Implementation of Propagation Rules for Set Constraints Revisited

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

Joachim Schimpf
IC-PARC, Imperial College, London
j.schimpf@icparc.ic.ac.uk

Abstract

This paper presents a set constraint solver that is the result from our attempt to improve the Conjunto solver. Our solver inherits the interval representation scheme for set domains from Conjunto, but represents the lower and upper bounds as two finite domain variables rather than as two sorted lists. Our solver is based on a set of propagation rules that propagate changes of the bounds of set variables. The advantage of our solver over Conjunto is that updates of bounds can be performed in constant time. Our solver is implemented in *action rules*, a high-level construct available in B-Prolog for programming active agents. Our solver is significantly faster than Conjunto and is comparable in performance with the Ilog and Oz solvers, two set solvers implemented in C++.

1 Introduction

Constraint Logic Programming (CLP) defines a family of programming languages that extend Prolog to support constraint solving over certain domains [4]. CLP(Set) is a member in the CLP family where each variable can have a set as its value. Set constraints over general sets are very hard to solve or even impossible to solve if certain set expressions are allowed [8]. For this reason, most systems handle finite sets only. Several set constraint solvers have been developed and incorporated into systems such as Eclipse [9], Ilog [10], and Mozart Oz [7]. These solvers employ *constraint propagation*, a constraint solving method originated in Artificial Intelligence [6] to maintain the consistency of constraints. These solvers allow for more natural modeling of certain combinatorial search problems. Some problems, such as the set partition problem, can be described in CLP(Set) with few variables than in Integer Programming or CLP(FD). Some other problems such as some scheduling problems that are hard to model in Integer Programming can be modeled naturally in CLP(Set).

The objective of this research is to improve the performance of the Conjunto solver [2]. In Conjunto, a set variable is represented as an interval of sets where the lower bound contains *definite* elements that must be in the set and the upper bound contains *possible* elements for the set. Interval bounds are set constants represented as sorted lists. Propagation rules are used to dynamically maintain the interval consistency of constraints. For example, for the constraint $R \subseteq S$, whenever the lower bound of R is updated the lower bound of S must be updated as well and whenever the upper bound of S is updated the upper bound of R must be updated as well to keep the constraint consistent. New bounds are computed from the existing ones. Since the bounds are represented as sorted lists, it takes linear time to update a bound

in the worst case. One question arises naturally: "is it possible to find an efficient data structure for the bounds that enable propagation rules to update them in constant time?".

This paper presents an affirmative answer to the question. The idea is to represent bounds as finite-domain variables from which an element can be excluded in constant time. To accommodate this representation, we reform the propagation rules such that new bounds are computed from changes rather than from the existing ones.

Our solver is implemented in action rules, a high-level language available in B-Prolog for programming active agents. The solver constantly outperforms the Conjunto solver. Although our solver is implemented in a high-level language, it offers comparable performance with the ones in Ilog solver and Mozart Oz that are implemented in C++.

This paper is organized as follows: Section 2 defines the CLP(Set) language we implemented; Section 3 presents the propagation rules for set constraints; Section 4 describes the implementation in B-Prolog's action rules; Section 5 compares the performance of the solver with Conjunto and three other solvers; and Section 6 compares our approach with related ones and discusses further directions of work.

2 CLP(Set)

CLP(Set) is a member in the CLP family where each variable can have a set as its value. Although a number of languages are named CLP(Set)[8], they are quite different. Some languages allow intentional and infinite sets, and some languages allows user-defined function symbols in set constraints. To avoid confusion, we first define our language.

We consider only finite sets of ground terms. A *set constant* is either the empty set `{}` or `{T1,T2,...,Tn}` where each `Ti` (i=1,2,...,n) is a ground term.

We reuse some of the operators in Prolog and CLP(FD) (e.g., `/\`, `\/`, `\`, `#=`, and `#\=`) and introduce several new operators to the language to denote set operations and set constraints. Since most of the operators are generic and their interpretation depends on the types of constraint expressions, the users have to provide necessary information for the system to infer the types of expressions.

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

```
V :: L..U
```

where `V` is a variable, and `L` and `U` are two set constants indicating respectively the lower and upper bounds of the domain. The lower bound contains all *definite* elements that are known to be in `V` and the upper bound contains all *possible* elements that may be in `V`. All definite elements must be possible. In other words, `L` must be a subset of `U`. If this is not the case, then the declaration fails. The special set constant `{I1..I2}` represents the set of integers in the range from `I1` to `I2`, inclusive. For example:

- `V :: {}..{a,b,c}` : V is subset of `{a,b,c}` including the empty set.
- `V :: {1}..{1..3}` : `V` is one of the sets of `{1}`,`{1,2}`, `{1,3}`, and `{1,2,3}`. The set `{2,3}` is not a candidate value for `V`.
- `V :: {1}..{2,3}` : Fails since `{1}` is not a subset of `{2,3}`.

We extend the notation such that `V` can be a list of variables. So the call

```
[X,Y,Z] :: {}..{1..3}
```

declares three set variables.

