2006

TR-2006013: The Dom Event and Its Use in Implementing Constraint Propagators

Neng-Fa Zhou

Mark Wallace

Peter J. Stuckey

The `dom` event and its use in implementing constraint propagators

Neng-Fa Zhou[1]*, Mark Wallace[2] and Peter J. Stuckey[3]

[1] CUNY Brooklyn College & Graduate Center
zhou@sci.brooklyn.cuny.edu
[2] Monash University
Mark.Wallace@infotech.monash.edu.au
[3] NICTA Victoria Laboratory
Department of Comp. Sci. and Soft. Eng.
University of Melbourne
pjs@cs.mu.OZ.AU

Abstract. This paper argues the usefulness of the `dom` event in programming several constraint propagators. The `dom` event is introduced for implementing the AC-4 algorithm. For a binary constraint, whenever a value is excluded from the domain of a variable, the propagator with the `dom` event can locate the no-good values in the domain of the other variable in constant time. In this paper we present three application examples of the `dom` event in addition to the AC-4 algorithm for binary support constraints: the `element` constraint, channeling constraints, and set constraints. For each example, we show that the implementation using the `dom` event is significantly more efficient than previous implementations that rely on reification constraints or other techniques.

1 Introduction

Finite domain propagation in constraint (logic) programming systems is a powerful technique for solving combinatorial problems. In [24] an event-handling language called *AR* (Action Rules) is proposed for programming constraint propagators. An action rule specifies, amongst other things, a set of event patterns for events that can activate propagators. Events include instantiations of variables, bounds changes, and events in the form of `dom`(X,E) which means that an inner value E is excluded from the domain of X. A new event pattern, called `dom_any`(X,E), has recently been introduced into AR for capturing the exclusion of any value E from the domain of X.

The rational for having the event pattern `dom`(X,E) is that it facilitates implementing propagators for maintaining arc consistency for functional constraints [10]. For a binary functional constraint, once an inner value is excluded from the domain of a variable, its supporting value in the domain of the other

* Part of this work was conducted while Neng-Fa Zhou visited Monash University and The University of Melbourne in 2005.

variable can be excluded in constant time. Arc consistency of functional constraints is maintained by propagators that watch `dom`(X,E) events together with propagators that handle bounds changes and variable instantiations.

The event pattern `dom_any`(X,E) is introduced for implementing the AC-4 algorithm [15] for general support constraints. For an arbitrary binary support constraint, once a value (either an inner value or a bound) is excluded from the domain of a variable, the counters of those values in the other domain supported by the value can be decremented in constant time [26].

A language construct like the `dom` event is not common in languages for implementing constraint propagators. Many languages and systems support a coarse-grained event, such as the `dom`(X) expression in Sicstus Prolog [3] and GNU-Prolog [5], demons in ECLiPSe, and the `whenDomain` event in CHARME and ILOG [16], which does not capture the excluded value.

Usually in finite domain constraint (logic) programming systems, AC-3 [13] is used to maintain arc consistency of binary constraints. In AC-3, binary constraints are revisited each time one of the domains of the involved variables changes. This accords well with the event notions typically supported by finite domain propagation systems, which are tied to variables but not their values. Propagation system designers are reluctant to introduce fine-grained events into their systems because of the complexity of doing so, and evidence that in many cases the AC-3 algorithm is as efficient as, if not more efficient than, the AC-4 algorithm [20, 22].

This paper presents several novel application examples of the `dom` event. Firstly, we show that with the `dom` event the AC-4 algorithm can be made more efficient than the AC-3 algorithm in practice because most support constraints encountered in practical applications are functional constraints for which no construction of value-based constraint graphs [22] is necessary. Secondly, we show that the `dom` event has applications beyond the AC-4 algorithm. We present three examples: the `element` constraint, channeling constraints, and set constraints. For each example, we show that the implementation using the `dom` event is significantly more efficient than previous implementations that rely on reified constraints or other techniques. The finite-domain and set solvers implemented in AR in B-Prolog are considerably faster than solvers in other constraint logic programming systems[4] and one important reason for the high performance can be attributed to the propagators implemented with the `dom` event.

This paper is organized as follows: Section 2 overviews the AR language and the supported events. Section 3 describes the use of the `dom` event in the implementation of the AC-4 algorithm. Each of the next three sections from 4 through 6 is devoted to an example of the `dom` event. For each example, we present the implementation and compare it with an implementation that uses reification constraints. All the experimental results are obtained with B-Prolog version 6.9 on a Windows XP machine (Intel 1.4GHz CPU, 1GB RAM). Section 7 gives related work and and Section 8 concludes the paper.

[4] See *www.probp.com/benchmark_clpfd.htm* for an up-to-date comparison with ECLiPSe, GNU-Prolog, and Sicstus Prolog.

