Programming Finite-Domain Constraint Propagators in Action Rules

Neng-Fa Zhou
Department of Computer and Information Science
CUNY Brooklyn College & Graduate Center
zhou@sci.brooklyn.cuny.edu

submitted 30 March 2003; revised 20 May 2004, 6 January 2005; accepted 30 May 2005

Abstract

In this paper, we propose a new language, called AR (*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. AR combines 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. AR has a much stronger descriptive power than *indexicals*, the language widely used in the current finite-domain constraint systems, and is flexible for implementing not only interval-consistency but also arc-consistency algorithms. As examples, we present a weak arc-consistency propagator 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 B-Prolog as a CLP(FD) system significantly outperforms other CLP(FD) systems.

KEYWORDS: constraint programming, constraint propagation, action rules

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 (Dincbas et al. 1990; Jaffar and Maher 1994). The key operation employed in CLP(FD) is called *constraint propagation* (Kumar 1992; Tsang 1993), 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, finite sets, floating-point numbers, and many other domains (Jaffar and Maher 1994).

In early CLP(FD) systems, such as the CHIP system (Dincbas et al. 1988), 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 success of compiling Prolog programs into the Warren Abstract Machine (WAM) (Warren 1983), a former research group at ECRC extended the WAM for compiling CLP(FD) (Aggoun and Beldiceanu 1991). 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 (Aggoun and Beldiceanu 1991) has over 100 instructions for compiling finite-domain constraints alone!

A language construct, called *indexicals*, has been quite popular as an intermediate language for compiling finite-domain constraints. The language was first proposed in (van Hentenryck et al. 1992) and then popularized by (Codognet and Diaz 1996). This language is also adopted by other systems (Carlsson et al. 1997; Sidebottom and Havens 1996). An indexical is a primitive constraint in the form of X *in* r, where X is a domain variable and r is a range expression 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) (Frühwirth 1998) 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, HAL, 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 descriptive 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 (Holzbaur and Fruhwirth 1999; Holzbaur et al. 2004).

This paper proposes a new language, called AR (*Action Rules*), which 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 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. AR is an extension of delay constructs such as delay clauses (Meier 1994) that allows for the descriptions of not only delay conditions but also activating events and actions (Zhou 1998). The syntax, operational semantics, and implementation of AR will be described in Section 3.

The focus of this paper is on how to implement various propagators for finite-domain constraints in AR. 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`. AR is 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 AR and several constraint solvers including the ones over finite domains, Boolean, trees, and finite sets have been developed in AR (Zhou 2002). Section 5 compares the performance of the finite-domain solver of B-Prolog with GNU-Prolog (GP), a state-of-the-art implementation of CLP(FD) (Diaz and Codognet 2001), and two other CLP(FD) systems: ECLiPSe and SICStus. Benchmarking results show that B-Prolog is significantly faster than GP and 4-6 times as fast as ECLiPSe and SICStus.

Readers are assumed to be familiar with logic programming and constraint satisfaction, but no knowledge about the compilation is assumed. In Section 2, we define some preliminary terms and concepts about CLP(FD) and constraint propagation. Readers are referred to (Marriott and Stuckey 1998), (Hentenryck 1989) and (Kumar 1992) for the details. A brief description of the implementation of AR is given in Section 3, and a more detailed description can be found in (Zhou 2003).

2 Preliminaries

2.1 CLP(FD)

CLP(FD) (Hentenryck 1989) 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:

$$Vars \ \texttt{in} \ D$$

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 pairwise different.

2.2 Constraint propagation

Constraint Propagation (Kumar 1992; Tsang 1993) 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 the domains of the variables in the constraints are updated. Propagating the updates to other variables may result in the shrinking of the domains of the variables or the instantiation of the variables.

There are different levels of consistency for constraints such as *node*, *interval*, *bounds*, *arc*, and *path* consistency (Tsang 1993; Marriott and Stuckey 1998). We define below three levels of consistency needed in this paper, namely node, interval and arc consistency, and define the propagators that maintain them.

