The dom Event and its Use in Implementing Constraint Propagators

Neng-Fa Zhou\(^1\), Mark Wallace\(^2\) and Peter Stucky\(^3\)

\(^1\) CUNY Brooklyn College & Graduate Center
zhou@sci.brooklyn.cuny.edu
\(^2\) Monash University
Mark.Wallace@infotech.monash.edu.au
\(^3\) The University of Melbourne
pjs@cs.mu.OZ.AU

Abstract. This paper argues the usefulness of the dom event in programming 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: 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

In [15] an event-handling language called AR (Action Rules) is proposed for programming constraint propagators. An action rule specifies, amongst other things, an event pattern 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\(_\text{any}\)(\(X,E\)), has recently been introduced for capturing the exclusion of any value \(E\) from the domain of \(X\). In the following of this paper, we use the term dom to refer to either dom\((X,E)\) or dom\(_\text{any}\)(\(X,E\)) if the distinction is not important.

The rational for having the event pattern dom\((X,E)\) is that it facilitates implementing propagators for maintaining arc consistency for functional constraints [8]. 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 variable can be excluded in constant time. The event pattern dom\(_\text{any}\)(\(X,E\)) is introduced for implementing the AC-4 algorithm [5] for general support constraints. For a non-functional binary constraint, once an arbitrary value (i.e. 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 [14].

A language construct like the dom event is not common in languages for
implementing constraint propagators. Language designers are reluctant to intro-
duce it due to 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 al-
gorithm [11, 13]. This paper aims to argue the importance of the dom event in
implementing constraint propagators. Firstly, we show that with the dom event
the AC-4 algorithm can be made more efficient than the AC-3 algorithm in prac-
tice because most support constraints encountered in practical applications are
functional constraints for which no construction of value-based constraint graphs
[13] is necessary. Secondly, we show that the dom event has applications beyond
the AC-4 algorithm. We present three application examples: the element con-
straint, 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.

This paper is organized as follows: Section 2 overviews the AR language and
the dom event. 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 application example of the dom event. For each example, we present
the implementation and compare it with the current implementation that does
not use the dom event. All the experimental results are obtained with B-Prolog
version 6.8 on a Windows XP machine. Section 7 gives related work and Section
8 concludes the paper.

Readers are assumed to be familiar with CLP(FD). The de facto standard
notation used for finite-domain constraints in major CLP(FD) systems such as
B-Prolog, ECLiPSe, GNU-Prolog, and Sicstus 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 an equivalence Boolean
constraint. The primitive X :: D restricts the domain of X to D and the
primitive Xnotin 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 Event Patterns

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
[15]. An action rule takes the following form:

\[
Agent, Condition, \{Event\} =\rightarrow Action
\]

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

2
of patterns for events that can activate the agents, and *Action* is a sequence of arbitrary subgoals. An action rule degenerates to a *commitment rule* if *Event* together with the enclosing braces are missing.

When an agent is created, the system searches in its definition for a rule whose agent pattern *matches* the agent and *Condition* is satisfied. This kind of rules is said to be *applicable* to the agent. Notice that since one-directional matching rather than full-unification is used to search for an applicable rule and only tests are allowed in *Condition*, the agent will remain the same after an applicable rule is found. After an applicable action rule is found, the agent will be suspended until it is *activated* by one of the events specified in *Event*. If the agent is suspended for the first time and *generated* is included in *Event*, then *Action* will be executed before the agent is suspended. Whenever the agent is activated by an event, *Condition* is tested again. If it is met, *Action* will be executed. A failure of *Action* will cause the agent to fail. The agent does not vanish after *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 [15] for a detailed description of the language.

The following event patterns are supported for programming constraint propagators:

- *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 is not allowed to occur in *Agent* or *Condition*. This pattern is not allowed to co-exist with any other pattern in a rule because otherwise *E* in *Action* may refer to no value if *Agent* is activated by a different event.
- *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*. Like *dom*(X,E), this pattern is not allowed to co-exist with any other pattern in a rule.
- *dom*(X): same as *dom*(X,E) but the excluded value is not of a concern. This pattern is allowed to co-exist with *generated* and other single-parameter patterns.
- *dom*any(X): same as *dom*any(X,E) but the excluded value is not of a concern. This pattern is allowed to co-exist with *generated* and other single-parameter patterns.