Readers are assumed to be familiar with constraint logic programming over finite domains, CLP(FD). The de facto standard notation used for finite-domain constraints in major CLP(FD) systems such as ECLiPSe, GNU-Prolog, Sicstus, and B-Prolog is used in this paper. Operators that begin with the symbol `#` denote constraints. So X `#=` Y is an equality constraint, X`#\=`Y a disequality constraint, X `#>=` Y an inequality constraint, X`#=>`Y an entailment Boolean constraint, and X`#<=>` Y a Boolean equivalence constraint. The primitive X `::` D restricts the domain of X to D and the primitive X `notin` D forbids X to take any value from D, where D is an interval $l..u$ or a list of atomic values.

2 Action Rules and Events

The AR (*Action Rules*) language is designed to facilitate the specification of event-driven functionality needed by applications such as constraint propagators and graphical user interfaces where interactions of multiple entities are essential [24]. 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 arbitrary subgoals. An action rule degenerates into a *commitment rule* if *Event* together with the enclosing braces are missing. In general, a predicate can be defined with multiple action rules. For the sake of simplicity, we assume in this paper that each predicate is defined with only one action rule possibly followed by a sequence of commitment rules.

Definition 1. *A subgoal is called an agent if it can be suspended and activated by events. For an agent α, a rule "H, C, $\{E\}$ => B" is applicable to the agent if there exists a matching substitution θ such that $H\theta = \alpha$ and the condition $C\theta$ is satisfied.*

When an agent is created, the system checks if the action rule in its predicate is applicable to it.[5] If so, the agent will be suspended until it is *activated* by one of the events specified in the rule. Whenever the agent is activated by an event, the condition of the action rule is tested *again*. If it is met, the action is executed. The agent does not vanish after the action is executed, but instead sleeps until it is activated again. There is no primitive for killing agents explicitly. An agent vanishes only when a commitment rule is applied to it. The reader is referred to [24] for a detailed description of the language and its operational semantics.

The following event patterns are supported for programming constraint propagators:

[5] Notice that since one-directional matching rather than full-unification is used to search for an applicable rule and only tests are allowed in the condition, the agent will remain the same after an applicable rule is found.

- `generated`: After an agent is generated but before it is suspended for the first time. The sole purpose of this pattern is to make it possible to specify preprocessing and constraint propagation actions in one rule.
- `ins`(X): when the variable X is instantiated.
- `bound`(X): when a bound of the domain of X is updated. There is no distinction between lower and upper bounds changes.
- `dom`(X,E): when an *inner* value E is excluded from the domain of X. Since E is used to reference the excluded value, it must be the first occurrence of the variable in the rule.
- `dom`(X): same as `dom`(X,E) but the excluded value is ignored.
- `dom_any`(X,E): when an arbitrary value E is excluded from the domain of X. Unlike in `dom`(X,E), the excluded value E here can be a bound of the domain of X.
- `dom_any`(X): equivalent to the disjunction of `dom`(X) and `bound`(X).

Note that when a variable is instantiated, no `bound` or `dom` event is posted. Consider the following example:

```
p(X),{dom(X,E)} => write(dom(E)).
q(X),{dom_any(X,E)} => write(dom_any(E)).
r(X),{bound(X)} => write(bound).
go:-X :: 1..4, p(X), q(X), r(X), X #\= 2, X #\= 4, X #\= 1.
```

The query `go` gives the following outputs: `dom(2)`, `dom_any(2)`, `dom_any(4)` and `bound`.[6] The outputs `dom(2)` and `dom_any(2)` are caused by `X #\= 2`, and the outputs `dom_any(4)` and `bound` are caused by `X #\= 4`. After the constraint `X #\= 1` is posted, `X` is instantiated to `3`, which posts an `ins(X)` event but not a `bound` or `dom` event.

A rule is allowed to specify multiple event patterns, but the `dom`(X,E) and `dom_any`(X,E) patterns are allowed to co-exist with `ins` patterns only. For each co-existing `ins`(X) pattern, there must be a condition `var`(X) in the guard so that the action is never executed when the rule is triggered by an `ins` event.

Note also that the `dom_any`(X,E) event pattern should be used only on small-sized domains. If used on large domains, constraint propagators could be flooded with a huge number of `dom_any` events. For instance, for the propagators defined in the previous example, the query

```
X :: 1..1002, q(X), X #>1000
```

posts 1000 `dom_any` events, while it would post only one `bound` event if `q(X)` were `p(X)` or `r(X)`. For this reason, in B-Prolog propagators for handling `dom_any`(X,E) events are generated only after constraints are preprocessed and the domains of variables in them become small.

For each event type, each domain variable has a slot for the list of watching propagators. Therefore, the `dom` event imposes little space overhead: one slot

[6] In the implementation of AR in B-Prolog, when more than one agent is activated the one that was generated first is executed first. This explains why `dom(2)` occurs before `dom_any(2)` and also why `dom_any(4)` occurs before `bound`.

for `dom`(X, E) and another slot for `dom_any`(X, E) for each domain variable X. There is almost no time overhead because an event is posted only when the watching list is not empty.