A unary constraint $p(X)$, where X has 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, the constraint becomes unary and 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 a linear equality constraint $c + a_1 \times X_1 + a_2 \times X_2 + \ldots + a_n \times X_n = 0$ where $a_i \neq 0$ and X_i is defined over the domain D_i $(i = 1, \ldots, n)$. Let

$$g_i(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n) =$$
$$\frac{-c - a_1 \times X_1 - \ldots - a_{i-1} \times X_{i-1} - a_{i+1} \times X_{i+1} - \ldots - a_n \times X_n}{a_i}$$

and l and u be the functions defined as follows:

$$l(g_i(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n)) =$$
$$min\{g_i(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) | x_k \in D_k, 1 \leq k \leq n, k \neq i\}$$
$$u(g_i(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n)) =$$
$$max\{g_i(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) | x_k \in D_k, 1 \leq k \leq n, k \neq i\}$$

The constraint C is said to be *interval consistent w.r.t.* X_i if:

$$\forall_{x \in D_i} (l(g_i(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n)) \leq x \leq u(g_i(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n))$$

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 said to be *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 an inequality constraint.

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. Maintaining arc consistency for a non-binary constraint requires examining Cartesian products of the domains (Dechter 2003) and is thus very costly. For this reason, some CLP(FD) systems maintain arc consistency only for binary constraints and many others do not consider arc consistency at all.

2.3 Domain variables

A *domain variable* is a suspension 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
bound_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 a domain is an interval without holes, no bit vector is necessary and *elms* is a null pointer.

The following built-in predicates and functions are available on domain variables.

- `dvar(`X`)`: Succeeds if X is a domain variable.

- `min(`X`)`, `max(`X`)`: Functions that return, respectively, the minimum and maximum elements of the domain of X.
- `size(`$X, Size$`)`: The size of the domain of X is $Size$.
- `exclude(`X, E`)`: Excludes the element E from the domain of X.
- X `in` D: The new domain for X is the intersection of its existing domain and D where D is a set of ground terms or a range $l..u$ of integers.

A failure occurs when the domain of a variable becomes empty. When the domain of a variable becomes a singleton, the variable is instantiated to the element automatically.

An event is posted whenever the domain of a variable is updated. For a domain variable X, instantiating X posts the event `ins(`X`)`,[1] updating the lower or upper bound of the domain posts the event `bound(`X`)`, and excluding an inner element E from the domain posts the event `dom(`X, E`)`. Notice that the event `bound(`X`)` is not posted when X is instantiated and the event `dom(`X, E`)` is not posted when either bound of the domain of X is updated. This implies that a propagator that maintains arc consistency has to handle not only `dom(`X, E`)` events but also `bound(`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, `bound(`X`)` activates the list *bound_cs*, and `dom(`X, E`)` activates the list *dom_cs*.

3 The AR Language

AR is designed for programming interactive agents. In this section, we describe the syntax, operational semantics, and implementation of AR.

3.1 Syntax

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 conjunction of conditions on the agents, *Event* is a non-empty disjunction of patterns for events that can activate the agents, and *Action* is a sequence of subgoals.[2] *Condition* and the following comma can be omitted if no condition is needed on *Agent*. *Action* cannot be empty. The subgoal `true` represents an empty action that always succeeds. An action rule degenerates into a *commitment rule* if *Event* together with the enclosing braces are missing. Conditions, event patterns, and actions are all atomic formulas where the delimiter ‘,’ is used to separate the constituents.

An AR *predicate* consists of a sequence of rules defining agents of the same

[1] A variable is said to be instantiated if it is bound to another term, possibly another variable.
[2] Subgoals in *Action* can be any subgoals including those defined by Prolog clauses.

predicate symbol. In a program, AR predicates can be intermingled with Prolog predicates. In this paper, the term *agents* is used to refer to subgoals that can be suspended and activated, and the term *predicate* is used refer to both AR and Prolog predicates unless explicitly specified.

All conditions must be *in-line tests*.[3] In the implementation of AR in B-Prolog, the following types of conditions are allowed:

- Type and mode checking: Predicates like `integer(`X`)`, `var(`X`)`, and `nonvar(`X`)`.
- Matching: A matching call takes the form of $X = Y$ where one of the arguments must be a non-variable term at compile time and the other must be a variable (again at compile time) that occurs before in the rule. The non-variable term serves as a pattern and the variable refers to a term to be matched against the pattern. This call succeeds if the pattern and the term become identical after a substitution is applied to the pattern. For instance, the condition $f(X) = Y$ succeeds if Y is a structure whose functor is $f/1$.
- Term inspection: Several built-in predicates including `arg/3`, `functor/3`, `==/2`, `\==/2`, and `n_vars_gt/2` can be used in the condition of a rule to inspect the arguments of an agent. The call `n_vars_gt(`$Term, N$`)` succeeds if the number of variables occurring in $Term$ is greater than N.
- Arithmetic comparison: Checks the arithmetic equality (=:=), disequality (=\=), or inequality (>, >=, <, and =<) of two terms which must be ground at runtime.

