The Picat-SAT Compiler

Neng-Fa Zhou[1] and Håkan Kjellerstrand[2]

[1] CUNY Brooklyn College & Graduate Center
[2] hakank.org

Abstract. SAT has become the backbone of many software systems. In order to make full use of the power of SAT solvers, a SAT compiler must encode domain variables and constraints into an efficient SAT formula. Despite many proposals for SAT encodings, there are few working SAT compilers. This paper presents Picat-SAT, the SAT compiler in the Picat system. Picat-SAT employs the sign-and-magnitude log encoding for domain variables. Log-encoding for constraints resembles the binary representation of numbers used in computer hardware, and many algorithms and optimization opportunities have been exploited by hardware design systems. This paper gives the encoding algorithms for constraints, and attempts to experimentally justify the choices of the algorithms for the addition and multiplication constraints. This paper also presents preliminary, but quite encouraging, experimental results.

1 Introduction

SAT solvers' performance has drastically improved during the past 20 years, thanks to the inventions of techniques from conflict-driven clause learning, backjumping, variable and value selection heuristics, to random restarts [1, 4, 23]. SAT has become the backbone of many software systems, including specification languages for model generation and checking [9, 17, 18, 22], planning [21, 27], program analysis and test pattern generation [28], answer set programming [6, 12], and solvers for general constraint satisfaction problems (CSPs) [15, 19, 29, 31, 33].

In order to fully exploit the power of SAT solvers, a compiler is needed to Booleanize constraints as formula in the conjunctive normal form (CNF) or some other acceptable form. The encodings of constraints have big impact on the runtime of SAT solvers [2]. Several encodings of domain variables have been proposed, including *sparse* encoding [13, 33], *order* encoding [8, 25, 31], and *log* encoding [16]. Log-encoding has less propagation power than sparse encodings for certain constraints [11], but is much more compact than other encodings. The FznTini compiler [15] adopts two's complement representation for domain variables.

For each encoding, there are different ways to Booleanize constraints. The hardness of a CNF formula is normally determined by several parameters, such as the number of clauses and the number of variables. There is no commonly accepted definition for the optimality of SAT formulas, and SAT compiler writers

have to rely on intensive experimentation to find good formulas that are compact and enhance propagation opportunities for the used SAT solver.

We have developed a SAT compiler for Picat [35], called *Picat-SAT*, which adopts the *sign-and-magnitude* log-encoding for domain variables. Log-encoding for constraints resembles the binary representation of numbers used in computer hardware, and many algorithms and optimization opportunities have been exploited by hardware design systems. Despite many proposals for SAT encodings, there are few working SAT compilers, and there are basically no working SAT compilers that are based on log-encoding.[3]

In this paper, we describe the algorithms for Booleanizing constraints employed in Picat-SAT. All of the algorithms are either well known or can be easily derived from the well-known algorithms used in hardware designs. This paper attempts to experimentally justify the choices of the algorithms for the addition and multiplication constraints. This paper also compares Picat-SAT with the following solvers: Sugar and Azucar, two SAT compilers based on order-encoding, and the winning CP solvers of MiniZinc Challenge 2015. The experimental results show that Picat-SAT is competitive with Sugar and Azucar, and that Picat-SAT outperforms the MiniZinc Challenge winners on some of the benchmarks.

Section 2 gives the API of the constraint modules in Picat. Section 3 describes the sign-and-magnitude log encoding adopted in Picat-SAT for domain variables. Section 4 details the algorithms used in Picat-SAT for compiling constraints, including the *enumeration* algorithm for the addition constraint $X + Y = Z$, the *shift-and-add* algorithm for the multiplication constraint $X \times Y = Z$, the use of the Espresso logic optimizer for the table constraint, and the decomposition algorithms for global constraints. Section 5 presents and analyzes the experimental results. We also implemented a transformation algorithm for the addition constraint, and the Karatsuba algorithm for the multiplication constraint. These two algorithms are described in the appendices.

2 Picat's Constraint Modules

Picat is a simple, and yet powerful, logic-based multi-paradigm programming language aimed for general-purpose applications [34, 35]. Picat provides solver modules, including the `sat`, `cp`, and `mip` modules. All these three modules implement the same set of basic linear constraints. The `cp` and `sat` modules also implement non-linear and global constraints, and the `mip` module also supports real-domain variables. The common interface that Picat provides for the solver modules allows seamless switching from one solver to another.

As a constraint programming language, Picat resembles CLP(FD): the operator `::` is used for domain constraints, the operators `#=`, `#!=`, `#>`, `#>=`, `#<`, and `#=<` are for arithmetic constraints, and the operators `#/\` (and), `#\/` (or), `#^` (xor), `#~` (not), `#=>` (if), and `#<=>` (iff) are for Boolean constraints. Picat supports many global constraints, such as `all_different/1`, `element/3`, and `cumulative/4`. In

[3] The FznTini compiler is not maintained.

addition to intensional constraints, Picat also supports extensional constraints, or *table constraints*.