3 The AC-4 Algorithm

The AC-3 algorithm [13] is a naive algorithm for maintaining arc consistency of constraints. For each pair of variables (X_i,X_j) connected in the constraint network, if any value is excluded from the domain of X_i, all the arcs in the network pointing to X_i are examined. The AC-4 algorithm [15] is a semi-naive algorithm for maintaining arc consistency. It propagates updates of values more intelligently: whenever a value is removed from the domain of a variable X_i, it only examines the values in the domains of the connected variables of X_i that are supported by the value. The AC-5 algorithm [10] specializes the AC-4 algorithm by taking the semantics of constraints into account. The AC-5 algorithm has a lower complexity than the AC-4 algorithm for proprocessing certain types of constraints to achieve arc consistency. As far as maintaining arc consistency during search is concered, there is no difference between AC-5 and AC-4.

The `dom`(X,E) event is introduced for implementing the AC-4 algorithm for binary functional constraints. For a functional binary constraint, there is only one supporting value for each value in a domain. Therefore, whenever a value is excluded from a domain, we only need to exclude its counterpart in the other domain to maintain arc consistency. Consider, for example, the constraint `X+Y #= C` where `X` and `Y` are domain variables and `C` is an integer. The propagator defined in the following propagates exclusions of values from the domain of `Y` to `X` to achieve arc consistency:

```
'X in C-Y_ac'(X,Y,C),var(X),var(Y),
    {dom(Y,Ey)}
    =>
    Ex is C-Ey,
    X #\= Ex.
'X in C-Y_ac'(X,Y,C) => true.
```

For the original constraint `X+Y #= C`, we need to generate two propagators, namely, `'X in C-Y_ac'(X,Y,C)` and `'X in C-Y_ac'(Y,X,C)`, to maintain the arc consistency. Note that in addition to these two propagators, we also need to generate propagators for maintaining interval consistency since no `dom(Y,Ey)` event is posted if the excluded value happens to be a bound. Note also that we need to preprocess the constraint to make it arc consistent before the propagators are generated.

For general binary constraints defined as extensional tables, each value in the domain of a variable can have multiple supporting values in the domain of the other variable. We set up a counter for each value in each domain for counting the support values in the other domain. Whenever the counter of a value

becomes zero, the value is excluded from its domain. So the job of maintaining arc consistency reduces to maintaining the counters.

Let `BinaryRelation` be a representation of the binary relation on two variables `X` and `Y`. An efficient data structure such as a hash table is used for the representation such that for each value in the domain of `X`, it takes constant time to retrieve its supporting values and their associated counters. The propagator for maintaining `Y`'s counters can be implemented easily as follows:

```
ac4(BinaryRelation,X,Y),var(X),var(Y),
    {dom_any(X,E)}
    =>
    decrement_counters(BinaryRelation,E,Y).
ac4(BinaryRelation,X,Y) => true.
```

Whenever a value `E` is excluded from the domain of `X`, the counters of the values in the domain of `Y` supported by `E` are decremented. If the counter of a value becomes zero, the value is excluded from the domain of `Y`.

It is also possible to implement the AC-3 algorithm in action rules. For example, the propagator based on the AC-3 algorithm for the constraint `X+Y #= C` can be implemented as follows:

```
'X in C-Y_ac'(X,Y,C),var(Y),
    {dom(Y),bound(Y),ins(Y)}
    => remove_no_good(X,Y,C).
'X in C-Y_ac'(X,Y,C) => X is C-Y.
```

where `remove_no_good(X,Y,C)` removes all no-good values from the domain of `X` that are not supported by any values in the domain of `Y`. Because the propagator does not have the information about what values are excluded, it has to go through the domain elements of `X` to locate possible no-good values.

As for the constraint `X+Y #= C`, the implementation of the AC-4 algorithm is clearly faster than the implementation of the AC-3 algorithm since updates of domains are propagated from one to another in constant time in the AC-4 algorithm and no value-based constraint graph is needed [22]. This is true for binary functional constraints in general. For a general binary support constraint, the propagator `ac4(BinaryRelation,X,Y)` takes an extra argument `BinaryRelation` which corresponds to the value-based constraint graph. Nevertheless, the construction of such a graph affects the complexity of the preprocessing phase but not the search phase.

Computational results

To experimentally compare the performance of the two implementations of `'X in C-Y_ac'`(X,Y,C), we use the propagator to maintain the arc consistency of the constraint X `#=` $Y+1$, where initially X and Y are defined over $0..n$ for a given n and then the values from 2 to $n-1$ are removed from the domain of Y. The test program follows. In order for B-Prolog to maintain arc consistency

Table 1. Comparing the AC-3 and AC-4 implementations (CPU time).

N	*AC-3* (ms)	*AC-4* (ms)	$\frac{AC-3}{AC-4}$
5000	696.79	1.26	550.38
6000	1006.30	1.53	657.27
7000	1373.40	1.73	791.58
8000	1792.20	2.01	888.98
9000	2264.10	2.31	979.27
10000	2806.19	2.51	1115.33

of the constraint, the domains must be represented as bit vectors when a hole occurs. The call `fd_vector_min_max(0,N)` resets the range such that the domains are represented as bit vectors for any given `N`.

```
go(N):-
    fd_vector_min_max(0,N),
    [X,Y] in 0..N,
    'X in C-Y_ac'(X,Y,1),
    N1 is N-2,
    make_holes(Y,2,N1).

make_holes(X,I,N):-I=:=N,!.
make_holes(X,I,N):-
    X #\= I,
    I1 is I+1,
    make_holes(X,I1,N).
```

Table 1 compares the time taken by the two implementations for different domain sizes. The AC-4 implementation clearly outperforms the AC-3 implementation. In general, the AC-4 implementation takes linear time in the size of the number of holes while the AC-3 implementation takes quadratic time.