A set of built-in events is provided.[4] As far as programming constraint propagators is concerned, an event pattern can be one of the following:

- `generated`: The action of the rule is executed when the agent is suspended for the first time.
- `ins(`X`)`: The agent is activated when an event `ins(`X`)` is posted.
- `bound(`X`)`: The agent is activated when an event `bound(`X`)` is posted.
- `dom(`X`)` and `dom(`X, E`)`: The agent is activated when an event `dom(`X, E'`)` is posted. Before the action is executed, E is made to reference the element E'.

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 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 `post(`E`)` posts the event E.

In an action rule, the event pattern `dom(`X, T`)` or `event(`X, T`)` is not allowed to coexist with any other event patterns and T must be a first-occurring variable so that when the action of the rule is executed T always refers to the second argument of the event.

[3] An in-line call is compiled into instructions that do not invoke any predicates. A test does not change the instantiation status of the variables in its arguments.

[4] In the implementation in B-Prolog, built-in events are provided for programming constraint propagators, graphical user interfaces, and interactive agents.

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))
```

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

The following defines the freeze predicate in Prolog-II (Colmerauer 1984).

```
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)` is suspended. Only when `X` becomes a non-variable term, can the second rule be applied.

Consider, as another 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),{generated,ins(Y),bound(Y),ins(Z),bound(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 action is also executed before the propagator is first suspended so that no preprocessing is needed to enforce interval consistency.

3.3 Operational Semantics

The operational semantics of AR can be presented as a state-transition system as shown in Figure 1. An agent may be in one of the following states: *start*, *sleep*,

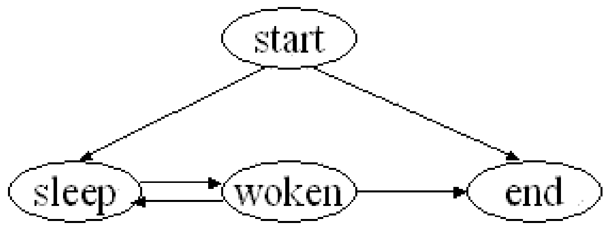

Fig. 1. Diagram of state transition of agents.

woken, and *end*. When an agent is generated, it enters the *start* state and is executed immediately. A typical agent transits to the *end* state through the *sleep* and *woken* states. An agent is said to be *floundering* if it stays in the *sleep* state forever. It is the programmer's responsibility to prevent agents from floundering.

When an agent is generated, the system searches in its predicate in textual order 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 there exists a unifier θ such that $H\theta = \alpha$ and $C\theta$ is satisfied.[5] If no rule is found applicable, the agent fails. If a commitment rule is found, the agent is substituted for the body,[6] and its state is changed to *end*.

If an action rule "$H, C, \{E\} => B$" is found for the agent, the agent is suspended, transiting from *start* to *sleep*. It will stay in the *sleep* state until it is activated by one of the events in E.

When an event is posted, all the sleeping agents waiting for the event in the system are woken up and the event is erased after that so that no agents generated later will be responsive to this event. The woken agents are added to the queue of active subgoals in some order. It is up to the implementer of the language to use a strategy to schedule activated agents. Whatever scheduling strategy is adopted, the programmer should not rely on the strategy to guarantee the correctness of programs.

Suppose an agent was put into *sleep* by the action rule "$H, C, \{E\} => B$" and was woken up by one of the events in E. After this agent is picked by the scheduler, the system tests the condition C again. If it is satisfied, the action B is executed. If the action succeeds, the agent is re-suspended. If the action fails, the agent fails as well. If the condition C of the action rule does not hold, the system searches for an alternative applicable rule for the agent just as for a newly generated agent.

