A Comparison of CP, IP, and SAT Solvers Through a Common Interface

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

Masato Tsuru
Faculty of Comp. Sci. & Sys. Eng.
Kyushu Institute of Technology
tsuru@cse.kyutech.ac.jp

Eitaku Nobuyama
Faculty of Comp. Sci. & Sys. Eng.
Kyushu Institute of Technology
nobuyama@ces.kyutech.ac.jp

***Abstract*—This paper presents a common interface for Prolog to three different types of discrete solvers including Constraint Programming (CP), Integer Programming (IP), and SAT solvers. The interface comprises primitives for creating decision variables, specifying constraints, and invoking a solver, possibly with an objective function to be optimized. Before a solver is actually called, the accumulated variables and constraints are transformed into a form acceptable to the solver. For a SAT solver, in particular, variables are Booleanized and constraints are compiled into CNF. Implemented in B-Prolog, the interface allows the programmer to use the features of the host language such as recursion, pattern matching, arrays, and loops to describe problems. The interface provides an easy and uniform platform for exploring different solvers and models. This paper compares the performance of the CLP(FD) of B-Prolog, the CPLEX IP solver, and the Lingeling SAT solver on several problems through the same interface and for each problem it compares a model that uses Boolean variables and another model that uses general integer variables. Our experience tells that it is effortless to switch from one solver to another.**

Keywords: Combinatorial search problems, Constraint programming, Integer programming, SAT solvers, Prolog.

I. Introduction

Many real-world problems, ranging from scheduling in industrial production lines, planning for intelligent robots, protein structure predication, resource allocation, cryptography, to various network optimization problems are combinatorial search problems that can be represented in terms of decision variables and constraints. There are three paradigms of systematic solvers for tackling these problems, namely, Constraint Programming (CP), Integer Programming (IP), and SAT solving. CP uses constraint propagation to prune search spaces and heuristics to guide search [22]; IP relies on LP relaxation and branch-and-cut to find optimal integer solutions [2]; and SAT solvers perform unit propagation and clause learning to prune search spaces, and employ heuristics and learned clauses to do non-chronological backtracking [18]. Past experiences tell us that finding an efficient solution normally requires extensive experimentation.

Many modeling languages have been developed for the solvers. For example, AMPL [1] and GAMS [9] are for mathematical programming (MP), Minizinc [20] is for CP, and ASP [5] can be considered as a modeling language for SAT and dynamic programming. Some ASP systems also provide constructs for describing constraints in a similar way as in CP. For example, Clasp [11] integrates ASP with Gecode[12] and EZASP [4] integrates ASP with B-Prolog. The OPL [14] is a modeling language for both CP and MP. Interfaces to solvers have also been added to Prolog. For example, the Eplex library [23] provides an interface for linking the ECL^{i}PS^e Prolog system with Mathematical Programming (MP), and the BEE compiler [19] for Prolog allows for efficiently checking the satisfibility of constraints with a SAT solver.

This paper presents a common interface for Prolog to three different types of discrete solvers. The interface comprises primitives for creating decision variables, specifying constraints, and invoking a solver, possibly with an objective function to be optimized. The supported constraints include Boolean, arithmetic, and the global constraints `alldifferent` and `element`. For a program, we can have it call a different solver just by changing the invoking call. Therefore, the interface greatly facilitates experimentation with different solvers and models. The interface is implemented in B-Prolog, which already has the language features for modeling such as recursion, pattern matching, declarative loops, and list builders. The interface makes B-Prolog a powerful modeling language for the solvers.

The implementation of the interface makes use of attributed variables in B-Prolog to accumulate constraints. When a constraint is posted, it is added to the list of accumulated constraints. Only when a solver-invoking call is executed are the constraints interpreted. If the solver is CP, the accumulated constraints are added into B-Prolog's constraint store and a labeling predicate is called to start the search. If the solver is SAT, all the variables are Booleanized and all the constraints are sent to the SAT solver after being compiled into CNF. If the solver is IP, all the constraints are converted to disequalities and sent to the IP solver. An answer found by the solver is returned to B-Prolog as bindings of the decision variables.

We have experimented through the interface with several solvers including the CLP(FD) solver of B-Prolog, the GLPK and CPLEX LP/MIP solvers, and the Lingeling SAT solver. In this paper, we compare the performance of the solvers on the graph coloring, N-queens, and the numberlink problems. For each problem, we consider two models, one using Boolean variables and the other using general integer variables. Our experiment results indicate that no solver is superior all the time: SAT is the best for Boolean models and CP tends to be better than SAT and IP for models that use general integer variables. Our experience tells that it is effortless to switch from one solver to another.