4 The Constraint `element`(I,L,X)

The constraint `element`(I,L,X) means that the Ith element of L is X, where I must be an integer or a domain variable, X a domain variable, and L a list of terms. The original version of the constraint presented in [2] requires X and the elements of L to be integers or domain variables. Here we consider the special case where L is restricted to be a list of ground terms. Notice that with this restriction, the constraint degenerates into a binary one but not necessarily a functional one. So the propagators given in this section are another application of the AC-4 algorithm.

Let L be a list of ground elements `[`L_1, ..., L_n`]`. Then I must be in the range of $1..n$. On one hand, each value in the domain of I must be supported by X. As long as X is known not to be equal to some element L_i, i can be excluded from the domain of I. This relationship can be expressed by the following entailment

constraint X `#\=` L_i `#=>` I `#\=` i. On the other hand, each value in the domain of X must be supported by values in the domain of I as well. We use a counter C_{L_i} for L_i that tells the number of occurrences of L_i in L. Each time a value i is excluded from the domain of I, the counter C_{L_i} is decremented. If it becomes zero, then we can post the constraint X `#\=` L_i.

The entailment constraints X `#\=` L_i `#=>` I `#\=` i $(1 \leq i \leq n)$ can be encoded using only one propagator thanks to the availability of the `dom_any` event. To achieve this, we represent L as an association map such that for each L_i its indexes and counter can be returned in constant time. The following shows the propagator:

```
element_X_to_I(X,I,Map),var(X),
   {dom_any(X,E)}
   =>
   map_get_indexes(Map,E,Indexes),
   I notin Indexes.
element_X_to_I(X,I,Map) => true.
```

Whenever a value `E` is excluded from `X`'s domain, the constraint `I notin Indexes` ensures that `I` cannot take the index of any occurrence of `E`. When `X` is instantiated to be a non-variable term, the propagator vanishes.

The propagation from I to X can be done using only one propagator as well. Let `Vect` be a vector representation of L with which the element of a given index can be returned in constant time. The following defines the propagator:

```
element_I_to_X(I,X,Vect,Map),var(I),
   {dom_any(I,E)}
   =>
   arg(E,Vect,Le),
   decrement_counter(Map,X,Le).
element_I_to_X(I,X,Vect,Counters) => true.
```

The call `decrement_counter(Map,X,Le)` decrements the counter of `Le` and posts the constraint `X #\= Le` if `Le`'s counter becomes zero.

In addition to the two propagators `element_X_to_I` and `element_I_to_X`, we need two extra propagators to handle `ins`(I) and `ins`(X) events. When I is instantiated to an integer i, the constraint X `#=` L_i is generated, and when X is instantiated, the domain of I is reduced to contain only the indexes of the occurrences of X in L.

Computational results

We experimentally compared the performance of the two implementations of the `element` constraint. We could find only two benchmark programs: one called *Cars*, which uses the `element` constraint for sequencing cars in assembly lines [9] and the other for solving a cutting stock problem [6]. To deal with the scarcity of programs, we implemented the `alldifferent` and `permutation` constraints

Table 2. Comparing the two implementations of `element` (CPU time).

Program	*dom_any* (ms)	*bool* (ms)	$\frac{bool}{dom\_any}$
Cars	4.70	7.19	1.52
Cutting-stock	278.10	1145.3	4.11
Sudoku 250 (element)	101.50	160.90	1.58
Sort	304.69	495.30	1.62

using the `element` constraint for this comparison. With these two constraints, we could use two new benchmarks: *Sudoku* for solving a Sudoku puzzle and *Sort* for sorting a randomly generated list using the `permutation` constraint.

For this comparison, the `alldifferent`(Xs) constraint is artificially implemented using the `element` constraint as follows:[7] Let Xs be a list of variables [X_1, ..., X_n] where each variable X_i has the domain $1..n$, and let L be the list of integers [1, ..., n]. We use the following n `element` constraints to encode `alldifferent`(Xs):

`element`(I_1,L,X_1) ... `element`(I_n,L,X_n)

where each pair I_i and I_j ($i \neq j$) are constrained to take different values by separate disequalities, so in reality we are simply mapping `alldifferent` on a set of variables to `alldifferent` on their indexes. The propagators for `element` are tested because the original set of variables Xs, rather than the new set, is enumerated.

