from low level to high level. Several constructs have been proposed to facilitate the implementation of constraint propagators. Examples include attribute variables [11], indexicals [4], extended indexicals called projection constraints [21], delay clauses [15, 27], and constraint handling rules [10]. In this paper, we proposed another construct called action rules. An action rule is an extension of a delay clause that allows for the descriptions of not only delay conditions for subgoals but also activating events and actions.

Action rules are more powerful and flexible than indexicals. We described in this paper several propagation algorithms in action rules, some of which are impossible to encode in indexicals (e.g., the hybrid algorithm for n-nary constraints) and some of which cannot be implemented as efficiently (e.g., arc-consistency for binary constraints). Consider the following indexical taken from [3],

```
X in dom(Y)+C
```

which maintains the arc-consistency for the constraint `X = Y+C` on `X`. Whenever an element `y` is excluded from the domain of `Y`, the indexical will be activated. Because the indexical does no know what the excluded element is, it has to go

through the domain elements of `Y` in the worst case to locate a possible no-good value in the domain of `X`. In contrast, in the propagator implemented in action rules, the propagator knows exactly what element is excluded from the domain of `Y` and thus can compute the counterpart in the domain of `X` in constant time.

Compiling constraints into indexicals enables the use of more specialized propagators and restricts propagation to within a small range of constraints [4]. Nevertheless, this approach has to introduce new temporary variables and lower the granularity of propagators. Our experiment reveals that B-Prolog outperforms GNU-Prolog for almost all the benchmarks that contain non-binary constraints. This result reveals that the disadvantages outweigh the advantages. Similar observations have been made independently in [3, 9]. In [9], the same two-phase algorithm is used to reduce domains of linear constraints.

Attribute variables [11] are variables with attached attributes each of which has a list of handlers. Touching an attribute will trigger the corresponding list of handlers. In order to make context-switch swift for handlers, systems such as Eclipse treats handlers as demons rather than as normal subgoals. A daemon is different from a normal subgoal in that it does not disappear after execution but instead waits for another activation. In this sense, agents in our system are similar to daemons. Nevertheless, an agent can be activated by different kinds of events and an agent may take different actions depending on the conditions. An agent can be defined by multiple action rules and the rules are compiled into a tree by the compiler such that shared tests are combined and conditions that are tested failure once need not be tested again. Systems, such as Oz [20] and Sicstus [3], provide interfaces for implementing propagators and provide also some sort of delay construct similar to attribute variables that triggers propagators when events are posted.

The action rule language is an extension of our early delay construct proposed in [27] that allows the event `dom(X,E)` and user-defined events. That delay construct is an extension of Meier's delay clause construct [15] that allows not only delay conditions but also events and actions. In Meier's delay clause, events are implicitly extracted from delay conditions and a delayed subgoal never takes actions as long as the delay condition is satisfied. In retrospect, all these constructs were inspired by early work by Colmerauer and Naish [5, 16].

CHR [10] resembles a production system. In CHR, the left-hand side of a rule specifies a pattern of constraints in the constraint store and the right-hand side specifies new constraints to replace those on the left-hand side or to be added into the store. It should be possible to implement in CHR all the propagation algorithms described in this paper provided certain built-ins are added. If events are treated as constraints, then an action rule can be translated into a CHR rule. Treating events as constraints, however, can hardly achieve the same performance. Events are removed automatically after all the agents that are waiting for them are activated. In CHR, there must be rules that remove the events explicitly. The left-hand sides of CHR rules can have multiple constraint patterns. Therefore, it is impossible in general to translate a CHR rule into action rules straightforwardly. It is not clear whether or not it is possible to simulate CHR rules in action rules and how if the answer is yes. It would be an interesting direction to explore in the

future.

7 Conclusion

There is a need for an implementation language for constraint propagators that is expressive enough and can be implemented efficiently. This paper presents such a language called action rules. The expressiveness of the language is illustrated though several examples that cannot be implemented in indexicals: the propagator for maintaining arc-consistency of binary equality constraints; a week arc-consistency algorithm for the `all_distinct` constraint; and a hybrid algorithm for non-binary equality constraints that combines interval and arc-consistency ones. The efficiency is evaluated through benchmarking. For a set of widely used benchmarks, our solver implemented in B-Prolog competes favorably with that in GNU-Prolog, one of the fastest finite-domain constraint solvers available now.

The results are encouraging and promising since our solver is implemented in a high-level language and B-Prolog is an emulator-based system which provides more facilities than GNU-Prolog such as garbage collection and constraint solving over other domains. The high-performance of our solver stems from the following facts. Firstly, only one propagator is generated for each non-binary equality constraint that maintains the interval-consistency. Our solver performs especially well for the benchmarks that have non-binary equality constraints. This reveals that compiling non-binary equality constraints into indexicals has more cons than pros. Secondly, the hybrid algorithm adopted in our solver is a good compromise between the need to achieve a high-level consistency to cut search spaces and the need to reduce the cost. The cost of achieving arc-consistency for binary constraints is relatively small, but the effect can be very big for certain programs. Thirdly, our solver employs optimization techniques that reduce redundant activations of propagators.

Our solver can be improved further in the following aspects: First, develop new optimization techniques for further avoiding redundant activations of propagators; Second, compile programs into native code to enhance the performance of action rules; and Third, try consistency algorithms beyond interval and arc consistency.

Acknowledgement

This is a significantly extended and revised version of [27]. The author thanks Warwick Harvey, Joachim Schimpf, and the referees for helpful comments and corrections on the early versions of this paper.

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

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

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

[5] 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.

[6] D.Diaz and P. Codognet: The GNU-Prolog System and Its Implementation, *ACM SAC*, 2000.

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

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

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

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

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

[12] C. Holzbaur and T.W. Fruhwirth: Compiling Constraint Handling Rules into Prolog with Attributed Variables, *International Conference on Principles and Practice of Declarative Programming*, pp.117-133, 1999.

[13] J. Jaffar and J.-L. Lassez: Constraint Logic Programming, *Proc. Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of programming languages*, pp.21-23, 1987.

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

[15] M. Meier: Better Late Than Never, *Implementations of Logic Programming Systems*, E. Tick and G. Succi, Eds., Kluwer Academic Publishers, 1994.

[16] Naish, L.: *Negation and Control in Prolog*, Lecture Note in Computer Science, 238, 1985.

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

[18] J.C. Regin: A Filtering Algorithm for Constraints of Difference in CSPs, In *Proc. AAAI*, 1994.

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

[20] C. Schulte: *Programming Constraint Services*, Dissertation, University of Saarlandes, 2000.

[21] G. Sidebottom and W.S. Havens: Nicolog: A Simple Yet Powerful cc(FD) Language, *Journal of Automated Reasoning*, Vol.17, pp.371-403, 1996.

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

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

[24] P. Van Hentenryck, V. Saraswat, and Y. Deville: Constraint Processing in cc(FD), In *Constraint Programming: Basics and Trends*, A. Podelski, Ed., LNCS 910, 1995.

[25] 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.

[26] N.F. Zhou: A Novel Implementation Method of Delay, *Proc. Joint International Conference and Symposium on Logic Programming*, pp.97-111, MIT Press, 1996.

[27] 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.

[28] N.F. Zhou: Garbage Collection in B-Prolog, *Proc. of the First Workshop on Memory Management in Logic Programming Implementations*, CL'2000, 2000.

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