The remainder of the paper is structured as follows. The next section gives a brief introduction to the non-standard features provided by B-Prolog for modeling. Section III defines the primitives in the interface. Section IV gives several examples. Section V shows the implementation of the interface, mainly on how different constraints are compiled. Section VI presents the experimental results of different solvers on the example programs. Section VII concludes the paper.

II. B-Prolog

B-Prolog is an implementation of the standard Prolog language with several useful new features including tabling, constraint solving, and language constructs for modeling. The reader is referred to [28] for a survey of the system. This section introduces two non-standard features that are useful for modeling: arrays and loops.

A. Arrays and the array subscript notation

A structure can be used as a one-dimensional array, and a multi-dimensional array can be represented as a structure of structures. To facilitate creating arrays, B-Prolog provides a built-in, called `new_array`($X, Dims$), where X must be a free variable and $Dims$ a list of positive integers that specifies the dimensions of the array. For example, the call `new_array(X,[10,20])` binds `X` to a two dimensional array whose first dimension has 10 elements and second dimension has 20 elements. All the array elements are initialized to be free variables.

The built-in predicate `arg/3` can be used to access array elements, but it requires a temporary variable to store the result, and a chain of calls to access an element of a multi-dimensional array. To facilitate accessing array elements, B-Prolog supports the array subscript notation X`[`I_1`,...,`I_n`]`, where X is a structure and each I_i is an integer expression. This notation is interpreted as an array access when it occurs in an arithmetic expression, an arithmetic constraint, or as an argument of a call to the built-in `@=/2`.

The array subscript notation can also be used to access elements of lists. For example, the `nth/3` predicate can be defined as follows:

```
nth(I,L,E) :- E @= L[I].
```

Note that, for the array access notation `A[I]`, while it takes constant time to access the `I`th element if `A` is a structure, it takes O(`I`) time when `A` is a list.

B. Loops with `foreach` and list builders

B-Prolog provides a construct, called `foreach`, for iterating over collections, and list builders for constructing lists.

The `foreach` call has a very simple syntax and semantics. For example,

```
foreach(A in [a,b], I in 1..2, write((A,I)))
```

outputs four tuples: `(a,1)`, `(a,2)`, `(b,1)`, and `(b,2)`. The base `foreach` call has the form:

`foreach(`E_1 `in` $D_1, \ldots, E_n$ `in` $D_n, LocalVars, Goal$`)`

where E_i `in` D_i is called an *iterator* (E_i is called the *pattern* and D_i the *collection* of the iterator), $Goal$ is a callable term, and $LocalVars$ (optional) specifies a list of variables in $Goal$ that are local to each iteration. The pattern of an iterator is normally a variable but it can be any term; the collection is a list of terms or a range expression.

A list builder takes the form:

`[`T `:` E_1 `in` D_1`,` $\ldots$`,` E_n `in` D_n`,` $LocalVars$`,` $Goal$`]`

where $LocalVars$ (optional) specifies a list of local variables, $Goal$ (optional) is a callable term. This construct means that for each permutation of values $E_1 \in D_1, \ldots, E_n \in D_n$, if the instance of $Goal$ with renamed local variables is true, then T is added into the list. For example,

```
L @= [(A,I) : A in [a,b], I in 1..2]
```

binds `L` to `[(a,1),(a,2),(b,1),(b,2)]`. A list of this form is interpreted as a list builder if it occurs as an argument of a call to the built-in `@=/2` or in arithmetic constraints. List builders are not allowed in arithmetic contexts of Prolog since the arithmetic predicates such as `is/2` and `=:=/2` are inline built-ins, but a list builder is compiled into a call to a tail-recursive predicate and are hence not inline. For this reason, a call like `X is sum([I : I in 1..100])` is not allowed. We have to write it as `L @= [I : I in 1..100], X is sum(L)`.

Calls to `foreach` and list builders are translated into tail-recursive predicates. Therefore, there is no penalty to using these loop constructs compared with using recursion.

III. The Interface to Solvers

The interface comprises primitives for creating decision variables, specifying constraints, and invoking a solver, possibly with an objective function to minimize or maximize. Note that each of the new operators in the interface has a counterpart in the CLP(FD) language. For example, the operator used for equality constraints is `$=` in the interface and its counterpart in CLP(FD) is #=.

A. Variable creation

A decision variable is a logic variable with a domain. In this paper, we only consider discrete solvers and integer domains. The Boolean domain is treated as a special integer domain where 1 denotes `true` and 0 denotes `false`. The primitive `X :: D` declares that the domain of the variable `X` is `D` where `D` can be a list of integers or a range expression in the form $l..u$ which denotes the set of integers $\{l, l+1, \ldots, u\}$. For example, the call `X :: [0,1]` declares that `X` is a Boolean variable and the call `Y :: 1..4` declares that `Y`'s domain is `[1,2,3,4]`. Multiple variables with the same domain can be declared with one call. For example, the call `[X,Y] :: 0..1` declares that both `X` and `Y` are Boolean variables.