The following primitives are provided to test and access set variables:

- `clpset_var(V)`: `V` is a set variable.
- `clpset_low(V,Low)`: The current lower bound of `V` is `Low`.
- `clpset_up(V,Up)`: The current upper bound of `V` is `Up`.
- `clpset_added(V,E)`: `E` is a definite element, i.e., an element included in the lower bound.

- `clpset_excluded(V,E)`: `E` has been forbidden for `V`. In other words, `E` has been excluded from the upper bound of `V`.

A *set expression* is defined recursively as follows: (1) a constant set; (2) a set variable; (3) a composite expression in the form of `S1 \/ S2`, `S1 /\ S2`, `S1 \ S2`, or `\ S1`, where `S1` and `S2` are set expressions. The operators `\/` and `/\` represent union and intersection, respectively. The binary operator `\` represents difference and the unary operator `\` represents complement. The complement of a set `\ S1` is equivalent to `U \ S1` where `U` is the universal set. Since the universal set of a constant is unknown, `S1` in the expression `\ S1` must be a variable whose universal set has been declared.

We extend the syntax for finite-domain constraint expressions to allow the expression `#S` which denotes the cardinality of the set represented by the set expression `S`.

Let `S`, `S1` and `S2` be set expressions, and `E` be a term. A set constraint takes one of the following forms:

- `S1 #= S2`: `S1` and `S2` are two equivalent sets (`S1`=`S2`).
- `S1 #\= S2`: `S1` and `S2` are two different sets (`S1`≠`S2`).
- `S1 subset S2`: `S1` is a subset of `S2` (`S1`⊆`S2`). The proper subset relation `S1` ⊂ `S2` can be represented as `S1 subset S2` and `#S1 #< #S2` where `#<` represents the less-than constraint on integers.
- `S1 #<> S2`: `S1` and `S2` are disjoint (`S1`∩`S2`=∅).
- `E #<- S`: `E` is a member of `S` (`E`∈`S`).
- `E #<\- S`: `E` is a not member of `S` (`E`∉`S`).

Boolean constraint expressions are extended to allow set constraints. For example, the constraint

```
(E #<- S1) #=> (E #<- S2)
```

says that if `E` is a member of `S1` then `E` must also be a member of `S2`.

As will be described later, we use constraint propagation to maintain the consistency of constraints. Constraint propagation alone, however, is inadequate for finding solutions for many problems. We need to use the *divide-and-conquer* or *relaxation* method to find solutions to a system of constraints. The call

- `indomain(V)`

finds a value for `V` either by enumerating the values in `V`'s domain or by splitting the domain. Instantiating variables usually triggers related constraint propagators.

Our CLP(Set) language is basically the same as the one defined in Conjunto [2]. Our language is not as powerful as many others found in the literature in the sense that sets are finite, and no intentional set expressions or user-defined function symbols are allowed in set expressions.

3 Propagation Rules

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. One method is to use intervals to represent set domains [2, 10]. We adopt the same method. 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.

We reform the propagation rules presented in [2] for set constraints such that new bounds are computed from changes rather than from existing bounds. For each constraint, there is a group of *static* rules that are applied when the constraint is added to the constraint store and a group of *dynamic* rules that propagate changes. The static rules achieve *interval consistency* when constraints are generated and the dynamic rules maintain *interval consistency* for constraints.

Definition 1 *A binary constraint* `p(X1,X2)` *is interval consistent on* `X1` *if for whichever bound of* `X1` *there exists a value in the domain of* `X2` *such that*

the constraint is satisfied. A constraint is interval consistent if it is interval consistent on both of the variables. This definition can be easily extended to non-binary constraints.

For example, the constraint $X \subseteq Y$ where X is defined in the domain `{1}..{1,2,3}` and Y is defined in the domain `{}..{1,2}` is not interval consistent since the upper bound of X (`{1,2,3}`) does not have a supporting value in the domain of Y and the lower bound of Y (`{}`) does not have a supporting value in the domain of `X` either. To make the constraint interval consistent, we must exclude 3 from the upper bound of X and add 1 into the lower bound of Y.

In the following of this section, we give the propagation rules for all types of constraints provided in CLP(Set). Each rule takes the form $\frac{precedent}{consequence}$ where *precedent* is a constraint or an event which may be followed a sequence of conditions, and *consequence* is a sequence of constraints. For presentation purposes , we use in this section mathematical symbols instead of the operators in the CLP(Set) language.

3.1 Domain declaration

A variable V becomes a set variable after the declaration $V :: L..U$. The following rule updates the attributes of the variable.

$$\text{r1. } \frac{V :: L..U}{V^l{=}L,\ V^u{=}U,\ V^c{\geq}|L|,\ V^c{\leq}|U|}$$

After the declaration $V :: L..U$, the lower bound of V is initialized to L, the upper bound is initialized to U, and the cardinality is constrained to be within the range of $|L|$ and $|U|$.