The `permutation`(L,P) constraint is true if P is a permutation of L. It is encoded for this comparison by using `element` as follows:

```
permutation(L,P):-
    length(L,N), length(Is,N),
    permutation(L,P,Is),
    alldifferent(Is).

permutation(L,[],[]).
permutation(L,[X|Xs],[I|Is]):-
    element(I,L,X),
    permutation(L,Xs,Is).
```

Table 2 reports the results. The column *dom_any* shows the time taken by the implementation that uses the `dom_any` event, while the column *bool* shows the time taken by the implementation that uses reified constraints to propagate information from `X` to `I`. The propagator `element_I_to_X` cannot be encoded easily using reified constraints. For this reason, this propagator is encoded using the `dom_any` event in both implementations.

It is not surprising that *dom_any* outperforms *bool* since the exclusion of each value from X activates only one propagator in *dom_any* while it activates n (the

[7] This program is not intended to be another implementation of `alldifferent` but is written only for the benchmarking purpose.

size of L in `element`(I,L,X)) propagators in the implementation that uses reified constraints.

5 Channeling Constraints

For certain problems, e.g., permutation problems where there are as many values as variables and each variable takes an unique value [4, 21], it is possible to have dual models and use channeling constraints to relate the two models. Using channeling constraints may increase the pruning power of constraint propagation [4]. Channeling constraints can be expressed as Boolean constraints. In this section we show that with the `dom` event we can use significantly fewer propagators to implement channeling constraints.

As an example, we consider the permutation channel `primal_dual`(Xs,Ys) where Xs and Ys are n variables ranging over the domain $1..n$, and they satisfy the following relationship:

$$\forall_{i,j}(X_i = j \Leftrightarrow Y_j = i) \text{ or equivalently } \forall_{i,j}(X_i \neq j \Leftrightarrow Y_j \neq i)$$

We can use the `primal_dual` constraint for improving propagation of `alldifferent` constraints. Many algorithms have been proposed for maintaining different levels of consistency for `alldifferent` [19]. The filtering algorithm by Regin [17] achieves hyper-arc consistency. However, because of the almost cubic order of complexity, many CLP(FD) systems such as B-Prolog and ECLiPSe employ Hall-set finding algorithms.[8]

Since there are an exponential number of potential Hall sets, we have to rely on some heuristics to choose what sets to test. In the implementation in B-Prolog, whenever the domain of a variable is updated, a propagator is activated to check if the updated domain is a Hall set [24]. Understandably, since no union of domains is considered, this heuristic has its limitations. Consider, for example, the constraint `alldifferent`($[X_1,X_2,X_3,X_4]$) where $X_1 \in \{1,2\}$, $X_2 \in \{1,3\}$, $X_3 \in \{2,3\}$, and $X_4 \in \{1,2,3,4\}$. The heuristic fails to find the Hall set $\{1,2,3\}$ and thus fails to bind X_4 to 4.

Using channeling can increase pruning power. By adding the constraints `primal_dual`(Xs,Ys) and `alldifferent`(Ys), the dual variables have the following domains: $Y_1 \in \{1,2,4\}$, $Y_2 \in \{1,3,4\}$, $Y_3 \in \{2,3,4\}$, and $Y_4 \in \{4\}$. After Y_4 is instantiated to 4, X_4 is instantiated to 4 as well. As demonstrated by this example, using dual models can to some extent remedy the limitation of the Hall-set finding algorithm.

The channeling constraint between primal and dual variables $\forall_{i,j}(X_i \neq j \Leftrightarrow Y_j \neq i)$ can be represented as Boolean constraints. Since for each primal variable X_i and each dual variable Y_j one Boolean constraint is needed to connect them, in total n^2 Boolean constraints are needed.

[8] For the constraint `alldifferent`($[X_1,\ldots,X_n]$) where X_i has the domain D_i ($1 \leq i \leq n$), a set H is a Hall set if the number of subsets of H among $D_1, \ldots, D_n$ is greater than or equal to the size of H. Formally, H is a Hall set if $|\{D_i \mid D_i \subseteq H\}| \geq |H|$.

With the `dom` event, we can use only $2 \times n$ propagators to implement the channeling constraint. Let `DualVarVector` be a vector created from the list of dual variables. For each primal variable `Xi` (with the index `I`), a propagator defined below is created to handle exclusions of values from the domain of `Xi`.

```
primal_dual(Xi,I,DualVarVector),var(Xi),
   {dom_any(Xi,J)}
   =>
   arg(J,DualVarVector,Yj),
   Yj #\= I.
primal_dual(Xi,I,DualVarVector) => true.
```

Each time a value `J` is excluded from the domain of `Xi`, assume `Yj` is the `J`th variable in `DualVarVector`, then `I` must be excluded from the domain of `Yj`. We need to exchange primal and dual variables and create a propagator for each dual variable as well. Therefore, in total $2 \times n$ propagators are needed.

Note that a preprocessing phase is needed to ensure that the channeling constraints are consistent before any propagator is generated. The preprocessing phase takes $O(n^2)$ time.