Note that when a variable is instantiated, no *bound* or *dom* event will be posted. Therefore, the pattern *ins*(X) is mutually exclusive with *bound*(X), *dom*(X,E), and *dom*any(X,E). Nevertheless, the pattern *dom*any(X,E) may overlap with *bound*(X) and *dom*(X,E). Consider the following example:
\texttt{p(X),\{dom(X,E)\} \Rightarrow \text{write(dom}(E))}. \\
\texttt{q(X),\{dom\textunderscore any(X,E)\} \Rightarrow \text{write(dom\textunderscore any}(E))}. \\
\texttt{r(X),\{bound(X)\} \Rightarrow \text{write(bound)}.} \\
\texttt{go:-X :: 1..4, p(X), q(X), r(X), X \#\neq 2, X \#\neq 4, X \#\neq 1.}

The query \texttt{go} gives the following outputs: \texttt{dom(2)}, \texttt{dom\textunderscore any(2)}, \texttt{dom\textunderscore any(4)} and \texttt{bound}.\textsuperscript{4} The outputs \texttt{dom(2)} and \texttt{dom\textunderscore any(2)} are caused by \texttt{X \#\neq 2}, and the outputs \texttt{dom\textunderscore any(4)} and \texttt{bound} are caused by \texttt{X \#\neq 4}. After the constraint \texttt{X \#\neq 1} is posted, \texttt{X} is instantiated to 3, which posts an \texttt{ins}(X) event but not a \texttt{bound} or \texttt{dom} event.

Note also that the \texttt{dom\textunderscore any}(X,E) event pattern should be used only on small-sized domains. If used on large domains, constraint propagators could be overflooded with a huge number of \texttt{dom\textunderscore any} events. For instance, for the propagator \texttt{q(X)} defined in the previous example, the query

\texttt{X :: 1..1002, q(X), X \#>1000}

polls 1000 \texttt{dom\textunderscore any} events. For this reason, it is suggested that propagators for handling \texttt{dom\textunderscore any}(X,E) events are generated only after constraints are preprocessed and the domains of variables in them become small.

\section{The AC-4 Algorithm}

The AC-3 algorithm \cite{AC3} 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 \(X_i\) will be examined. The AC-4 algorithm \cite{AC4} 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 \texttt{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:

\texttt{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 \texttt{dom(2)} occurs before \texttt{dom\textunderscore any(2)} and also why \texttt{dom\textunderscore any(4)} occurs before \texttt{bound}.}
'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 \(\text{dom}(Y,Ey)\) 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 such as support constraints allowed in the solver competition [7], 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 \(\text{BinaryRelation}\) be a representation of the binary relation on two variables X and Y. An efficient data structure should be 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:

\[
\begin{align*}
\text{ac4}(\text{BinaryRelation}, X, Y), \\
\quad \text{var}(X), \text{var}(Y), \\
\quad \{\text{dom}_\text{any}(X, E)\} \\
=>
\quad \text{decrement_counters}(\text{BinaryRelation}, E, Y).
\end{align*}
\]

\text{ac4}(\text{BinaryRelation}, X, Y) \Rightarrow \text{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:

\[
\begin{align*}
'X in C-Y_ac'(X,Y,C), \text{var}(Y), \\
\quad \{\text{dom}(Y), \text{bound}(Y), \text{ins}(Y)\} \\
=>
\quad \text{remove_no_good}(X,Y,C).
\end{align*}
\]

'X in C-Y_ac'(X,Y,C) \Rightarrow X \text{ is } C-Y.
where \texttt{remove\_no\_good(X,Y,C)} removes all no-good values from the domain of \texttt{X} that are not supported by any values in the domain of \texttt{Y}. Because the propagator does not have the information about what values are excluded, it has to go through the domain elements of \texttt{X} to locate possible no-good values.

As for the constraint \texttt{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 [13]. This is true for binary functional constraints in general. For a general binary support constraint, the propagator \texttt{ac4(BinaryRelation,X,Y)} takes an extra argument \texttt{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 \texttt{X in C-Y\_ac'(X,Y,C)}, we use the propagator to maintain the arc consistency of the constraint \texttt{X \#= Y + 1}, where \texttt{X} \in \texttt{0..n}, \texttt{Y} \in \texttt{0..n} and \texttt{Y} \notin \texttt{2..n-2} for a given \texttt{n}. The following shows the program. In order to maintain arc consistency of the constraint, the domains must be represented as bit vectors when a hole occurs. The call \texttt{fd\_vector\_min\_max(0,N)} resets the range such that the domains will be represented as bit vectors for any given \texttt{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(\texttt{I,L,V})

The constraint \texttt{element(I,L,V)} means that the \texttt{I}th element of \texttt{L} is \texttt{V}, where \texttt{I} must be an integer or an integer domain variable, \texttt{V} a term, and \texttt{L} a list of
Table 1. Comparing the AC-3 and AC-4 implementations (CPU time).