The following dynamic rules maintain the consistency of the attributes of the variable V:

$$\text{r2. } \frac{x{\in}V}{V^c{\geq}||V^l|,\ V^c{=}{=}|V^l|{\rightarrow}V{=}V^l}$$

$$\text{r3. } \frac{x{\notin}V}{V^c{\leq}|V^u|,\ V^c{=}{=}|V^u|{\rightarrow}V{=}V^u}$$

$$\text{r4. } \frac{ins(V^c)}{V^c{=}{=}|V^l|{\rightarrow}V{=}V^l,\ V^c{=}{=}|V^u|{\rightarrow}V{=}V^u}$$

The first two rules ensure that the cardinality attribute be in the range whenever an element is added to the lower bound or an element is removed from the upper bound. The third rule is triggered when the cardinality is instantiated. All the three rules instantiate the variable V when the cardinality attribute becomes equal to the size of the lower or upper bound. Here the operator== tests whether or not the two operands are identical.

3.2 Unification of a set variable with a constant

Let V be a set variable and c be a set constant. When the constraint $V = c$ is posted, all the elements in the constant c are added into V and the cardinality of V is constrained to be the same as c.

$$\text{r5. } \frac{S{=}c}{S^c{=}|c|,\forall_{x\in c}x{\in}S}$$

We assume in the following that no set constants occur in constraints. Let C be a constraint and c be a set constant. The constraint is translated into one without the constant c as follows.

$$\text{r6. } \frac{C,\ \exists_c occurs(c,C)}{V::\{\}..c,\ C',\ V{=}c}$$

where V is a new variable, C' is C with the constant c being replaced by V. The rule is repeatedly applied to a constraint until the constraint does not contain any more constants. Notice that the order of the constraints in the consequence is important. Placing $V = c$ after C' guarantees that C' does not contain the constant c when it is posted.

3.3 Subset

When the subset constraint $R \subseteq S$ is added to the store, the static rule ensures the following: (1) the cardinality of R is not greater than that of S; (2) all definite elements of R must be definite elements of S; and (3) no impossible elements for S can be permitted in R.

$$\text{r7. } \frac{R{\subset}S}{R^c{\leq}S^c,\ \forall_{x\in R^l}(x{\in}S),\ \forall_{x\in R^{univ}}(x{\notin}S^u{\rightarrow}x{\notin}R)}$$

The following two rules dynamically maintain the consistency of the constraint:

$$\text{r8. } \frac{x{\in}R}{x{\in}S} \qquad \text{r9. } \frac{x{\notin}S}{x{\notin}R}$$

Whenever an element is added to the lower bound of R it is added to S as well, and whenever an element is removed from the upper bound of S it is also removed from R.

3.4 Equality

The equality constraint $R = S$ is equivalent to two subset constraints: $R \subseteq S$ and $S \subseteq R$.

3.5 Disequality

The disequality constraint $R \neq S$ is delayed until both R and S are instantiated. The constraint succeeds when R and S are two different set constants.

3.6 Intersection

For the intersection constraint $R \cap S = T$, the static rule ensures the following: (1) T is a subset of both R and S[1]; (2)the definite elements of both R and S must also be definite in T; (3) any elements that are definite in R (S) but impossible in T must be forbidden for S (R); and (4) the constraint on the cardinalities $T^c \geq R^c + S^c - |U|$ where U is the union of universal sets of R and S.

$$\text{r10. } \frac{R \cap S = T}{\begin{array}{c} T \subseteq R,\ T \subseteq S,\ \forall_{x \in R^l \cap S^l}(x \in T), \forall_{x \in R^l}(x \notin T^u \rightarrow x \notin S), \\ \forall_{x \in S^l}(x \notin T^u \rightarrow x \notin R), \\ T^c \geq R^c + S^c - |U| \end{array}}$$

The dynamic rules for the subset constraints $T \subseteq R$ and $T \subseteq S$ guarantee that whenever an element is added to T it will be added to both R and S, and that whenever an element is removed from R or S it will be removed from T. We still need to take care of the cases when elements are added to R or S, and when elements are removed from T.

Whenever an element is added to R, if it is definite in S, then it must be added also to T. Similarly, Whenever an element is added to S, if it definite in R, then it must be added also to T. The following two rules handle these two cases:

$$\text{r11. } \frac{x \in R}{x \in S^l \rightarrow x \in T} \qquad \text{r12. } \frac{x \in S}{x \in R^l \rightarrow x \in T}$$

[1] For the sake of simplicity, we use the subset constraint in the definition. In the real implementation, propagators are defined directly.

Whenever an element is removed from the upper bound of T, if the element is a definite element of either one of R or S, then the element must be forbidden for the other set variable.

$$\text{r13. } \frac{x \notin T}{x \in R^l \rightarrow x \notin S,\ x \in S^l \rightarrow x \notin R}$$

3.7 Union

The propagation rules for the union constraint are similar to those for the intersection constraint. For the union constraint $R \cup S = T$, the static rule ensures the following: (1) R and S are subsets of T; (2) any elements that are definite in T but impossible in R (S) must be definite in S (R); and (3) the constraint on the cardinalities $T^c \leq R^c + S^c$.

$$\text{r14. } \frac{R \cup S = T}{\begin{array}{c} R \subseteq T,\ S \subseteq T,\ \forall_{x \in T^l}(x \notin R^u \rightarrow x \in S,\ x \notin S^u \rightarrow x \in R) \\ T^c \leq R^c + S^c \end{array}}$$

The dynamic rules for the subset constraints $R \subseteq T$ and $S \subseteq T$ guarantee that whenever an element is added to R or S it will be added to T as well, and whenever an element is removed from T, it will also be removed from both R and S. We still need to take care of the cases when elements are removed from R or S, and when elements are added to T.