B. Constraints

There are three types of constraints: Boolean, arithmetic, and global.

A basic Boolean expression is made from constants (0 and 1), Boolean variables, and the following operators: `$/\` (and), `$\/` (or), `$\` (not or xor), `$<=>` (equivalent), and `$=>` (implication). The operator `$\` is used for two different purposes: `$\ X` means the negation of `X`, and `X $\ Y` means the exclusive or of `X` and `Y`.

An arithmetic constraint takes the form of $E_1\ R\ E_2$ where E_1 and E_2 are two arithmetic expressions and R is one of the following constraint symbols `$=` (equal), `$\=` (not equal), `$>=`, `$>`, `$=<`, and `$<`. An arithmetic expression is made of integers, domain variables, and the following arithmetic functions: `+` (sign or addition), `-` (sign or subtraction), `*` (multiplication), `div` (division), `mod` (remainder), `abs`, `min`, `max`, and `sum`. In addition to the basic standard syntax for expressions, the following forms of extended expressions are also acceptable. Let C be a Boolean expression, E_1 and E_2 be expressions, and L be a list of expressions `[`E_1`,`E_2`,`$\ldots$`,`E_n`]`. The following are valid expressions as well:

- `if`(C, E_1, E_2) The same as $C * E_1 + (1 - C) * E_2$.
- `min`(L) The minimum element of `L`, where L can be a list builder.
- `max`(L) The maximum element of `L`, where L can be a list builder.
- `min`(E_1, E_2) The minimum of E_1 and E_2.
- `max`(E_1, E_2) The maximum of E_1 and E_2.
- `sum`(L) The sum of the elements of L, where L can be a list builder.

An extended Boolean expression can also include arithmetic constraints as operands. In particular, the constraint `B $<=> (E1 $= E2)` is called a *reification* constraint which uses a Boolean variable `B` to indicate the satisfiablity of the arithmetic constraint `E1 $= E2`.

Only two global constraints, namely, `$alldifferent`(L) and `$element`(I, L, V), are currently supported.

C. Solver Invocation

The following primitives are provided to invoke a solver to find a valuation of the list of variables L that satisfies all the constraints:[1]

- `cp_solve`$(Options, L)$ Invoke the CP solver.
- `ip_solve`$(Options, L)$ Invoke the IP solver.
- `sat_solve`$(Options, L)$ Invoke the SAT solver.

where $Options$ is a list of options for the solver (e.g., `min`(E) and `max`(E) for optimization, and `dump` for dumping the model in some format). The reader is referred to the manual for the details of the primitives.

IV. Examples

We present in this section several modeling examples that use the interface. In all the examples, we use `sat_solve` to call the SAT solver. It can be replaced by `ip_solve` to call the IP solver, and `cp_solve` to call the CP solver. In section VI, we will use these examples to compare the three types of solvers.

[1]The primitive `lp_solve`$(Options, L)$ is provided to invoke the LP/MIP solver.

```
color(NV,NC):-
    new_array(A,[NV]),
    term_variables(A,Vars),
    Vars :: 1..NC,
    foreach(I in 1..NV-1, J in I+1..NV,
        ((edge(I,J);edge(J,I)) ->
             A[I] $\= A[J]
        ;
             true
        )
    ),
    sat_solve(Vars),
    writeln(Vars).
```

Fig. 1. Color a graph using one variable per node.

A. Graph coloring

Given an undirected graph $G = (V, E)$ where V is a set of vertices and E is a set of edges, and a set of colors, the graph coloring problem is to assign a color to each node in V so that no two adjacent vertices share the same color.

One model is to use one variable for each node whose value is the color assigned to the node. The program in Figure 1 encodes this model. The predicate `color(NV,NC)` colors a graph with `NV` vertices and `NC` colors. It is assumed that the vertices are numbered from 1 to `NV`, the colors are numbered from 1 to `NC`, and the edges are given as a predicate named `edge/2`,

The call `new_array(A,[NV])` creates an array of `NV` variables. The call `term_variables(A,Vars)` is a built-in which binds `Vars` to the list of variables occurring in `A`. The call `Vars :: 1..NC` restricts the domains of the variables to `1..NC`. The `foreach` loop posts constraints: for each pair of vertices `(I,J)`, if it is connected by an edge, then the color assigned to `I` (i.e., `A[I]`) is different from the color assigned to `J` (i.e., `A[J]`).