<table>
<thead>
<tr>
<th>N</th>
<th>AC-3 (ms)</th>
<th>AC-4 (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>696.79</td>
<td>1.26</td>
</tr>
<tr>
<td>6000</td>
<td>1006.30</td>
<td>1.53</td>
</tr>
<tr>
<td>7000</td>
<td>1373.40</td>
<td>1.73</td>
</tr>
<tr>
<td>8000</td>
<td>1792.20</td>
<td>2.01</td>
</tr>
<tr>
<td>9000</td>
<td>2264.10</td>
<td>2.31</td>
</tr>
<tr>
<td>10000</td>
<td>2806.19</td>
<td>2.51</td>
</tr>
</tbody>
</table>

terms. The original version of the constraint presented in [1] requires \( V \) and the elements of \( L \) to be integers or integer 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 to a binary one. So the propagators given in this section are another application of the AC-4 algorithm.

Let \( L \) be a list of ground elements \([X_1, \ldots, X_n]\). Then \( I \) must be in the range of \( 1..n \). On one hand, each value in the domain of \( I \) must be supported by \( V \). As long as \( V \) is known not to be equal to some element \( X_i \), \( i \) can be excluded from the domain of \( I \). This relationship can be expressed by the following entailment constraint:

\[
V \not= X_i \implies I \not= i.
\]

On the other hand, each value in the domain of \( V \) must be supported by values in the domain of \( I \) as well. We use a counter \( C_{X_i} \) for \( X_i \) that tells the number of occurrences of \( X_i \) in \( L \). Each time a value \( i \) is excluded from the domain of \( I \), the counter \( C_{X_i} \) is decremented. If it becomes zero, then we can post the constraint \( V \not= X_i \).

The entailment constraints \( V \not= X_i \implies I \not= i \) \((1 \leq i \leq n)\) can be implemented using only one propagator thanks to the availability of the \texttt{dom\_any} event. To achieve this, we represent \( L \) as an association map such that for each \( X_i \) its indexes and counter can be returned in constant time. The following shows the propagator:

\[
\text{element\_V\_to\_I}(V,I,\text{Map}), \text{var}(V),
\{\text{dom\_any}(V,X)\}
\implies
\text{map\_get\_indexes}(\text{Map},X,\text{Indexes}),
I \not\in \text{Indexes}.
\text{element\_V\_to\_I}(V,I,\text{Map}) \implies \text{true}.
\]

Whenever a value \( X \) is excluded from \( V \)'s domain, the constraint \( I \not\in \text{Indexes} \) ensures that \( I \) cannot take the index of any occurrence of \( X \). When \( V \) is instantiated to be a non-variable term, the propagator vanishes.

The propagation from \( I \) to \( V \) can be done using only one propagator as well. Let \( \text{Vect} \) be a vector representation of \( L \) for which the element of a given index can be returned in constant time. The following defines the propagator:
element(I_to_V(I,V,Vect,Map),var(I),
{dom_any(I,Index)})
=>
arg(Index,Vect,X),
decrement_counter(Map,V,X).
element(I_to_V(I,V,Vect,Counters) => true.

The call decrement_counter(Map,V,X) decrements the counter of X and posts the constraint V \#\neq X if X’s counter becomes zero.

In addition to the two propagators element(V_to_I) and element(I_to_V), we need two extra propagators to handle ins(I) and ins(V) events. When I is instantiated to an integer i, the constraint V \#\neq X_i is generated, and when V is instantiated, the domain of I is reduced to contain only the indexes of the occurrences of V in L.