Whenever an element is removed from R, if it is known to be impossible in S, then it must be removed from T. Similarly, Whenever an element is removed from S, if it is impossible in R, then it must be removed also from T. The following two rules handle these two cases:

$$\text{r15. } \frac{x \notin R}{x \notin S^u \rightarrow x \notin T} \qquad \text{r16. } \frac{x \notin S}{x \notin R^u \rightarrow x \notin T}$$

Whenever an element is added into T, if the element is impossible in R or S, then the element must be added to the other set variable.

$$\text{r17. } \frac{x \in T}{x \notin R^u \rightarrow x \in S,\ x \notin S^u \rightarrow x \in R}$$

3.8 Complement

Let R and S be two set variables with the same universal set U, the complement constraint $R = \overline{S}$ is equivalent to:

$$R \cup S = U, R \cap S = \emptyset$$

While it is possible to define the complement constraint in terms of the intersection and union constraints, it is more efficient to use special propagation rules to maintain the complement relationship.

$$\text{r18. } \frac{x\in S}{x\notin R} \qquad \text{r19. } \frac{x\in R}{x\notin S}$$
$$\text{r19. } \frac{x\notin S}{x\in R} \qquad \text{r21. } \frac{x\notin R}{x\in S}$$

Whenever an element is added into a set it must be removed from the opposite set, and whenever an element is removed from a set it must be added to its opposite set.

3.9 Disjoint

Two sets R and S are disjoint iff $R\cap S=\emptyset$. Instead of using intersection, we can use special propagation rules to maintain the relationship. When an element is added to one set, it must be forbidden for the other set.

$$\text{r22. } \frac{x\in R}{x\notin S} \qquad \text{r23. } \frac{x\in S}{x\notin R}$$

3.10 Difference

The difference of two set variables $R-S$ is equivalent to $R\cap\overline{S}$ in theory. Since the universal set of S may not be known, it is impossible to define difference in terms of complement in practice. For this reason, we develop a set of propagation rules specially for this constraint.

When the constraint $R-S=T$ is generated, the static rule achieves the following: (1) T is a subset of R; (2) S and T are disjoint; (3) all elements that are definite in R but impossible in S must be definite in T; and (4) the constraint on the cardinalities $T^c\geq R^c-S^c$.

$$\text{r24. } \frac{R-S=T}{\begin{array}{c}T\subseteq R,\ T\cap S=\emptyset,\ \forall_{x\in R^l}(x\notin S^u\rightarrow x\in T)\\ T^c\geq R^c-S^c\end{array}}$$

The constraints $T\subseteq R$ and $T\cap S=\emptyset$ entail the following: (1) whenever an element is added into T it is added into R and removed from S; (2) whenever an element is removed from R it is also removed from T; and (3) whenever an element is added to S it is removed from T. We need rules to take care of the cases when an element is added to R and when an element is removed from S or T. When an element is added to R, if it is known to be impossible in S then it must be added to T. When an element is removed from S, if it is known to be definite in R then it must be added to T. When an element is removed from T, if it is definite in R then it must be added to S. The following three rules take care of these three cases:

$$\text{r25. } \frac{x\in R}{x\notin S^u\rightarrow x\in T}$$
$$\text{r26. } \frac{x\notin S}{x\in R^l\rightarrow x\in T}$$
$$\text{r27. } \frac{x\notin T}{x\in R^l\rightarrow x\in S}$$

4 Implementation

In this section, we describe the implementation in B-Prolog of a set constraint solver based on the propagation rules given in the previous section. B-Prolog provides a new language, called *action rules*, which is useful for programming active agents. An agent is like a sub-goal of Prolog but behaves in an event-driven manner. As long as constraint programming is concerned, a constraint propagator is an agent that dynamically maintains the consistency of the constraint. Before we describe how to implement the set constraint solver in action rules, we briefly introduce the syntax and semantics of action rules. The reader is referred to [13] for the details.

4.1 Action rules and suspension variables

An *action rule* takes the following form:

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

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

All conditions in `Condition` must be in-line tests. The event set `Event` together with the enclosing braces is optional. If an action rule does not have any event patterns specified, then the rule is called a *commitment rule*. A set of built-in events is provided for programming constraint propagators

and interactive graphical user interfaces. For example, `ins(X)` is an event that is posted when the variable `X` is instantiated and `dom(X,E)` is posted when an inner element `E` is excluded from the domain of the finite-domain variable `X`. A user program can create and post its own events and define agents to handle them. A user-defined event takes the form of `event(X,T)` where `X` is a variable, called a *suspension variable*, that connects the event with its handling agents, and `T` is a Prolog term that contains the information to be transmitted to the agents. If the event poster does not have any information to be transmitted to the agents, then the second argument `T` can be omitted. The built-in action `post(E)` posts the event `E`.