Computational results

Table 3 compares the performance of the two encodings of channeling constraints on three benchmarks:

- *Sudoku*: This program contains only `alldifferent` constraints. Dual models and the Hall-set finding algorithm described above are used for the constraints. Two problem instances are tested: one with 250 variables and the other with 368 variables.
- *Queens*: Dual models are used to solve the 100-queens problem.
- *Hamilton*: This program finds a Hamilton circuit in a graph. It contains the `circuit` constraint. The `circuit`(L) constraint, which is the same as the `cycle`(L) constraint introduced in [2], entails `alldifferent`(L). A valuation L=$[X_1,\ldots,X_n]$ satisfies the constraint if the list of arcs $[1 \rightarrow X_1,\ldots,n \rightarrow X_n]$ forms a Hamilton cycle. Dual models are used for the `circuit` constraint.

The column *dom_any* shows the time taken by the implementation that uses the `dom_any` event, and the column *bool* shows the time taken by the implementation that uses equivalent Boolean constraints for relating primal and dual variables. Note that the search space explored is the same in each case.

The speed-ups for channeling constraints are higher than those for the `element` constraint. The speed-up for Sudoku-368 is over 30. Just as the results for the `element` constraint, *dom_any* is much faster than *bool* since the exclusion of each value from a domain activates only one propagator in *dom_any* rather than a linear number of propagators as in *bool*.

Table 3. Comparing the two implementations of channeling constraints (CPU time).

Program	*dom_any* (ms)	*bool* (ms)	$\frac{bool}{dom\_any}$
Sudoku-250 (alldifferent)	71.90	275.00	3.82
Sudoku-368 (alldifferent)	110.00	3422.00	31.10
Queens	51.60	468.80	9.08
Hamilton	737.50	1487.5	2.01

6 Set Constraints

As the final example, we present a set solver implemented using the `dom` event. One of the key issues in implementing set constraints concerns how to represent set domains. Since a set of size n has 2^n subsets, it is unrealistic to enumerate all the values in a domain and represent them explicitly when n is large. Our solver inherits the interval representation scheme for set domains from Conjunto [8], but represents the lower and upper bounds as two finite domain variables rather than as two sorted lists. In our representation, originally presented in [25], updates of bounds of set domains can be captured in constant time and propagated to other domains quickly thanks to the availability of the `dom` event.

Let V be a set variable. We use the following notations to reference the attributes: V^l for the lower bound, V^u for the upper bound, V^c for the cardinality, and V^{univ} for the universal set. V^l is represented as a finite-domain variable whose domain is the *complement* of the set of all *definite* elements that are known to be in V, V^u is represented as a finite-domain variable whose domain is the set of *possible* elements of V. V^c is represented as a domain variable whose domain is $1..|V^{univ}|$. To prevent V^l and V^u from being instantiated, we include two dummy elements in them that are not in the universal set. This representation facilitates updates of bounds of set domains. Both updates of lower and upper bounds can be modeled as `dom` events.

For example, consider the set variable V over the domain $\{1\}..\{1,2,3\}$. V^l is a finite-domain variable with the domain [0,2,3,4] (the complement of $\{1\}$ is $\{2,3\}$) and V^u has the domain [0,1,2,3,4] where 0 and 4 are dummy elements. Suppose 2 is known to be an element of V. Updating the lower bound means excluding 2 from V^l, which results in a new lower domain [0,3,4]. Suppose 3 is known to be an infeasible element of V. Updating the upper bound means excluding 3 from V^u, which results in a new upper domain [0,1,2,4].

The complete set of rules for maintaining interval consistency for set constraints is given in [25]. The following gives the two rules for maintaining the bounds consistency of the subset constraint $R \subseteq S$:

$$\text{if } x \in R \text{ then } x \in S \qquad \text{if } x \notin S \text{ then } x \notin R$$

Whenever an element x is added into R, it must be added into S as well; and whenever an element x is excluded from the domain of S, it must be excluded from the domain of R as well. The two propagation rules can be implemented in

the following way, where for simplicity we assume a set variable S is represented as a term `set(`S^l`,`S^u`)` containing two domain variables:[9]

```
subset_from_R_to_S(set(Rl,_Ru),S),
    {dom(Rl,E)}
    =>
    clpset_add(S,E).

subset_from_S_to_R(R,set(_Sl,Su)),
    {dom(Su,E)}
    =>
    clpset_exclude(R,E).
```

Where `clpset_add`(S,E) adds the element E into the lower bound of S by excluding it from S^l and `clpset_exclude`(R,E) removes E from the upper bound of R by excluding it from R^u. Note that because of the existence of dummy elements, no bound of the finite-domain variables R^l or S^u will ever change, and therefore the use of the `dom` event pattern rather than the `dom_any` pattern suffices.

The propagator `subset_from_R_to_S(R,S)` would have to be encoded as reification constraints as follows if the `dom` event were not available: For each element `E` in the domain of the lower bound `Rl` of `R`,

```
Rl #\= E #=> Flag #=1, freeze(Flag,clpset_add(S,E))
```

Instead of one propagator with the `dom` event, we need a linear number of propagators to implement the propagation rule.