Another model is to use `NC` Boolean variables for each node, each variable corresponding to a color. The program in Figure 2 is based on this model. The call `new_array(A,[NV,NC])` creates a two dimensional array of size `NV`$\times$`NC`. The call `Vars :: [0,1]` make the variables Boolean. The first `foreach` loop ensures that for each node only one of its Boolean variables can take value 1. The next `foreach` loop ensures that no two adjacent vertices can have the same color. The formula

```
$\ A[I,K] $\/ $\ A[J,K]
```

ensures that the color `K` cannot be assigned to both node `I` and node `J`.

```
bcolor(NV,NC):-
    new_array(A,[NV,NC]),
    term_variables(A,Vars),
    Vars :: [0,1],
    foreach(I in 1..NV,
        sum([A[I,K] : K in 1..NC]) $= 1
    ),
    foreach(I in 1..NV-1, J in I+1..NV,
        ((edge(I,J);edge(J,I)) ->
             foreach(K in 1..NC,
                 $\ A[I,K] $\/ $\ A[J,K]
             )
         ;
             true
         )
    ),
    sat_solve(Vars),
    writeln(A).
```

Fig. 2. Color a graph using Boolean variables.

```
queens(N):-
    length(Qs,N),
    Qs :: 1..N,
    foreach(I in 1..N-1, J in I+1..N,
            (Qs[I] $\= Qs[J],
             abs(Qs[I]-Qs[J]) $\= J-I)
    ),
    sat_solve(Qs),
    writeln(Qs).
```

Fig. 3. A program for the N-queens problem.

B. N-queens problem

The N-queens problem is given as follows: Find a layout for N queen pieces on an N by N chessboard so that no queens attack each other. Two queens attack each other if they are placed in the same row, the same column, or the same diagonal.

For the N-queens problem, several models are possible. Since there are N queens and N rows, each column must have exactly one queen. One model is to use a variable for each column whose value indicates the number of row for the queen in this column. Figure 3 gives the program based on this model.

The call `length(Qs,N)` creates a list `Qs` of `N` variables. The call `Qs :: 1..N` restricts the domains of the variables to `1..N`. The `foreach` loop posts constraints to ensure that for each two columns `I` and `J` (`J` $>$ `I`) the two queens in the two columns are not in the same row (`Qs[I] $\= Qs[J]`) and not in the same diagonal (`abs(Qs[I]-Qs[J]) $\= J-I`).

Another possible model is to use a Boolean variable for each square on the board, 1 indicating a queen and 0 indicating empty. Figure 4 shows a program based on this model. The

```
bqueens(N):-
    new_array(Qs,[N,N]),
    term_variables(Qs,Vars),
    Vars :: 0..1,
    foreach(I in 1..N,     % 1 in each row
       sum([Qs[I,J] : J in 1..N]) $= 1
    ),
    foreach(J in 1..N,     % 1 in each column
       sum([Qs[I,J] : I in 1..N]) $= 1
    ),
    foreach(K in 1-N..N-1, % at most one
       sum([Qs[I,J] :
            I in 1..N, J in 1..N, I-J=:=K]
       ) $=< 1
    ),
    foreach(K in 2..2*N,   % at most one
       sum([Qs[I,J] :
            I in 1..N, J in 1..N, I+J=:=K]
       ) $=< 1
    ),
    sat_solve(Vars),
    writeln(Vars).
```

Fig. 4. A model using Boolean variables for N-queens.

constraint

```
foreach(K in 1-N..N-1,
   sum([Qs[I,J] :
        I in 1..N, J in 1..N, I-J=:=K]
   ) $=< 1
)
```

ensures that each diagonal that is parallel to the main diagonal contains at most one queen, and the constraint

```
foreach(K in 2..2*N,
   sum([Qs[I,J] :
        I in 1..N, J in 1..N, I+J=:=K]
   ) $=< 1
)
```

ensures that each diagonal that is parallel to the minor diagonal contains at most one queen.

C. The Numberlink problem

Numberlink is a logic puzzle made popular by Nikoli [21]. Figure 5 gives a solution to an example problem. Given a grid board of a certain dimension with some cells preoccupied by pairs of numbers, the puzzle is to find a path for each pair of the same numbers such that no paths overlap or intersect with each other. In Figure 5, the path for each pair of numbers is shown as connected cells filled with the same numbers.