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

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

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

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

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

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

Suspension variables

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

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

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

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

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

Suspension variables are similar to attribute variables [3], but do not rely on goal expansion to define the behaviors associated with them. Whenever a suspension variable `X` is bound to another term, which may be another variable, the event `ins(X)` will be posted. The user can specify the action to be taken after a suspension variable is bound, but not the action to be taken before unification takes place.

The following example illustrates the use of suspension variables:

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

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

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

4.2 Representation of set variables

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

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

where `Low` and `Up` are two finite domain variables that represent respectively the lower and upper bounds, `Card` is another finite domain variable that represents the cardinality, and `Univ` is a term that represents the universal set. For a declared set variable, the universal set is the upper bound specified in its declaration. Not all set variables need to be declared. For a constraint such as `R subset S`, the universal set of `R` is assumed to be that of `S` if `R` is not declared as a set variable. The `Univ` field is redundant since the universal set can be always computed from `Up`. It is stored explicitly to facilitate the computation.

The idea of using two finite domain variables to represent the bounds is novel. `Low`'s domain is the complement of the set of elements that are known to be in `V`, and `Up`'s domain contains those elements that are possible to be included in `V`. To add an element into `V`, we exclude it from the domain of `Low`, and to remove an element from `V`, we exclude it from the domain of `Up`. In order to prevent the finite domain variables `Low` and `Up` from being instantiated when their domains become singletons, we add two dummy elements `$d1` and `$d2` into the domains. For example, after the call `V :: {}..{a,b,c}`, both `Low` and `Up` have the domain `[$d1,a,b,c,$d2]`. If `a` is known to be a member of `V`, then `Low`'s domain becomes `[$d1,b,c,$d2]`. If `c` is known not to be a member of `V`, then `Up`'s domain becomes `[$d1,a,b,$d2]`.

4.3 Implementation of propagation rules

Let `V` be a set variable with the attached term `set(Low,Up,Card,Univ)`. The primitive `clpset_add(V,E)` ensures that `E` is included in `V` and `clpset_exclude(V,E)` ensures that `E` is not included in `V`. The following gives the definitions. Notice again that `Low`'s domain is the *complement* of the set of definite elements plus two dummy elements.

```
clpset_add(V,E),
     V^attached=set(Low,Up,Card,Univ)
     =>
     fd_true(Up,E),
     fd_exclude(Low,E).

clpset_exclude(V,E),
     V^attached=set(Low,Up,Card,Univ),
     fd_true(Up,E)
     =>
     fd_true(Low,E),
     fd_exclude(Up,E).
clpset_exclude(V,E) => true.
```

where `fd_true(X,E)` succeeds iff `E` is in the domain of the finite-domain variable `X` and `fd_exclude(X,E)` excludes `E` from the domain of `X`.

The primitive `clpset_add(V,E)` fails if `E` is not possible for the set `V`, i.e., if `fd_true(Up,E)` fails. Notice that the event `dom(Low,E)` will be posted when `E` is excluded from `Low`'s domain.

The primitive `clpset_exclude(V,E)` does nothing if `E` has already been forbidden for the set `V`, i.e., if `fd_true(Up,E)` fails. Otherwise, `fd_exclude(Up,E)` excludes `E` from the domain of `Up` if `E` hasn't been added to the set `V`. Notice that the event `dom(Up,E)` will be posted when `E` is excluded from `Up`'s domain.

The propagation rules presented in the previous section can be encoded in action rules quite straightforwardly. For example, consider the fol-

lowing two rules that maintain the consistency of the subset constraint $R \subseteq S$:

$$\text{r8. } \frac{x \in R}{x \in S} \qquad \text{r9. } \frac{x \notin S}{x \notin R}$$

These two rules can be encoded as follows:

```
propagate_subset_low(R,S),
     R^attached = set(RLow,RUp,RCard,RUniv),
     {dom(RLow,E)} =>
     clpset_add(S,E).

propagate_subset_up(R,S),
     S^attached = set(SLow,SUp,SCard,SUniv),
     {dom(SUp,E)} =>
     clpset_exclude(R,E).
```

The first rule says that whenever an element is added to `R`, add it also to `S` and the second rule says that whenever an element is removed from `S`, remove it also from `R`.