Computational results

Table 4 reports the comparison results of the two implementations of set constraints: The column *dom* shows the time taken by the implementation that uses the `dom` event, and the column *bool* shows the time taken by the one that uses reification constraints. The following benchmark programs are used:

- *Steiner*: The ternary Steiner problem of order n is to find $n(n-1)/6$ sets over the universal set $\{1, 2, ..., n\}$ such that each set contains three elements and any two sets have at most one element in common. This program was taken from [8]. No constraint for breaking symmetry is used.
- *Golf*: This is taken from the ECLiPSe sample program suite. It schedules a round-robin golf tournament on which each player plays in a group in every round and each player can only play with the same person once.

For Steiner, *dom* is 23 times as fast as *bool*, and for Golf, *dom* finds a solution in 609 milliseconds while *bool* fails to find one in 1000 seconds because of repeated invocations of the garbage collector.

[9] In the real implementation in B-Prolog, a set domain variable is represented as an attributed variable with the lower and upper bounds attached as attributes.

Table 4. Comparing the two implementations of set constraints (CPU time).

Program	*dom* (ms)	*bool* (ms)	$\frac{bool}{dom}$
Steiner (9)	125	2,950	23.6
Golf (32-9-8)	609	$> 1^6$	$> 1{,}600$

Table 5 compares the performance of two set solvers: the *fd_sets* solver as provided in ECLiPSe 5.8 #107, and the BP set solver presented in this paper as provided in B-Prolog 6.8. Both *fd_sets* and the BP set solver adopt the same domain representation originally presented in [25]. The propagation rules from the Conjunto solver [8] are implemented in both solvers.

The BP set solver is significantly faster than for both programs although the same domain representation and propagation rules are used in both solvers. In *fd_sets*, set domain variables are represented as attributed variables and propagation rules are encoded in demons. The speed difference is mainly caused by the lack of a swift mechanism in ECLiPSe for handling the `dom` event.

The *fd_sets* solver is several times faster than the Conjunto solver. In Conjunto, set bounds are represented as sorted lists and it takes linear time in the worse case to update a bound.

Table 5. Comparison of two set solvers (CPU time).

Program	*fd_sets* (ms)	*BP* (ms)	$\frac{fd\_sets}{BP}$
Steiner(9)	2,025	125	16.19
Golf	2,000	609	3.28

7 Related Work

The AR language is an extension of delay clauses [14, 23] for supporting events and actions. In very early versions, only `ins`, `bound` and `dom`(X) events were supported. The `dom`(X, E) event was first introduced into AR in year 2000 [25] for implementing propagators for set constraints. An elaboration of the use of the AR language in programming basic propagators for arithmetic and alldifferent constraints is given in [24]. The `dom_any`(X, E) event is new in this paper.

This paper is a successor of [24], which demonstrates the use of the `dom` event in propagators for arbitrary binary support constraints, the element constraints, channeling constraints and set constraints. The propagators presented in this paper could be implemented with ease in any language that supports `dom`-like events such as CHOCO [12][10]. ILOG solver allows a propagator to access a domain delta structure which records the changes in the domain of a variable since the last time all propagators on the variable were executed. This allows

[10] Since the original workshop paper on CHOCO is unavailable and the manual of the Choco system does not mention similar event constructs, we cannot give a detailed comparison.

access to the deleted values of a variable in an approximate way, and hence an indirect support of the `dom` event.

Most constraint logic programming systems such as Eclipse, SICStus, and GNU-Prolog provide non-value specific events similar to `ins`(X), `bound`(X) and `dom`(X). In ECLiPSe, a finite-domain variable has an attribute called `hole` and demons can be attached to the attribute. Whenever inner values are excluded from a domain, i.e., whenever values are added into the hole, the attached demons are activated. In Sicstus and GNU-Prolog, a range expression or wakeup condition of the form `dom`(X) can be used, which activates the associated propagators whenever the domain of X is updated. Nevertheless, in these systems no value can be transmitted to the propagators and thus it is impossible to achieve the same effect as the `dom`(X, E) or `dom_any`(X, E) event.

cc(FD) [11] compiles a functional constraint into Boolean implication constraints to maintain its arc consistency. An optimization technique is then used to combine implication constraints to achieve better space efficiency. But, as far we know, cc(FD) provides no construct like the `dom` event to the user for implementing arc consistency algorithms.

CHR (Constraint Handling Rules) [7] has been used to implement various kinds of constraint propagators (see e.g., [1]). If events are treated as constraints, then all the propagators presented in this paper could be translated into CHR. Treating events as constraints, however, can hardly achieve the same performance. Events are removed automatically after all the watching agents are activated. In CHR, however, there must be rules to remove events explicitly. Recently it has been found that action rules can serve as an efficient alternative intermediate language for compiling CHR [18].

There are two reasons for the reluctance of introducing `dom`-like construct into CLP languages for implementing propagators. Firstly, in register-based abstract machines like the WAM a considerable cost must be paid to pass an extra value to a propagator when it is activated. In those systems, propagators are stored as terms on the heap and the arguments must be rearranged into appropriate registers before they can be executed. In B-Prolog, in contrast, propagators are stored as stack frames, and passing an extra value into a propagator means placing it in a designated slot in the frame [23, 24]. Therefore, the overhead of the `dom` event is extremely small in B-Prolog. It remains an open issue how to implement the `dom` event in a register-based machine with low overhead.