The problem can be generalized as a graph labeling problem as follows: given an undirected graph G whose nodes are numbered from 1 to NV and a set of NC connection requirements each of which takes the form `connection`(I, V_1, V_2) where $1 \leq I \leq NC$, and V_1 and V_2 are nodes in G, the problem is

1	1	1	6	6	6	4	4	4	4
1	7	7	7	7	6	6	6	6	4
1	7	5	5	5	5	5	5	6	4
1	7	5	2	2	6	6	6	6	4
1	7	5	2	3	6	4	4	4	4
1	7	5	2	3	6	6	6	6	6
1	7	5	2	3	3	3	3	3	3
1	7	5	2	2	2	2	2	2	3
1	7	5	1	1	1	1	1	2	3
1	1	1	1	3	3	3	3	3	3

Fig. 5. A solution to an example numberlink problem.

to label the nodes with numbers from 1 to NC such that for each connection requirement connection(I, V_1, V_2) there is a path of nodes all labeled with I between the terminal nodes V_1 and V_2. We assume that the edges of the graph are given as a predicate edge/2 and we also assume that the predicate neighbors$(V, Neibs)$ is available which states that the set of nodes that are directly connected to V is $Neibs$.

The first model is to use a domain variable for each node whose value indicates the label assigned to the node. Figure 6 shows the encoding of this model. The loop

```
foreach(C in 1..NC, [V1,V2],
    (connection(C,V1,V2),
     Arr[V1] @= C,Arr[V2]@=C))
```

initializes the pre-occupied nodes. For each connection requirement connection(C,V1,V2), the label of V1 (Arr[V1]) and the label of V2 (Arr[V2]) are initialized to C.

The predicate constrain_node(Arr,V) ensures that the label assigned to the node V meets the requirement. If V is a terminal node in a connection requirement, then only one neighbor can receive the same label as V; otherwise, there are two neighbors with the same label as V.

Another model is to use a Boolean variable for each pair of connections and nodes. For a pair of a connection and a node, the node belongs to the connection if and only if the value of the Boolean variable for the pair is 1. Figure 7 shows the encoding of this model. The constraint

```
sum([Arr[C,Neib] : Neib in Neibs]) $=1
```

ensures that there is exactly one neighbor that belongs to the connection C and the constraint

```
Arr[C,V] $=>
  sum([Arr[C,Neib] : Neib in Neibs]) $= 2
```

ensure that if V belongs to the connection C then there are exactly two neighbors that belong to the connection.

V. Implementation

The implementation of the interface utilizes domain variables in B-Prolog. The primitive X :: D already exists in B-Prolog for creating finite domain variables. When D is

```
numberlink(NV,NC):-
    new_array(Arr, [NV]),
    term_variables(Arr, Vars),
    Vars::1..NC,
    foreach(C in 1..NC, [V1,V2],
        (connection(C,V1,V2),
         Arr[V1] @= C,
         Arr[V2]@=C)),
    foreach(V in 1..NV,constrain_node(Arr,V)),
    sat_solve(Vars),
    writeln(Vars).

constrain_node(Arr,V1):-
   neighbors(V1,Neibs),
   ((connection(C,V1,_);
     connection(C,_,V1))->
       sum([(Arr[V2]$=C) : V2 in Neibs]
       ) $=1
   ;
       sum([(Arr[V1]$=Arr[V2]) :
            V2 in Neibs]
       ) $=2
   ).
```

Fig. 6. A model for the numberlink problem.

an interval, the lower and upper bounds are attached to the variable X as two attributes. When D is a set of integers with holes, a bit vector is attached to X to represent the domain. Each domain variable also has an attached attribute that tells the list of constraints this variable is participating. When a constraint is posted, this constraint is just added into the lists of the participating variables. When a solver-invoking call is executed, the attached constraints are transformed into the form acceptable to the called solver. This section describes how constraints are transformed. All the transformation rules are well known. It's hoped that these transformation rules will help explain the experimental results to be presented in the next section.

A. Transformation for the CP solver

The transformation of constraints for the CP solver is straightforward. Each constraint is transformed to its corresponding CLP(FD) constraint. For example, X $= Y is transformed to X #= Y, and $alldifferent(L) is transformed to alldistinct(L). The primitive cp_solve(Options,L) is transformed to labeling(Options,L).

B. Transformation for the IP solver

For the IP solver, constraints are transformed into arithmetic equality and disequality ($\leq$ or $\geq$) constraints. It's straightforward to transform Boolean constraints. The following table gives the transformation rules for Boolean constraints where X and Y are Boolean variables.

```
numberlink(NV,NC):-
    new_array(Arr, [NC,NV]),
    term_variables(Arr, Vars),
    Vars::0..1,
    foreach(C in 1..NC, [V1,V2],
        (connection(C,V1,V2),
         Arr[C,V1] @= 1,
         Arr[C,V2]@=1)),
    foreach(V in 1..NV,
        sum([Arr[C,V] : C in 1..NC]
         ) $=1
    ),
    foreach(C in 1..NC, V in 1..NV,
        constrain_cv(C,V,Arr)
    ),
    sat_solve(Vars),
    writeln(Vars).

constrain_cv(C,V1,Arr):-
    neighbors(V1,Neibs),
    ((connection(C,V1,_);
      connection(C,_,V1))->
        sum([Arr[C,Neib] : Neib in Neibs]
        ) $= 1
    ;
        Arr[C,V] $=>
          sum([Arr[C,Neib] : Neib in Neibs])
          $=2
    ).
```

Fig. 7. Another model for the numberlink problem.