Consider another propagation rule:

$$\text{r2. } \frac{x \in V}{V^c \geq |V^l|,\ V^c == |V^l| \rightarrow V = V^l}$$

This rule is encoded as follows:

```
when_low_bound_updated(V),
     V^attached = set(Low,Up,Card,Univ),
     {dom(Low,_)}
     =>
     fd_size(Low,LowSize),
     univ_size(Univ,UnivSize),
     NewCardLow is UnivSize-LowSize+2,
     Card #>= NewCardLow,
     ((integer(Card),Card=:=NewCardLow) ->
          clpset_low(V,V);
     true).
```

Whenever an element is excluded from the lower bound, this rule ensures that the cardinality is no less than the number of definite elements. The predicate `fd_size` gets the size of a given finite-domain variable and the predicate `univ_size` returns the size of the universal set. As the lower bound is a finite-domain that contains the complement of the set of definite elements plus two dummy elements, the size of the lower bound is computed as `UnivSize-LowSize+2`. When the cardinality is equal to the size of the lower bound, the call `clpset_low(V,V)` instantiates `V` to be the lower bound of `V`.

The primitive `indomain(X)` is implemented as follows:

```
indomain(X):-
     clpset_var(X),!,
     clpset_up(X,Up),
     set_to_list(Up,UpList),
     indomain(X,UpList).
indomain(X).

indomain(X,_):-nonvar(X),!.
indomain(X,[]).
indomain(X,[E|Es]):-
     clpset_added(X,E),!,
     indomain(X,Es).
indomain(X,[E|Es]):-
     (clpset_add(X,E);
      clpset_exclude(X,E)),
     indomain(X,Es).
```

For each element in the upper bound that is not definite, it creates a fork of two branches, one leading the inclusion of the element in the set and the other leading to the rejection of the element. With elements being added into or excluded from the set, one of the propagation rules of r2, r3, and r4 will be activated. If the cardinality is equal to the size of the lower bound or upper bound, the set variable will be instantiated. Once `X` is instantiated, `indomain(X)` will stop creating any further forks.

5 Performance Evaluation

Our solver has been implemented and incorporated into B-Prolog [15]. In this section, we compare the performance of our solver with that of the Conjunto solver in Eclipse 5.3. Comparing two libraries which run on different Prolog/CLP engines is not very informative, since it is difficult to tell whether a particular performance difference is caused by the underlying engine or by the library's algorithms and implementation technique.

To obtain a somewhat better indication, we also measure the performance of the `fd_sets` solver, which is implemented in Eclipse, and uses ideas similar to the ones used in our B-Prolog solver.

We also compare indirectly our solver with two other solvers, the Ilog solver, and the solver in Oz.

We first conduct unit testing [12], comparing the speed of the basic operations in the two solvers, and then conduct benchmarking, comparing the speed of the solvers on several benchmarks. All the benchmarks used in the comparison are available from *www.probp.com/bench/clpset.tar.gz*.

5.1 The fd_sets solver

The fd_sets solver that we use for comparison purposes is completely implemented in Eclipse and uses the same primitives as the Conjunto solver (attributed variables and the suspension mechanism). Neither Conjunto nor fd_sets use any components implemented in a lower-level language, in particular, they don't rely on the finite domain solver other than for implementing the cardinality constraint. Fd_sets uses a propagation system based on the same dynamic rules that we have presented in section 3, but without the static rules. The domain representation does not use finite domain variables, but a standard Prolog term with one argument per set element. Comparing the results for Conjunto and fd_sets is therefore useful to appraise the impact of the new propagation rules as such. The comparison between fd_sets and B-Prolog can give an indication of the effects of domain representation and underlying Prolog machine properties.

5.2 Unit testing

The performance of a constraint solver depends on a set of basic operations. As long as set constraint solving is concerned, the set of basic operations includes *variable creation*, *constraint installation*, *constraint propagation*, *unification*, and *labeling*. The following defines these operations and shows the programs we used in the test.

- Variable creation: The call `X :: L..U` turns a Prolog variable `X` into a set variable. When a set variable is created, its attributes are initialized. The following program is used to test the operation:

```
create:-
    S1 :: {}..{1..10},
    S2 :: {}..{1..100}.
```

- Constraint installation: When a constraint is generated, the constraint is preprocessed to achieve interval consistency and the propagators are initialized. This operation is performed only once for each constraint. The following program is used to test the operation:

```
install:-
    [S1,S2,S3] :: {}..{1..10},
    S1 /\ S2 #= S3.
```

- Constraint propagation: Constraint propagation takes place when an element is added to the lower bound or an element is excluded from the upper bound of a variable. The following programs are used to test the operation. We use two different domain sizes because we expect the performance to be dependent on domain size:

```
propagate :-
    [S1,S2] :: {}..{1..10},
    S1 #<> S2,
    1 #<- S1, 2 #<- S1, 3 #<- S1,
    4 #<- S1, 5 #<- S1, 6 #<- S1,
    7 #<- S1, 8 #<- S1, 9 #<- S1.

propagate_big :-
    [S1,S2] :: {}..{1..100},
    S1 #<> S2,
    10 #<- S1, 20 #<- S1, 30 #<- S1,
    40 #<- S1, 50 #<- S1, 60 #<- S1,
    70 #<- S1, 80 #<- S1, 90 #<- S1.
```

- Unification: The following program is used to test the unification of two variables and the unification of a variable with a set constant.

```
unify:-
    [S1,S2,S3,S4] :: {}..{1..10},
    S1 = S2,
    S3 = S4,
    S1 = {1,2,2,4,5,6,7,8,9,10},
    S3= {}.
```

- Membership: The following program is used:

Table 1: Comparison on basic operations (CPU time (ms), Pentium-III/933, Linux).