The second reason why the `dom`-like construct has not been widely accepted is because of the perception that maintaining interval consistency is efficient enough in practice and even for those problems that do require arc consistency the AC-3 algorithm is as efficient as, if not more efficient than, the AC-4 algorithm [20, 22]. As reported in the computational results in Section 3, the difference between the time complexities of the AC-3 and AC-4 algorithms is significant for functional constraints although it can be erased in theory for general constraints [20, 22]. For example, the AC-3.1 algorithm [22] does not perform better than original AC-3 algorithm for bi-directional functional con-

straints since each value has only one supporting value in the other domain and there is no need to remember the resumption point for each value.

8 Concluding Remarks

We have presented several application examples, for which the `dom` event facilitates propagating updates of domains and/or makes it possible to describe a relationship with an-order-of-magnitude fewer propagators. The contributions of this paper are as follows:

- It describes the `dom` event and illustrates its use in the implementation of the AC-4 algorithm for not only functional constraints but also arbitrary support constraints and the `element` constraint.
- It proposes an innovative use of the `dom` event in encoding propagators for channeling and set constraints. As far as we know, no similar attempt has been made by other authors.
- It gives experimental results to confirm the importance of the `dom` event.

The availability of the `dom` event together with an efficient mechanism for handling it is a key factor for the high performance of the finite-domain and set solvers in B-Prolog.

We believe that the `dom` event can be found useful in more applications such as global constraints. For example, in the incremental version of Regin's filtering algorithm [17], the `dom` event could be used to detect if an edge in the current maximal match has been removed. The `dom` event can also be used in propagators for problem-specific constraints. The future work is to explore new applications.

References

1. Krzysztof R. Apt and Eric Monfroy. Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming*, 1(6):713–750, 2001.
2. N. Beldiceanu and E. Contjean. Introducing global constraints in CHIP. *Mathematical and Computer Modeling*, 12:97–123, 1994.
3. Mats Carlsson, Greger Ottosson, and Björn Carlson. An Open-Ended Finite Domain Constraint Solver. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *PLILP'97: Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206, Southhampton, UK, September 1997. Springer.
4. B. M. W. Cheng, Kenneth M. F. Choi, Jimmy Ho-Man Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4(2):167–192, 1999.
5. D. Diaz and P. Codognet. Design and implementation of the GNU Prolog system. *Journal of Functional and Logic Programming*, 2001(1):1–29, 2001.
6. Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving a cutting-stock problem in constraint logic programming. In *ICLP/SLP*, pages 42–58, 1988.

7. Th. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37:95–138, 1998.
8. Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
9. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
10. Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artif. Intell.*, 57(2-3):291–321, 1992.
11. Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). *J. Log. Program.*, 37(1-3):139–164, 1998.
12. M. Laburthe. Choco: implementing a cp kernel. In *Proceedings of the CP00 Post Conference Workshop on Techniques for Implementing Constraint Programming Systems*, pages 71–85, 2000.
13. Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
14. M. Meier. Better late than never. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*. Kluwer Academic Publishers, 1994.
15. Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
16. J.F. Puget and M. Leconte. Beyond the glass box: Constraints as objects. In *Proc. International Logic Programming Symposium*, pages 513–527. MIT Press, 1995.
17. J.C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the National Conference on Artificial Intelligence(AAAI-94)*, pages 362–367. AAAI Press, 1994.
18. Tom Schrijvers, Neng-Fa Zhou, and Bart Demoen. Translating constraint handling rules into action rules. In *The Third Workshop on Constraint Handling Rules*, 2006.
19. W J van Hoeve. The alldifferent constraint: A survey, 2001.
20. Richard J. Wallace. Why AC-3 is almost always better than ac4 for establishing arc consistency in csps. In *Proceedings of IJCAI*, pages 239–247, 1993.
21. Toby Walsh. Permutation problems and channelling constraints. In *LPAR*, pages 377–391, 2001.
22. Yuanlin Zhang and Roland H.C. Yap. Making AC-3 an optimal algorithm. In *Proceedings of IJCAI*, pages 316–321, 2001.
23. Neng-Fa Zhou. A novel implementation method of delay. In *Proc. Joint International Conference and Symposium on Logic Programming*, pages 97–111. MIT Press, 1996.
24. Neng-Fa Zhou. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming (TPLP)*, 2006.
25. Neng-Fa Zhou and Joachim Schimpf. Implementation of propagation rules for set constraints revisited. *http://www.sci.brooklyn.cuny.edu/~ zhou/papers/clpset.pdf,* Unpublished manuscript, preliminary results published in *First Workshop on Rule-Based Constraint Reasoning and Programming, 2000*, 2000.
26. Neng-Fa Zhou and Mark Wallace. A simple constraint solver in action rules for the cp'05 solver competition. In *Proceedings of the CP workshop on Constraint Propagation and Implementation*, 2005.