Boolean	Arithmetic
`X $/\ Y`	`X $= 1, Y $= 1`
`X $\/ Y`	`X+Y $>= 1`
`X $=> Y`	`X-Y $=< 0`
`X $<=> Y`	`X $= Y`
`X $\ Y`	`X+Y $= 1`
`$\ X`	`X $= 0`

Reification constraints are logical constraints and can be transformed into IP formulations using the existing techniques [2]. For example, consider the constraint `B $<=> (X $=< Y)` where `B` is a Boolean variable, and `X` and `Y` are two arbitrary integer domain variables. The constraint can be transformed to:

```
X-Y-M1*(1-B) $=< 0,
Y-X+1-M2*B $=< 0
```

where `M1` is `ubd(X)-lbd(Y)+1` and `M2` is `ubd(Y)-lbd(X)+2`.[2] Whenever `B=0`, the first constraint is trivially satisfied and `Y-X+1 $=< 0`, i.e., `X $> Y`, must be satisfied. Whenever `B=1`, the second constraint is trivially satisfied, and `X-Y $=< 0` must be satisfied. The implication constraint `B $=> (X $=< Y)` only requires the constraint `(X $=< Y)` to be satisfied whenever `B=1`, so it can be transformed to `X-Y-M1*(1-B) $=< 0`.

[2]`ubd(X)` stands for the upper bound and `lbd(X)` the lower bound of the domain of `X`.

Inequality constraints ($\neq$) are transformed to disequality constraints by introducing Boolean variables. For example, the constraint `X $\= Y` is transformed to:

```
B1 $=> (X $< Y),
B2 $=> (X $> Y),
B1+B2 $= 1
```

The last constraint `B1+B2 $= 1` entails that `B1` and `B2` cannot both be 0, i.e, `X` cannot be equal to `Y`.

C. Transformation for the SAT solver

Several methods have been proposed for compiling CSPs into SAT [8], [26], [10], [25], and several modeling languages have been developed for SAT such as ASP [5], the Sugar CSP [24], and BEE [19]. Our implementation supports direct encoding [8], [26] and log-encoding [10], [13], [15], [16]. For a 0-1 CSP, these two encoding methods generate the same CNF code. For a variable X whose domain is $\{a_1, a_2, \ldots, a_m\}$, the direct encoding method uses m Boolean variables $x_1, x_2, \ldots, x_m$, where $x_i = 1$ means $X = a_i$. Direct encoding generates clauses to ensure that exactly one of these variables can be 1. For a constraint, this method generates clauses to disallow conflict assignments. For example, for the constraint $X \neq Y (X, Y \in 1..2)$, four Boolean variables $(x_1(X = 1), x_2(X = 2), y_1(Y = 1), y_2(Y = 2))$ are used to encode the domains, and the two clauses $\neg x_1 \vee \neg y_1$ and $\neg x_2 \vee \neg y_2$ are generated to disallow the conflict assignments $(X = 1, Y = 1)$ and $(X = 2, Y = 2)$. In the log-encoding method, the domain $\{a_1, a_2, \ldots, a_m\}$ is encoded with $\lceil log_2(a) \rceil$ Boolean variables, where a is the maximum absolute value of the domain values, and a Boolean variable for the sign if the domain contains negative integers. Each of the valuations of the Boolean variables represents a value in the domain and clauses are generated to disallow combinations for the values that are not in the domain. Equality and disequality constraints are flattened to two types of primitive constraints in the form of $x > y$ and $x + y = z$, which are compiled further to logic adders and comparators in CNF. For other types of constraints, clauses are generated to disallow conflict valuations of the variables.

It is time consuming to compile constraints that involve a large number of variables. To speed up compilation, our implementation enforces interval consistency of the constraints before compilation and exploits bounds information of domains to avoid enumerating all permutations of domain values to find conflicts. In particular, for some constraints that involve only Boolean domain variables, it never enumerates all the permutations. For example, if the constraint is `B1+B2+...+Bn $> 0` where `Bi`'s are all Boolean variables, it converts the constraint into `B1 $\/ B2 $\/ ... $\/ Bn` without enumerating the values.

TABLE I
CPU TIMES FOR `color`.