There is no primitive for killing agents explicitly. An agent never disappears as long as action rules are applied to it successfully. An agent transits to the *end* state only when a commitment rule is applied to it.

[5] Notice that since one-directional matching rather than full unification is used to search for an applicable rule and in the condition no variable in α can be instantiated, the agent remains the same after an applicable rule is found.
[6] A commitment rule is similar to a guarded clause in concurrent logic languages (Shapiro 1989), but an agent can never be blocked while it is being matched against an agent pattern.

3.4 The Implementation

The abstract machine of B-Prolog, called ATOAM (Zhou 1996b), is extended to support agents (Zhou 2003). This subsection briefly describes this implementation. A more detailed description can be found in (Zhou 2003).

3.4.1 The frame structure for agents

The ATOAM is a variant of the Warren Abstract Machine (WAM) (Warren 1983). 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 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:[7]

STATE:	State of the agent
EVENT:	Activating event
REEP:	Re-entrance program pointer
PREV:	Previous agent in the chain

The STATE slot has one of those states shown in Figure 1 as its value. The EVENT slot stores the last event that activated the agent. The action rule of the agent can have access to this event. The REEP slot stores the program pointer to continue when the agent is activated. The PREV slot stores 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*[8] that are being executed, the chain of *choice points*, i.e., subgoals that have alternative clauses to be tried when execution backtracks to them, and the chain of *agents* in either *sleep* or *woken* states.

Storing agents on the stack facilitates context switching for agents (Zhou 1996a) but complicates memory management. With frames of agents on the stack, the chronological order of frames is no longer preserved, and therefore a garbage collector is needed to collect useless frames on the control stack and run-time checking is needed to determine whether the current frame can be reused.

3.4.2 When and how to invoke agents?

For the sake of efficiency, events are not checked after each instruction but checked at the entry and exit points of each predicate. If it is found that the list of events

[7] A normal frame has the following slots: arguments, FP (parent frame), CPS (continuation program pointer on success), TOP (top of the control stack), BTM (bottom of the frame), and local variables. A choice point frame has the following additional slots: CPF (continuation program pointer on failure), B (parent choice point), H (top of the heap), and T (top of the trail stack).

[8] i.e., the frames connected by FP.

is not empty, those agents that are waiting for the events are added into the active chain and the current predicate is interrupted. After the activated agents complete their execution successfully, the interrupted predicate resumes its execution. The programmer has no control over the order in which agents are added. In our implementation, the *first-generated-first-served* strategy is used.

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 must be activated once for each of the events. If an agent is found to be active already when the system tries to add it into the active chain, the sytem makes a copy of it and adds the copy into the chain.

The actions of constraint propagators are to reduce the domains of variables. This characteristic is exploited 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 `bound(X)` are posted at the same time, then only one of them needs to be handled, and if `bound(X)` and `ins(X)` are posted at the same time, then the `bound(X)` event is ignored. In this way, many redundant activations of rules that do not contribute to the reduction of any domains can be suppressed. This optimization is applied to constraint propagators only and not general agents.

In a CLP(FD) program, constraint propagation is normally intertwined with non-deterministic subgoals such as `labeling` that assign values to variables. If a non-deterministic predicate is interrupted by events, then no choice point can be created for the interrupted predicate until the activated agents are all executed. Consider the following example:

```
?-p(X),X=f(Y),q(X),write(X).

p(X),var(X),{ins(X)} => true.
p(X) => X=f(a).

q(X):-fail.
q(X).
```

First the agent `p(X)` is generated, waiting for `X` to be instantiated. The subgoal `X=f(Y)` posts an event `ins(X)` after `X` bound to `f(Y)`. At the entry of `q(X)`, the event is detected and `p(X)` is activated. If there were a choice point created for `q(X)` before `p(X)` is activated, the binding `Y=a` given by the second rule of `p(X)` would be lost when `fail` in `q/1` is executed since `Y` is older than the choice point, and the output from `write(X)` would be `f(Y)` not `f(a)`.

4 Programming Constraint Propagators in AR

The high descriptive power of AR 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 weak arc-consistency propagator for the global constraint `all_distinct`.

4.1 Binary constraints

We consider how to implement propagators for the binary constraint $A \times X = B \times Y + C$, 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. It can be proved easily that no value outside the range `L..U` satisfies the constraint.