Operation	Conjunto	fd_sets	BP
create	44	224	131
install	82	201	38
propagate	143	60	7.4
propagate_big	170	60	7.4
unification	23	342	21
add	83	46	24
remove	53	47	25

```
add :-
    create(Set),
    1 #<- Set, 9 #<- Set, 2 #<- Set,
    8 #<- Set, 3 #<- Set, 7 #<- Set,
    4 #<- Set, 6 #<- Set, 5 #<- Set.
remove :-
    create(Set),
    1 #<\- Set, 9 #<\- Set, 2 #<\- Set,
    8 #<\- Set, 3 #<\- Set, 7 #<\- Set,
    4 #<\- Set, 6 #<\- Set, 5 #<\- Set.
```

Table 1 shows the CPU times taken by each operation in both solvers on a Linux machine. Each test program is run 10000 times and the mean is taken. For each test program, the time spent in operations that are irrelevant to the tested operation is deducted from the execution time. So for example, the propagation cost does not include the domain creation and constraint installation costs.

The results show that domain creation has become much more expensive in both fd_sets and B-Prolog. This is expected because the domain representation is more complex: the ground representation of the bound sets has to be converted into a bitmap-style representation.

Unification and instantiation requires the opposite conversion and in addition the generation of inclusion or exclusion events for every previously uncertain set element. The fd_sets solver shows the expected slowdown, but B-Prolog does not. This requires further investigation.

Membership and non-membership however take advantage from the constant time set-bound updates and are faster than in Conjunto for both fd_sets and B-Prolog. Speedup factors vary with the position of the included/excluded element within the domain, but will generally increase with increasing domain size.

As expected, propagation is faster (a factor of 2-3 for fd_sets) or even much faster (about 20 for B-Prolog). Moreover, for both fd_sets and B-Prolog, the cost of propagation is independent of the domain size, while in Conjunto it increases with domain size.

Constraint setup becomes somewhat more expensive, although for B-Prolog this is compensated by the greater overall performance. Fd_sets may also suffer in this respect from not using separate static rules during constraint setup, but generating dynamic updates instead. This choice may have to be revisited.

5.3 Benchmarking

We compared the two solvers using the following benchmarks:

- `Clique` Find the largest clique in a graph[2]. A clique is a complete sub-graph in which every two vertexes are connected. This program employs the generate-and-test algorithm. Let V be the set of vertexes in a given graph. For each cardinality from $|V|$ down to 1, the program iterates until it finds a subset of V that comprises a clique.

- `Steiner` The ternary Steiner problem of order n is to find $n(n-1)/6$ sets over $\{1, 2, ..., n\}$ such that each set contains three elements and any two sets have at most one element in common. This program was taken from [2]. No constraints 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.

Table 2 shows the CPU times taken by the two solvers to run the programs. BP is faster than

[2]Taken from *www.cs.sunysb.edu/~ algorithm/.*

Table 2: Comparison on CPU times (Linux).

Program	Conjunto (ms)	BP (ms)
Clique	26,780	5,900
Steiner(9)	25,320	880
Golf	52,260	3,620

Table 3: Number of backtracks.

Program	Conjunto	BP
Clique	379,730	379,730
Steiner(9)	6,924	6,924
Golf	1,808	1,808

Conjunto for all the benchmarks. The speed-up for `Golf` is over 10 and that for Steiner is over 20.

In comparing CLP systems, it is inadequate to just measure CPU times [12]. The data are meaningless if the compared systems use different heuristics to solve the problem. We measured the number of backtracks that are made in the execution (see Table 3) and confirmed that the same strategy is used in both solvers to instantiate variables.

The results are very encouraging. The `Steiner` program, which spends over 99 percent of its time in enumerating and propagating values, exceeds the speedup shown by the unit propagation benchmark, even though the domain sizes are very small (3). This could indicate that there is a factor involved which we have not addressed in this paper, probably related to differences in the propagation order between Eclipse and B-Prolog. It could also be caused by the performace of the cardinality constraint. which we have omitted from the unit tests.

5.4 Comparison with other solvers

Table 4 compares the speed of four solvers with that of Conjunto. The `fd_sets` is another solver that comes with the Eclipse system. The data for Oz and Ilog are taken from Tobias Muller's thesis [7]. We agree that it is a dangerous practice to compare systems indirectly [12]. In [7], the `fd_sets` package in Eclipse 5.2 is compared. The results would favor Oz and Ilog if Eclipse 5.3 has any speed-up over 5.2 and would favor BP if Eclipse 5.3 has any slow-down over 5.2. The figures only give a rough picture on the performance of the compared systems.

Table 4: Speed-ups of four solvers over Conjunto (Linux).