Instance	CLP(FD)	Lingeling	CPLEX
1-graph_125-0	0.70	10.15	>200
2-graph_125-0	>200	>200	>200
3-graph_125-0	0.05	2.90	>200
4-graph_125-0	4.17	>200	>200
5-graph_125-0	0.74	1.98	>200

The *branch-and-bound* method is used to optimize an objective function. For the first run, the objective function is disregarded and only constraints are sent to the solver. Once an answer is returned, the objective function has a value. For the subsequent run, a new constraint is added to force the objective function to have a better value than the current best answer. This step is repeated until the set of constraints become unsatisfiable. At that point, the current best answer is the final best answer.

VI. Experimental Results

The interface has been implemented in B-Prolog and is available with version 7.8 [3]. We have compared three solvers including CLP(FD) of B-Prolog, the LP/MIP solver CPLEX [7], and the SAT solver Lingeling [17] on the three problems given in Section IV. Each of these solvers represents the state of the art in it's category. For this comparison, direct encoding was used by the SAT compiler. In general, log-encoding is more efficient that direct-encoding. The BProlog+SAT solver submitted to the MiniZinc Challenge 2012[3] employs log-encoding. We did the comparison on a PC with 3GHz Xeon(R) CPU 5160 and 160GB running CentOS Linux. All the CPU times given in the tables are in seconds.

Table I shows the CPU times taken by the solvers to run the `color` program on five instances from the third ASP solver competition [6]. All the instance graphs have 125 nodes. Clearly, CPLEX is not suited for this model. For each disequality constraint, the translated model contains two new Boolean variables. CLP(FD) failed to solve one instance within the 200-second limit and SAT failed two. Note that the CLP(FD) encoding of the problem that exploits symmetry breaking and global constraints solves all the instances easily [29].

Table II shows the CPU times for the `bcolor` program that uses Boolean variables. For this program, SAT performed better than CLP(FD), mainly because the first-fail labeling strategy of CLP(FD) becomes ineffective for Boolean variables. Again, CPLEX is not suited for this model.

Table III shows the CPU times for the `queens` program on five instances (N is the number of queens). CLP(FD) is a clear winner for this program. We have to mention that a large portion of the time taken by SAT is spent on compiling constraints into CNF. For example, for N=100, that portion accounts for 50% of the time.

[3]http://www.g12.csse.unimelb.edu.au/minizinc/challenge2012/challenge.html

TABLE II
CPU TIMES FOR `bcolor`.

Instance	CLP(FD)	Lingeling	CPLEX
1-graph_125-0	>200	53.04	>200
2-graph_125-0	>200	>200	>200
3-graph_125-0	>200	0.86	>200
4-graph_125-0	>200	>200	>200
5-graph_125-0	>200	0.61	191.11

TABLE III
CPU TIMES FOR `queens`.

Instance	CLP(FD)	Lingeling	CPLEX
10	0.00	0.02	0.18
20	0.00	0.99	47.71
30	0.00	7.97	>200
50	0.01	21.10	>200
100	0.02	34.02	>200

Table IV shows the CPU times for the `bqueens` program that uses Boolean variables. CPLEX is a clear winner, which is followed closely by Lingeling. CLP(FD) is the slowest.

Table V shows the results for the `numberlink` program on five instances taken from the third ASP competition. Lingeling is a clear winner: it solved all the instances while CLP(FD) and CPLEX each failed to solve the last instance.

Table VI shows the results for the `bnumberlink` program. Again, Lingeling is a clear winner. This model works so well with Lingeling that it solved all the 60 instances submitted to the third ASP competition including a very hard instance [27].

Overall, no solver is superior all the time. CLP(FD) won the non-Boolean models for the graph coloring and N-queens problems; CPLEX outperformed Lingeling by a small margin on the Boolean model of the N-queens problems; and Linegeling won the rest three models. Our results also confirm that models that use Boolean variables are not suited for CLP(FD). This is because, as mentioned above, the first-fail labeling strategy is not effective for Boolean variables. On the other hand, Lingeling works better on 0-1 integer programming models than general integer models.

VII. Final Remarks

This paper has presented a common Prolog interface to three different types of discrete solvers including CP, IP, and SAT solvers. With the interface and the new language constructs such as arrays and loops, Prolog can serve as a powerful modeling language for these solvers. CP, IP, and SAT solving are three different paradigms for solving combinatorial search problems. Each paradigm has its strengths and weaknesses: CP tends to be well suited to problems for which global constraints, symmetry breaking, and problem-specific propagation and labeling can be exploited; IP tends to be well suited to problems that can be naturally expressed with disequality constraints; and SAT tends to be suited to problems that are intrinsically Boolean. Our experimental results confirm to some extent these observations.

In reality, it requires extensive experimentation to find a right model and a right solver. The interface presented in this

TABLE IV
CPU TIMES FOR `bqueens`.