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),
      {generated,bound(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 takes 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 excludes 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 takes 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. Firstly, it generates linear-size code. Secondly, indexicals can be implemented in a low-level language to achieve better performance. Thirdly, information propagation can be restricted to only those constraints for which the domains have the possibility to be reduced (Codognet and Diaz 1996). For example, consider the two ternary 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.

In comparison with indexicals, the high descriptive power of AR 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 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 $A_1 \times X_1 + \ldots + A_n \times X_n + C = 0$ be an n-ary constraint where each A_i $(i = 1, \ldots, n)$ 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),
      {generated,ins(X1),bound(X1),...,ins(Xn),bound(Xn)}
=>
      ... % reduce the domains of X1,...,Xn.
```

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

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

$$\begin{aligned}
&T_0 = C,\\
&T_1 = T_0 + A_1 * X_1,\\
&T_2 = T_1 + A_2 * X_2,\\
&\ldots\\
&T_n = T_{n-1} + A_n * X_n
\end{aligned}$$

The generated reducer first computes the lower and upper bounds of the temporary variables[9] by propagating information forward from T_0 to T_n. The lower and upper bounds of T_i are computed from those of T_{i-1} and $A_i \times X_i$ $(i = 1, \ldots, n)$. After that, the reducer propagates information backward from T_n to T_1. For each tuple $T_i = T_{i-1} + A_i \times X_i$, 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)
      {generated,ins(X1),bound(X1),ins(X2),bound(X2),
       ins(X3),bound(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,                        % T3 = 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.

[9] Temporary variables are plain variables, not domain variables. Therefore, this compilation scheme is different from compiling constraints into indexicals.

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 $A1 \times X1 + ... + An \times Xn + C{=}0$.

```
'A1*X1+...+An*Xn+C=0'(C,A1,A2,...,An,X1,X2,..,Xn),
      n_vars_gt([X1,...,Xn],2),
      {generated,ins(X1),bound(X1),...,ins(Xn),bound(Xn)}
=>
      ... % reduce domains of X1,...,Xn.
'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` fails and the second rule is tried. The subgoal `nary_to_binary` transforms the constraint into the binary constraint $B1 \times Y1 + B2 \times Y2 + NewC{=}0$, and the next subgoal invokes an appropriate propagator for the binary constraint.[10]

4.3 Propagators for `all_distinct`

The constraint `all_distinct(`L`)` holds if the elements in L are pairwise 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 fine-grained ones may lose possible propagation opportunities (Regin 1994; Puget 1998). This subsection presents two propagators for the constraint. The propagation algorithms are not new. The goal of this subsection is to illustrate the expressive power of AR.

4.3.1 A linear-space propagator

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

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

all_distinct([],Left) => true.
```

[10] In the implementation in B-Prolog, the two built-ins `n_vars_gt` and `nary_to_binary` do not take the constraint as an argument but instead access the constraint in the parent subgoal. In this way, no copy of the constraint needs to be made.

```
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` in `L`. 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 non-variable, `exclude_list(X,Left)` and `exclude_list(X,Right)` exclude `X` from the domains of the variables in `Left` and `Right`, respectively.

There is one 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; otherwise, the constraint would take quadratic space.

4.3.2 Weak arc consistency

In terms of pruning ability, the linear-space propagator is the same as the naive one that splits a constraint of `all_distinct` into binary disequality constraints. In this subsection, we present a propagator that has stronger prunning power than the naive propagator.