Program	fd_sets	Oz*	Ilog*	BP
Steiner	1.87	7.41*	35.04*	28.77
Golf	4.56	16.34*	37.41*	14.44

*Adjusted data from [7]

Our solver is not as fast as that of the Ilog solver. The solvers of Oz and Ilog are implemented in C++, while our solver is implemented in action rules which are compiled into byte code and interpreted by an emulator. It is expected that our solver can reach or even exceed the speed of Ilog if some of the hot predicates are translated into C or byte code is further compiled into native code.

6 Discussion

This paper presents a set constraint solver that is the result from our attempt to improve the Counjunto solver. Our solver inherits the interval representation scheme for set domains from Conjunto, but represents the lower and upper bounds as two finite domain variables rather than as two sorted lists. This scheme was first used in [14] to illustrate the implementation of propagators in action rules. Our solver is based on the same set of propagation rules used in Conjunto, but the propagation rules are reformed such that new bounds of affected variables are computed from changes rather from existing bounds. The advantage of our solver over Conjunto is that updates of bounds can be done in constant time. Our solver is implemented in a high-level language called action rules. It is significantly faster than Conjunto and is comparable in performance with the solvers in Oz and Ilog solver, which are implemented in C++.

Our technique for achieving constant-time up-

dates of bounds is similar to the one used in the AC-5 algorithm [5] that maintains arc consistency of binary finite-domain constraints. The key idea is to propagate not just the information that some domain has been updated but the individual element that has been excluded. According to [2], the Ilog solver also implements an AC-5 like algorithm. To our knowledge, the domain representation scheme and the propagation rules used in the Ilog solver are not available in the public domain.

We have implemented in Eclipse [9] a set solver, called `fd_sets`, based on the set of propagation rules presented in this paper. In `fd_sets`, set domain variables are represented as attribute variables and propagation rules are encoded in demons. The `fd_sets` solver is not as fast as the one in B-Prolog because of the sohisticated priority-based scheduling strategy used in Eclipse. In addition, the efficiency of the primitives on finite-domain variables also makes a difference.

Azevedo and Barahona implemented in Eclipse another set solver [1]. In the solver, the lower bound of a set domain is represented as a sorted list, and the upper bound is represented as the difference between the set of possible elements and the set of definite elements. One novel feature of the solver is that it performs cardinal reasoning, i.e., reasoning about cardinalities of set variables. Cardinal reasoning dramatically reduces the search space for some problems such as circuit diagnostic problems. The set of cardinal reasoning rules can be easily integrated into our solver.

The set solver in Mozart Oz [7] is based on the propagation rules of Conjunto and the cardinal reasoning rules by Azevedo and Barahona. The propagation rules are implemented in CPI (Constraint Propagator Interface) and the filters that reduce the domains of variables are encoded in C++. It is known that cardinal reasoning has no positive effect on the two benchmarks `Steiner` and `Golf`. The speed-up of the Oz solver over Conjunto may come from the low level implementation language.

Thornary and Gensel [11] proposed a hybrid representation scheme for set domains in which large domains are represented as intervals and small domains are represented vectors of candidate sets. This scheme makes it possible to achieve higher-level consistency such as arc consistency of set constraints if only small domains are involved. This scheme would be very effective for the circuit diagnostic problem [1] since all the domains in the problem are very small.

Set constraint solvers are far less mature than solvers over other domains such as finite-domains, reals, and rationals, and new implementation techniques need to be further explored. Set constraint solvers facilitate modeling of many kinds of combinatorial problems. Unfortunately only a few set constraint programs are available now. We expect that more application programs will be developed once fast solvers become available. These application programs will in return lead to new implementation and optimization techniques for set constraints.

Acknowledgement

We would like to thank Carmen Gervet for useful comments on an early version of this paper and Francisco Azevedo for explaining to us his cardinal set solver.

References

[1] F. Azevedo and P. Barahona: Modelling Digital Circuits Problems with Set Constraints, *Proc. of the First International Conference on Computational Logic*, pp.414-428, 2000.

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

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

[4] J. Jaffar and J.-L. Lassez: Constraint Logic Programming, *Proc. Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of programming languages*, pp.111-119, 1987.

[5] P. van Hentenryck, Y. Deville and C.M. Teng: A Generic Arc-consistency Algorithm and its

Specializations, *Artificial Intelligence*, Vol. 57, pp.291-321, 1992.

[6] A.K. Mackworth and E.C. Freuder: The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems, *Artificial Intelligence*, 1985.

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

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

[9] J. Schimpf, et al.: *Eclipse User Manual, version 5.3*, *www.icparc.ic.ac.uk/eclipse/*, ICPARC, 2001.

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

[11] V. Thornary and J. Gensel: An Hybrid Representation for Set Constraint Satisfaction Problems, *http://www.inrialpes.fr*.

[12] M. Wallace, J. Schimpf, K. Shen and W. Harvey: On Benchmarking Constraint Logic Programming Platforms, to appear *Constraints, An International Journal*.

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

[14] N.F. Zhou: Programming Constraint Propagation in Action Rules, in *Proc. of Workshop on Constraint Handling Rules*, CL'2000.

[15] N.F. Zhou: *B-Prolog User's Manual*, *www.probp.com*, 2002.