Instance	CLP(FD)	Lingeling	CPLEX
10	0.00	0.00	0.01
20	0.15	0.07	0.02
30	64.82	0.28	0.19
50	>200	1.21	0.08
100	>200	10.97	5.66

TABLE V
CPU TIMES FOR `numberlink`.

Instance	CLP(FD)	Lingeling	CPLEX
`2-numberlink-0-0.asp`	0.00	0.00	0.01
`9-numberlink-0-0.asp`	0.01	0.05	0.08
`111-numberlink-0-0.asp`	0.01	0.15	0.16
`120-numberlink-0-0.asp`	0.01	0.22	0.13
`48-numberlink-0-0.asp`	>200	6.42	>200

TABLE VI
CPU TIMES FOR `bnumberlink`.

Instance	CLP(FD)	Lingeling	CPLEX
`2-numberlink-0-0.asp`	0.00	0.00	0.00
`9-numberlink-0-0.asp`	0.01	0.01	0.15
`111-numberlink-0-0.asp`	0.03	0.02	0.37
`120-numberlink-0-0.asp`	0.04	0.04	0.97
`48-numberlink-0-0.asp`	>200	2.23	>200

paper provides a common and easy platform for experimenting with different solvers and models. Compared with other hosting languages such as C++ and Java, Prolog allows for natural encodings of constraints; and compared with other modeling languages such as OPL, Prolog as a general-purpose language allows for easy integration of constraint solving as a component into a large application.

With tabling for dynamic programming solutions and CLP(FD) for CSPs, the newly added common interface with SAT and LP/MIP certainly enriches the B-Prolog toolbox for solving combinatorial search and optimization problems. Nevertheless, unlike for ASP, the user of B-Prolog has to choose which tool to use for a problem specification. An interesting direction for further work is to develop a compiler for ASP in B-Prolog that automatically chooses what tools to use for a problem specification. Other further work includes investigating different compilation methods for SAT solvers, hybridization of solvers, and running multiple solvers in parallel in a distributed environment.

REFERENCES

[1] AMPL. www.ampl.com.

[2] Gautam M. Appa, Leonidas Pitsoulis, and H. Paul Williams. *Handbook on Modelling for Discrete Optimization*. Springer, 2010.

[3] B-Prolog. www.probp.com.

[4] Marcello Balduccini and Yulia Lierler. Practical and methodological aspects of the use of cutting-edge ASP tools. In *PADL*, pages 78–92, 2012.

[5] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[6] Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale, Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri. The third answer set programming competition. In *LPNMR*, pages 388–403, 2011.

[7] CPLEX. www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[8] Johan de Kleer. A comparison of ATMS and CSP techniques. In *IJCAI*, pages 290–296, 1989.

[9] GAMS. www.gams.com.

[10] Marco Gavanelli. The log-support encoding of CSP into SAT. In *CP*, pages 815–822, 2007.

[11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A User's Guide to gringo, clasp, clingo and iclingo. Technical report, University of Potsdam, 2011.

[12] Gecode. www.gecode.org.

[13] Allen Van Gelder. Another look at graph coloring via propositional satisfiability. *Discrete Applied Mathematics*, 156(2):230–243, 2008.

[14] Pascal Van Hentenryck. Constraint and integer programming in OPL. *INFORMS Journal on Computing*, 14:2002, 2002.

[15] Jinbo Huang. Universal booleanization of constraint models. In *CP*, pages 144–158, 2008.

[16] Kazuo Iwama and Shuichi Miyazaki. Sat-varible complexity of hard combinatorial problems. In *IFIP Congress (1)*, pages 253–258, 1994.

[17] Lingeling. fmv.jku.at/lingeling.

[18] Sharad Malik and Lintao Zhang. Boolean satisfiability: from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, 2009.

[19] Amit Metodi and Michael Codish. Compiling finite domain constraints to sat with bee. *TPLP*, 12(4-5):465–483, 2012.

[20] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *CP*, pages 529–543, 2007.

[21] Nikoli. www.nikoli.com/.

[22] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[23] Kish Shen and Joachim Schimpf. Eplex: Harnessing mathematical programming solvers for constraint logic programming. In *CP*, pages 622–636, 2005.

[24] Sugar. bach.istc.kobe-u.ac.jp/sugar/.

[25] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

[26] Toby Walsh. SAT v CSP. In *CP*, pages 441–456, 2000.

[27] Neng-Fa Zhou. www.probp.com/cp_sat_lp/numberlink_hard.html.

[28] Neng-Fa Zhou. The language features and architecture of B-Prolog. *TPLP, Special Issue on Prolog Systems*, 12(1-2):189–218, 2012.

[29] Neng-Fa Zhou, Agostino Dovier, and Yuanlin Zhang. BPSolvers solutions to the third ASP competition problems. *ALP Newsletter*, (June), 2011.