Given any set of values D of size n, the constraint `all_distinct`(L) is said to be *weak arc consistent* if there are at most n variables in L whose domains are subsets of D. For each variable X in L, let $L - \{X\}$ be the list of variables 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, we can safely 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 propagator 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 propagator is not as powerful as the algorithm proposed by Regin (Regin 1994) in terms of pruning ability but is much easier to implement. To incorporate weak arc-consistency checking into the linear-space propagator, we only need to redefine `outof(X,Left,Right)` as follows:

```
outof(X,Left,Right), var(X), {generated,ins(X),bound(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 variables in `Left` and `Right` whose domains are subsets of the domain of `X` and then decides what action to take depending on the count 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 of 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), {generated,ins(X),bound(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 Performance Evaluation

B-Prolog has been extended to accommodate AR and the finite-domain constraint solver described in this paper has been developed in AR. In this section, we evaluate the performance of the finite-domain constraint solver.

Table 1. *Comparison of CPU times.*

Program	XP					Linux	
	BP-IC	BP-AC	GP	EP	SP	BP-IC	GP
alpha	1	0.82	1.02	7.23	3.17	1	0.77
bridge	1	0.94	1.20	3.60	3.57	1	0.92
cars	1	1.00	1.67	7.03	4.67	1	1.60
color	1	1.14	1.00	6.29	3.01	1	1.25
eq10	1	0.98	3.77	4.77	4.92	1	3.78
eq20	1	1.06	2.00	4.23	3.34	1	1.96
magic3	1	1.38	1.98	8.44	4.41	1	1.52
magic4	1	1.18	1.96	8.45	6.27	1	1.57
olympic	1	0.75	2.25	11.25	4.75	1	1.43
queens(25)	1	1.01	0.43	4.24	5.03	1	0.43
sendmoney	1	1.09	3.65	6.74	7.78	1	2.62
sudoku81	1	1.00	2.28	6.67	6.18	1	1.36
zebra	1	1.13	2.33	7.86	9.61	1	1.77
<arithmetic mean>	1	1.04	1.96	6.68	5.13	1	1.61
<geometric mean>	1	1.03	1.72	6.36	4.84	1	1.42

We compared the performance of B-Prolog version 6.7 (BP)[11] with three other CLP(FD) systems: ECLiPSe 5.8 #77 (EP), GNU-Prolog version 1.2.16 (GP), and SICStus 3.12.0 (SP). There are two solvers available in BP: one is called BP-AC which adopts the hybrid algorithm presented in this paper for equality constraints and the other called BP-IC which maintains only interval consistency for equality constraints. BP-AC is the default solver.[12]. The linear-space propagator is used for `all_distinct` in both solvers.

Table 1 shows the CPU times taken by the four solvers to run a set of benchmarks,[13] assuming the time taken by BP-IC be 1. Most of the benchmarks have been widely used by other authors to compare CLP(FD) systems (Carlsson et al. 1997; Codognet and Diaz 1996; Hentenryck 1989). 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 1.7GHz CPU running Windows XP. 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. Garbage collection was disabled. GP

[11] Available from www.probp.com.
[12] To switch to BP-IC, set the Prolog flag `constr_consistency` to `int`
[13] Available from probp.com/bench.tar.gz. See www.probp.com/benchmark_clpfd.htm for the latest results.

Table 2. *Comparison of numbers of backtracks.*

Program	BP-AC	BP-IC	GP
alpha	*4605*	8440	8440
bridge	0	0	0
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

has a native code compiler for Linux. The comparison with GP was also conducted on Linux.

The BP solvers compare favorably with GP and are significantly faster than EP and SP. EP is the slowest among the compared solvers, probably because of the overhead of supporting priority-based scheduling (Wallace et al. 2004). BP outperforms GP remarkably for programs that contain non-binary equality constraints, such as `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 more than twice as fast as BP for `queens(25)`. The high speed for `queens` may be attributed to an optimization technique adopted in GP that combines indexicals. If the propagators for the disequality constraints were combined for the program in BP,[14] the speed would be doubled.

Comparing BP-AC and BP-IC reveals that the hybrid algorithm is effective for `alpha` and `olympic` only. The BP-AC solver is adopted as the default one since for some programs, such as the queens program given in (Puget and Leconte 1995),[15] BP-AC is exponentially faster than BP-IC. BP-AC is slightly slower than BP-IC for some 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 three solvers. `BP-AC` makes the same number of backtracks as `BP-IC` except for `alpha` and `olympic`, and `GP` makes fewer backtracks than `BP-AC` for `cars`. Basically, the three solvers explore

[14] For the three disequality constraints $X \neq Y$, $X \neq Y + N$, and $X \neq Y - N$, we can use one propagator rather than three to handle the $ins(X)$ and $ins(Y)$ events.

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

