

A Simple Constraint Solver in Action Rules for the CP'05 Solver Competition

Neng-Fa Zhou¹ and Mark Wallace²

¹ The City University of New York & Monash University

² Monash University

1 Introduction

In response to the call for solvers by the CPAI'05 solver competition organizing committee, we implemented a constraint solver, called *Mosar*, using action rules (AR) in B-Prolog [3] particularly for this competition. The problem format used in the competition is very different from those used by practical solvers. For example, constraints are extensional numeric relations not implicit relations, and only support and conflict constraints are allowed. Our goal was to build a working solver very quickly (in one or two days) rather than to implement a winning solver. Therefore, no optimization is implemented at all.

Mosar is a very simple solver. It performs forward checking for unary constraints, maintains a weak-form of interval consistency for non-binary support constraints, and maintains arc-consistency for binary support constraints. Thanks to the high descriptive power of action rules, the solver, which consists of only about 200 lines of AR and Prolog code, was completed in only one day. As there is no solver of this type available for comparison, the performance of *Mosar* is unclear. *Mosar* has successfully solved most of the sample benchmarks including the quasi-group, Domino, and all-interval series problems collected by the organizing committee.

This note describes the solver and reports on the performance. It also gives several optimizations that could lead to better performance. The source code of the solver is available at: www.probp.com/mosar.

2 Action Rules

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 [3]. 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. The reader is referred to [3] for a detailed description of the language.

AR has been successfully used to implement several efficient constraint solvers in B-Prolog, including the ones over finite domains, Boolean, trees, and finite sets. So it is not surprising that AR can be used to easily implement propagators for support and conflict constraints. As an example, a naive propagator for conflict constraints can be implemented as follows:

```

conflict_constraint(Rel,Constr),
    no_vars_gt(1,0),{ins(Constr)}
=>
    true.
conflict_constraint(Rel,Constr)
=>
    not member(Constr,Rel).

```

where `Rel` is the list of tuples in the relation and `Constr` is the list of variables in a constraint. When the constraint contains at least one variable (i.e., when the condition `no_vars_gt(1,0)` succeeds)³, the propagator is suspended waiting for a variable in `Constr` to be instantiated; and when the constraint becomes ground, the test `not member(Constr,Rel)` is performed to ensure that the valuation of the variables in `Constr` is not a tuple in the relation.

3 The Mosar Solver

This section describes the propagators for conflict and support constraints. Mosar adopts the *first-fail principle*⁴ as the labeling strategy for ordering variables. As mentioned in the section on Future Improvements, a sophisticated labeling strategy is normally needed to solve hard problems.

3.1 Conflict constraints

A conflict relation is represented as a hashtable. Let $R = [T_1, \dots, T_n]$ be a conflict relation where T_1, \dots, T_n are the tuples, and let R_i be the projection of R onto the columns except for column i . For each column i and for each tuple T in R_i , there is an element in the hashtable with the key $k(i, T)$ and the value $[A_1, \dots, A_k]$ which is a list of no-good values for column i if T is a partial solution.

The propagator for a conflict constraint is implemented as follows:

³ In general, the condition `no_vars_gt(m,n)` succeeds if the number of variables in the last `m` arguments of the agent is greater than `n`.

⁴ Selecting a variable with the minimum domain size, breaking the tie by selecting a variable involved in the largest number of constraints.

```

conflict_constraint(Rel,Constr),
    no_vars_gt(1,1),{ins(Constr)}
=>
    true.
conflict_constraint(Rel,Constr)
=>
    project_constr_on_one_arg(Constr,T,I,Xi),
    conflict_constraint_action(Rel,k(I,T),Xi).

```

where `Rel` is the conflict relation represented as a hashtable and `Constr` is the list of variables of a constraint. The propagator is suspended if there are more than one free variables in the constraint. The second rule is executed if the constraint contains at most one variable. The call `project_constr_on_one_arg(Constr,T,I,Xi)` extracts the free variable `Xi`, its column number `I`, and the list of ground arguments `T` from `Constr`, and the call `conflict_constraint_action(Rel,k(I,T),Xi)` retrieves the no-good values for `Xi` from `Rel` and excludes them from the domain of `Xi`.

3.2 Support constraints

Given a support relation, we extract the following information from it: (1) *static bounds information*: the minimum and maximum elements in each column; (2) *dynamic bounds information*: for each value x in each column i , the minimum and maximum support elements in each column j ($j \neq i$); and (3) *projected binary relations*: the projected binary relations of the original relation onto each two columns. Each projected binary relation is represented as a hashtable, so that for each value in a column we can retrieve its support values in the other column in the binary relation.