Computational results

We experimentally compared the performance of the two implementations of the element constraint. We could find only one benchmark program, called cars, which uses the element constraint for sequencing cars in assembly lines. To deal with the scarcity of programs, we implemented the alldifferent and permutation constraints 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(L) constraint is implemented using the element constraint as follows: Let L be a list of variables [V_1, ..., V_n] where each variable V_i has the domain 1..n, and let D be the list of integers [1, ..., n]. We use the following n element constraints to encode alldifferent(L):

\[
element(I_1,D,V_1) \\
... \\
element(I_n,D,V_n)
\]

where each pair I_i and I_j (i \neq j) take different values. Although alldifferent is encoded by using the relation itself on a new set of variables, the propagators for element can be tested because the original set, not the new set, of variables 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:

\[
\text{permutation}(L,P):- \\
\text{length}(L,N), \\
\text{length}(Is,N), \\
\text{permutation}(L,P,Is), \\
\text{alldifferent}(Is).
\]
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, and the column *bool* shows the time taken by the implementation that employs entailment constraints to propagate information from V to I. The propagator *element\_I\_to\_V* cannot be encoded using entailment constraints. For this reason, this propagator is encoded using the *dom\_any* event in both implementations.

<table>
<thead>
<tr>
<th>Program</th>
<th><em>dom_any</em> (ms)</th>
<th><em>bool</em> (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cars</td>
<td>4.70</td>
<td>7.19</td>
</tr>
<tr>
<td>sudoku (element)</td>
<td>101.50</td>
<td>160.90</td>
</tr>
<tr>
<td>sort</td>
<td>304.69</td>
<td>495.30</td>
</tr>
</tbody>
</table>

It is not surprising that *dom\_any* outperforms *bool* since the exclusion of each value from V activates only one propagator in *dom\_any* while it activates n (the size of L in *element*(I,L,V)) propagators in the implementation that uses entailment 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 [2, 12], 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 [2]. 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 *alldifferent*(L) constraint where L has n variables over the domain of 1..n. Let L be a list of variables [X_1,...,X_n]. The dual constraint has the form *alldifferent*(Y_1,...,Y_n) where each Y_i also has the domain 1..n. The variables X_i in the original constraint are called *primal* variables and the variables Y_i in the dual constraint are called *dual* variables. The primal and dual variables have the following relationship:

\[
\forall_{i,j}(X_i = j \leftrightarrow Y_j = i) \\
\forall_{i,j}(X_i \neq j \leftrightarrow Y_j \neq i)
\]
A lot of algorithms have been proposed for maintaining different levels of consistency for `alldifferent` [10]. The filtering algorithm by Regin [6] 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.\(^5\)

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 [15]. Understandably, since no union of domains is considered, this heuristic has its limitation. Consider, for example, the constraint `alldifferent([X_1,X_2,X_3,X_4])` where \(X_1 \in \{1,2\}, X_2 \in \{1,3\}, X_3 \in \{2,3\}, \text{ and } X_4 \in \{1,2,3,4\}\). The heuristic fails to find the Hall set \(\{1,2,3\}\) and thus fails to bind \(X_4\) to 4.

Using dual models can increase the pruning power. The dual variables in the dual constraint `alldifferent([Y_1,Y_2,Y_3,Y_4])` have the following domains: \(Y_1 \in \{1,2,4\}, Y_2 \in \{1,3,4\}, Y_3 \in \{2,3,4\}, \text{ and } Y_4 \in \{4\}\). 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.

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

\[
\text{primal_dual}(X_i,I,\text{DualVarVector}), \text{var}(X_i), \\
\{\text{dom_any}(X_i,J)\} \\
=> \\
\arg(J,\text{DualVarVector},Y_j), \\
Y_j \neq I.
\]

Each time a value \(J\) is excluded from the domain of \(X_i\), assume \(Y_j\) is the \(J\)th variable in \(\text{DualVarVector}\), then \(I\) must be excluded from the domain of \(Y_j\). 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.

\(^5\) For the constraint `alldifferent([X_1,...,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 \(D_1,...,D_n\) is greater than or equal to the size of \(H\). Formally, \(H\) is a Hall set if \(\{|D_i \mid D_i \subseteq H\}| \geq |H|\).
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.
- **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 [1], 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 equivalence Boolean constraints for relating primal and dual variables.

The speed-ups for channeling constraints are higher than those for the `element` constraint. 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).