the same search trees 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. Although the GP Prolog engine may not have much impact on the performance of constraint programs, the BP 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(Wallace et al. 2004). GP supports only finite domains of positive integers, while BP supports not only finite integer domains but also trees and finite domains of ground terms and sets (Zhou 2002). In BP, integer domains are represented as described in Subsection 2.3. BP adopts a sound and complete arithmetic that guarantees that solutions found are correct and no solution is lost. 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 as is done in GP.[16] BP has a garbage collector that collects garbage on the heap and the control stack, but GP does not support garbage collection yet. Garbage collection may suppress some optimization techniques.

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 attributed variables (Holzbaur 1992), indexicals (Codognet and Diaz 1996), extended indexicals called projection constraints (Sidebottom and Havens 1996), delay clauses (Meier 1994; Zhou 1998), and constraint handling rules (Frühwirth 1998). An action rule is an extension of a delay clause that allows for the descriptions of not only delay conditions on subgoals but also activating events and actions. This section compares AR with these constructs introduced into Constraint Logic Programming. Constructs introduced into other languages such as ILOG (Puget and Leconte 1995) and Oz (Schulte 2002) are not compared.

AR is more powerful and flexible than indexicals. We have described in this paper several propagation algorithms in AR, some of which cannot be encoded in indexicals (e.g., the hybrid algorithm for n-ary constraints) and some of which cannot be implemented as efficiently (e.g., arc consistency for binary constraints). Consider the following indexical taken from (Carlsson et al. 1997),

$$X \ in \ dom(Y) + C$$

[16] Generating a disequality constraint is less efficient than changing an interval into a bit vector since the disequality constraint needs to be checked each time the variable is instantiated.

which maintains arc consistency for the constraint $X = Y + C$ w.r.t. X. Whenever an element y is excluded from the domain of Y, the indexical is activated. Because the indexical does not 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 AR, 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 number of constraints (Codognet and Diaz 1996). 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 of splitting constraints outweigh the advantages. Similar observations have been made independently in (Carlsson et al. 1997; Harvey and Stuckey 2003; Zhou 1998). In (Harvey and Stuckey 2003), the same two-phase algorithm is used to reduce domains of linear constraints.

Attributed variables (Holzbaur 1992) are variables with attached attributes each of which has a list of handlers. Touching an attribute triggers the corresponding list of handlers. In order to make context switching swift for handlers, systems such as ECLiPSe treats handlers as demons rather than as normal subgoals. A demon 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 demons. 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 failed once need not be tested again. SICStus (Carlsson et al. 1997) provides interfaces for implementing propagators and also some sort of delay construct similar to attributed variables that triggers propagators when events are posted.

AR is an extension of our early delay construct proposed in (Zhou 1998) that allows for the event $dom(X, E)$ and user-defined events. The support of the event $dom(X, E)$ is essential for implementing arc consistency algorithms and also propagators for set constraints (Zhou 2002). Our delay construct is an extension of Meier's delay clause construct (Meier 1994) that allows for 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 (Colmerauer 1984; Naish 1985).

Other rule-based languages have been designed for implementing constraint propagators. 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. 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 to 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 has presented such a language called AR. 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 weak arc-consistency propagator 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 is significantly faster than that of 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 interval consistency. Our solver performs especially well for the benchmarks that contain 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 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: (1) develop new optimization techniques for further avoiding redundant activations of propagators; and (2) implement consistency algorithms beyond interval and arc consistency such as path consistency and Regin's algorithm for `all_distinct`.

Acknowledgement