For a support constraint on variables $[V_1, \dots, V_m]$, the collected information from its relation is used in the following way:

- Static bounds information is used to preprocess the support constraint. Let V_i be the variable in column i , and L_i and U_i be the minimum and maximum elements in the column, the domain constraint $V_i :: L_i..U_i$ is posted in the preprocessing phase.
- Dynamic bounds information is used to maintain a weak-form of interval consistency. Whenever the variable V_i in column i is bound to a value x , let L_j and U_j be the minimum and maximum of the elements in column j that support x , the domain constraint $V_j :: L_j..U_j$ is enforced.
- The projected binary relations are used to maintain arc consistency. Whenever a support constraint becomes binary, say on variables V_i and V_j , we set up a counter for each value in each column for counting the support elements in the other column. Whenever the counter of an element becomes zero, the element is excluded from its domain.

Let `BinaryRelation` be a projected binary relation on \widehat{V}_i and V_j . The propagator that maintains the counters is implemented as follows:

```

arc_propagator(Counters,BinaryRelation,I,Vi,J,Vj),
  var(Vi),var(Vj),
  {dom(Vi,Ei)}
=>
  get_support_elements(BinaryRelation,k(I,Ei),SupportEi),
  decrement_counters(Counters,J,SupportEi,Vj).
arc_propagator(Counters,BinaryRelation,I,Vi,J,Vj) => true.

```

Whenever an element E_i is excluded from the domain of V_i , the counters of the elements in the domain of V_j supported by E_i is decremented. If the counter of an element becomes zero, the element is excluded from the domain of V_j .

In B-Prolog, the constraint $X \neq E$, where X is a domain variable and E is a constant, does not post the $dom(X, E)$ event if E happens to be a bound of the domain of X . For implicitly represented relations, this is not a problem because there are separate propagators for handling exclusions of inner elements and updates of bounds. In Mosar, however, in order to maintain arc consistency, the solver must post a $dom(X, E)$ event even if E is a bound. For this reason, we change the implementation of $X \neq E$ into the following:

```

v_neq_c(X,E):-
  fd_min_max(V,Min,Max),
  exclude_value(X,E),
  (nonvar(X)->true;
  E==Min->post(dom(X,E));
  E==Max->post(dom(X,E));
  true).

```

4 Performance

We have tested Mosar using part of the benchmarks made available by the organizing committee. Mosar has successfully solved many problems we used. For example, it solved all the problem instances of the quasi-group, Domino, and all-interval series benchmarks. Table 1 shows the times taken to solve the hardest problem instances in these benchmark groups on a 2GHz Linux machine. We have also confirmed that Mosar failed to solve many other benchmark problems within a reasonable time limit, such as the Latin Square problem ($N \geq 8$), the Pigeons problem ($N \geq 14$), and the Job-shop scheduling problem. Most of the hard problems are unsatisfiable problems and some of the problems such as the Job-shop scheduling problem are known to require sophisticated labeling strategies.

5 Future Improvements

The following improvements could lead to significant speed-ups of Mosar:

Table 1. Benchmarking results.

Problem	Instance	Time
All interval series	series50	0.72s
Domino	domino-500-300	56.39s
Quasi-group	bqwh-15-106-99	0.48s

1. *More efficient data structures.* Currently, hashtables are used to represent relations. Multi-dimensional arrays should be better than hashtables if the domains of variables are not sparse.
2. *Handling specific constraints.* For specific constraints such as functional, anti-functional, and monotonic relations [2], more efficient representations and propagators can be used. For relations that can be represented implicitly as equality, inequality, and disequality ones, existing propagators in B-Prolog should be used.
3. *Other consistency algorithms.* Currently, the propagator implemented in Mosar for support constraints is based on the AC-4 algorithm. Along the line of using optimized propagators for specific constraints, the algorithm should be optimized to exploit special characteristics of constraints, such as bidirectionality [1], to archive better performance. Also, for certain relations path consistency should be considered to reduce the search space.
4. *Labeling strategies.* No labeling strategy is known to work for every problem. A labeling strategy should make use of the probabilistic features of constraint networks. In addition, labeling should be performed in several stages if a given problem can be decomposed into independent subproblems.

References

1. BESSI, C., FREUDER, E. C., AND REGIN, J.-C. 1999. Using constraint metaknowledge to reduce arc consistency computation. *Artif. Intell.* 107, 1, 125–148.
2. HENTENRYCK, P. V., DEVILLE, Y., AND TENG, C.-M. 1992. A generic arc-consistency algorithm and its specializations. *Artif. Intell.* 57, 2-3, 291–321.
3. ZHOU, N.-F. 2005. Programming finite-domain constraint propagators in action rules. *To appear in Theory and Practice of Logic Programming (TPLP)*.