<table>
<thead>
<tr>
<th>Program</th>
<th><code>dom any</code> (ms)</th>
<th><code>bool</code> (ms)</th>
<th><code>bool</code> <code>dom any</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>sudoku(alldifferent)</td>
<td>71.90</td>
<td>275.00</td>
<td>3.82</td>
</tr>
<tr>
<td>queens</td>
<td>51.60</td>
<td>468.80</td>
<td>9.08</td>
</tr>
<tr>
<td>hamilton</td>
<td>737.50</td>
<td>1487.5</td>
<td>2.01</td>
</tr>
</tbody>
</table>

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 [3], but represents the lower and upper bounds as two finite domain variables rather than as two sorted lists. In our representation, 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 $V^{\text{univ}}$ 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 $V$. $V^n$ 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 $1..|V^{\text{univ}}|$. To prevent $V^l$ and $V^n$ 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 $V^n$ 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 $V^n$, which results in a new upper domain $[0,1,2,4]$.

This representation was first adopted in the set solver implemented in B-Prolog version 6.2 and later adopted by the fd_sets solver in ECLiPSe.

The propagation rules for set constraints can be encoded in action rules quite straightforwardly. For example, consider the following two rules that maintain the bounds consistency of the subset constraint $R \subseteq S$:

$$\begin{align*}
  x \in R &\quad x \in S \\
  x \notin S &\quad x \notin R
\end{align*}$$

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:\footnote{In the real implementation, a set domain variable is represented as a suspension variable with an attached term. The built-in susp_attached_term($V,T$) is used to retrieve the attached term of a variable.}

```prolog
subset_from_R_to_S(R,S),
  dom(R^l,E)
  =>
  clpset_add(S,E).

subset_from_S_to_R(R,S),
  dom(S^n,E)
  =>
  clpset_exclude(R,E).
```

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

12
Computational results

Table 4 compares the performance of two set solvers: the \texttt{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. The \texttt{fd_sets} solver adopts the same domain representation as in B-Prolog. The propagation rules from the Conjunto solver [3] are implemented in both solvers.

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 [3]. 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.

The BP solver is significantly faster than the \texttt{fd_sets} solver for both programs although the same domain representation and propagation rules are used in both solvers. In \texttt{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 \texttt{dom} event.

The \texttt{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.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|}
\hline
Program & \texttt{fd_sets} (ms) & BP (ms) & \texttt{fd_sets}/BP \tabularnewline
\hline
Steiner(9) & 2,025 & 125 & 16.19 \tabularnewline
Golf & 2,000 & 609 & 3.28 \tabularnewline
\hline
\end{tabular}
\caption{Comparison of two set solvers (CPU time).}
\end{table}

7 Concluding Remarks

We have presented several application examples of the \texttt{dom} event. For some applications, such as the \texttt{AC-4} algorithm and the propagators for set constraints, the \texttt{dom} event facilitates propagating updates of domains. For some other applications, such as the constraint \texttt{element} and channeling constraints, the \texttt{dom} event makes it possible to describe a relationship with an-order-of-magnitude fewer propagators. Our experimental results confirm the importance of the \texttt{dom} event. B-Prolog's CLP(FD) system is considerably faster than its peers\textsuperscript{7} and the high performance is attributed to a large extent to the support of the \texttt{dom} event.

\textsuperscript{7} See \url{www.probp.com/benchmark_clpfd.htm} for detailed comparison results.
As we know no other language for implementing constraint propagators provide constructs like the `dom` event. Both ECLiPSe and Sicstus provide constructs for implementing the AC-3 algorithm. 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. But since demons are not informed of the particular values excluded, they cannot propagate the updates in constant time as in the AC-4 algorithm.

In Sicstus, 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, no value can be transmitted to the propagators and thus it is impossible to achieve constant-time updates.

`cc(FD)` [9] compiles a functional constraint into implication constraints to maintain its arc consistency. An optimization technique is used to combine implication constraints to achieve better space efficiency. As we know, `cc(FD)` provides no construct like the `dom` event to the user for implementing arc consistency algorithms.

There are two reasons for the reluctance of introducing `dom`-like construct into implementation languages for 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 [15]. 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 [11, 13]. 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 [11, 13]. For example, the AC-3.1 algorithm [13] does not perform better than original AC-3 algorithm for bi-directional functional constraints since each value has only one supporting value in the other domain and there is no need to remember resumption point for each value.

We believe that the `dom` event can be found useful in more applications. For example, in the incremental version of Regin’s filtering algorithm [6], 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 such as constraints in bio-sequence analysis and predication. As more applications are found, we expect that the `dom` event will be widely accepted as a mandatory feature for implementation languages of constraint propagators.
Acknowledgement

The work by Neng-Fa Zhou was done while he visited Monash University and The University of Melbourne in 2005.

References