The author would like to thank Kazunori Ueda and the referees for very detailed and helpful comments on the presentation. This research is supported in part by RF-CUNY and CUNY Software Institute.

References

Aggoun, A. and Beldiceanu, N. 1991. Overview of the CHIP compiler system. In *ICLP'91: Proceedings 8th International Conference on Logic Programming*, K. Furukawa, Ed. MIT Press, 775–789.

Carlsson, M., Ottosson, G., and Carlson, B. 1997. An open-ended finite domain constraint solver. In *Proceedings of Programming Language Implementation and Logic Programming (PLILP)*. LNCS 1292, Springer-Verlag, 191–205.

Codognet, P. and Diaz, D. 1996. Compiling constraints in clp(FD). *Journal of Logic Programming 27,* 3, 185–226.

Colmerauer, A. 1984. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-84)*. ICOT, Tokyo, Japan, 85–99.

Dechter, R. 2003. *Constraint Processing.* Morgan Kaufmann Publishers.

Diaz, D. and Codognet, P. 2001. Design and implementation of the GNU Prolog system. *Journal of Functional and Logic Programming 2001(1)*, 1–29.

Dincbas, M., Simonis, H., and van Hentenryck, P. 1990. Solving large combinatorial problems in logic programming. *Journal of Logic Programming 8*, 75–93. Special Issue: Logic Programming Applications.

Dincbas, M., van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F. 1988. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-88)*. ICOT, 693–702.

Frühwirth, T. 1998. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming 37*, 95–138.

Harvey, W. and Stuckey, P. J. 2003. Improving linear constraint propagation by changing constraint representation. *Constraints: An International Journal 8,* 2, 173–207.

Hentenryck, P. V. 1989. *Constraint Staisfaction in Logic Programming.* MIT Press.

Holzbaur, C. 1992. Metastructures vs. attributed variables in the context of extensible unification. In *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, M. Bruynooghe and M. Wirsing, Eds. LNCS 631, Springer-Verlag, 260–268.

Holzbaur, C., de la Banda, Garcia, M., Stuckey, P. J., and Duck, G. J. 2004. Optimizing compilation of constraint handling rules in HAL. *To appear in Theory and Practice of Logic Programming (TPLP).*

Holzbaur, C. and Fruhwirth, T. 1999. Compiling constraint handling rules into Prolog with attributed variables. In *International Conference on Principles and Practice of Declarative Programming (PPDP)*. LNCS 1702, Springer-Verlag, 117–133.

Jaffar, J. and Maher, M. J. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–582.

Kumar, V. 1992. Algorithms for constraint satisfaction problems: A survey. *AI Magazine 13*, 32–44.

Marriott, K. and Stuckey, P. 1998. *Programming with Constraints: An Introduction.* MIT Press.

Meier, M. 1994. Better late than never. In *Implementations of Logic Programming Systems*, E. Tick and G. Succi, Eds. Kluwer Academic Publishers.

Naish, L. 1985. Negation and control in Prolog. Ph.D. thesis, University of Melbourne.

Puget, J. and Leconte, M. 1995. Beyond the glass box: Constraints as objects. In *Proc. International Logic Programming Symposium.* MIT Press, 513–527.

Puget, J.-F. 1998. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-98)*. AAAI Press, 359–366.

Regin, J. 1994. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the National Conference on Artificial Intelligence(AAAI-94)*. AAAI Press, 362–367.

Schulte, C. 2002. *Programming constraint services: high-level programming of standard and new constraint services*. Lecture Notes in Computer Science, vol. 2302. Springer-Verlag.

Shapiro, E. 1989. The family of concurrent logic programming languages. *ACM Comput. Surveys 21*, 412–510.

Sidebottom, G. and Havens, W. 1996. Nicolog: A simple yet powerful cc(FD) language. *Journal of Automated Reasoning 17*, 371–403.

Tsang, E. 1993. *Foundations of Constraint Satisfaction*. Academic Press.

van Hentenryck, P., Saraswat, V., and Deville, Y. 1992. Constraint Processing in cc(FD). Technical report, Brown University.

Wallace, M., Schimpf, J., Shen, K., and Harvey, W. 2004. On benchmarking constraint logic programming platforms. *Constraints, An International Journal 9,* 1, 5–34.

Warren, D. H. D. 1983. An abstract Prolog instruction set. Technical report, SRI International.

Zhou, N.-F. 1996a. A novel implementation method of delay. In *Proc. Joint International Conference and Symposium on Logic Programming*. MIT Press, 97–111.

Zhou, N.-F. 1996b. Parameter passing and control stack management in Prolog implementation revisited. *ACM Transactions on Programming Languages and Systems 18,* 6, 752–779.

Zhou, N.-F. 1998. A high-level intermediate language and the algorithms for compiling finite-domain constraints. In *Proc. Joint International Conference and Symposium on Logic Programming*. MIT Press, 70–84.

Zhou, N.-F. 2002. Implementing constraint solvers in B-Prolog. In *IFIP World Congress, Intelligent Information Processing*. Kluwer Academic Publishers, 249–260.

Zhou, N.-F. 2003. A high-performance abstract machine for Prolog and its extensions. Technical Report TR-2003014, CUNY Computer Science (www.cs.gc.cuny.edu/tr/).