For example, the following gives a program in Picat for solving the Sudoku problem:

```
import sat.

main =>
    Board = {{5,3,_,_,7,_,_,_,_},
             {6,_,_,1,9,5,_,_,_},
             {_,9,8,_,_,_,_,6,_},
             {8,_,_,_,6,_,_,_,3},
             {4,_,_,8,_,3,_,_,1},
             {7,_,_,_,2,_,_,_,6},
             {_,_,_,_,_,_,_,_,_},
             {_,_,_,_,_,_,_,_,_},
             {_,_,_,_,_,_,_,_,_},
             {_,_,_,_,_,_,_,_,_}},
    sudoku(Board),
    foreach(Row in Board) writeln(Row) end.

sudoku(Board) =>
   N = Board.len,
   Vars = Board.vars(),
   Vars :: 1..N,
   foreach (Row in Board) all_different(Row) end,
   foreach (J in 1..N)
      all_different([Board[I,J] : I in 1..N])
   end,
   M = round(sqrt(N)),
   foreach (I in 1..M..N-M, J in 1..M..N-M)
      all_different([Board[I+K,J+L] : K in 0..M-1, L in 0..M-1])
   end,
   solve(Vars).
```

The first line imports the `sat` module, which defines the used built-ins by this program, including the operator `::`, the global constraint `all_different`, and the `solve` predicate for labeling variables. For a given board, the `sudoku` predicate retrieves the length of the board (`Board.len`), extracts the variables from the board (`Board.vars()`),[4] generates the constraints, and calls `solve(Vars)` to label the variables. The first `foreach` loop ensures that each row of `Board` has different values. The second `foreach` loop ensures that each column of `Board` has different values. The list comprehension `[Board[I,J] : I in 1..N]` returns the `J`th column of `Board` as a list. Let `M` be the size of the sub-squares (`M = round(sqrt(N))`). The third `foreach` loop ensures that each of the `N` `M`×`M` squares has different values. As demonstrated by this example, Picat's language constructs such as functions, arrays, loops, and list comprehensions make Picat as powerful as other modeling languages, such as OPL [14] and MiniZinc [26], for CSPs.

3 The Sign-and-Magnitude Log Encoding

Picat-SAT employs the so called *log-encoding* for domain variables. For a domain variable, $\lceil log_2(n) \rceil$ Boolean variables are used, where n is the maximum

[4] Picat supports the dot-notation for chaining function calls. The function call `Board.vars()` is the same as `vars(Board)`.

absolute value in the domain. If the domain contains both negative and positive values, then another Boolean variable is used to represent the sign. In this paper, for a log-encoded domain variable X, X`.s` denotes the sign, X`.m` denotes the magnitude, which is a vector of Boolean variables $<X_{n-1}X_{n-2}\ldots X_1X_0>$.

This *sign-and-magnitude* encoding is simple, and is well suited to Booleanizing certain constraints, such as `abs(`X`) =` Y and `-`X `=` Y. However, this encoding requires a clause to disallow negative zero if the domain contains values of both signs. This extra clause is unnecessary if 2's complement encoding is used, as in [15]. Each combination of values of these Boolean variables represents a valuation for the domain variable. If there are holes in the domain, then disequality ($\neq$) constraints are generated to disallow assignments of those hole values to the variable. Also, inequality constraints ($\geq$ and $\leq$) are generated to prohibit assigning out-of-bounds values to the variable.

For small-domain variables that can be encoded with 14 bits, including the sign bit, Picat-SAT calls the logic optimizer, Espresso [5], to generate an optimal CNF formula. For example, for the domain constraint `X :: [-2,-1,2,1]`, Espresso returns the following two CNF clauses:

$\texttt{X}_0 \vee \texttt{X}_1$
$\neg\texttt{X}_0 \vee \neg\texttt{X}_1$

which make it impossible to assign -3, 0, or 3 to `X`.

4 Booleanization of Constraints

Picat-SAT flattens intensional constraints into *primitive*, *reified*, and *implicative* constraints. A primitive constraint is one of the following: $\Sigma_i^n B_i\ r\ c$ (r is $=$, $\geq$, or $\leq$, and c is 1 or 2) , $X\ r\ Y$ (r is $=$, $\neq$, $>$, $\geq$, $<$, or $\leq$), `abs(`X`) =` Y, `-`X `=` Y, $X+Y=Z$, and $X\times Y=Z$, where B_i is a Boolean variable, and X, Y, and Z are integers or log-encoded integer domain variables.[5]

The sign-and-magnitude encoding facilitates Booleanization of some of the primitive constraints. For example, `abs(`X`) =` Y is translated to:

$X\texttt{.m} = Y\texttt{.m} \wedge Y\texttt{.s = 0}$

and the constraint `-`X `=` Y is translated to:

$(X\texttt{.m} = Y\texttt{.m}) \wedge (X\texttt{.s} \neq Y\texttt{.s} \vee X\texttt{.m = 0})$

The at-least-one constraint $\Sigma_i^n B_i \geq 1$ is encoded into one CNF clause:

$B_1 \vee B_2 \vee \ldots \vee B_n$

[5] The operators `div` and `mod` can be expressed by using the multiplication operator $\times$. In the implemented version of Picat-SAT, Pseudo-Boolean constraints are treated in the same way as other linear constraints, except for the special case $\Sigma_i^n B_i\ r\ c$ (c =1 or 2).

The at-least-two constraint $\Sigma_i^n B_i \geq 2$ is converted into n at-least-one constraints: for each $n-1$ variables, the sum of the variables is at least one. The at-most-one constraint $\Sigma_i B_i \leq 1$ is encoded into CNF by using Jingchao Chen's algorithm [7], which splits the sequence of Boolean variables into two subsequences, and encodes the sum $\Sigma_i^n B_i$ as the Cartesian product of the two subsequences. The at-most-two constraint is converted into n at-most-one constraints. The exactly-one constraint $\Sigma_i^n B_i = 1$ is converted into a conjunction of an at-least-one constraint and an at-most-one constraint. The exactly-two constraint is compiled in the same way.

A recursive algorithm is utilized to encode each of the binary primitive constraints of the form $X\ r\ Y$, where r is $=, \neq, >, \geq, <$, or $\leq$. For example, consider $X \geq Y$. This constraint is translated to the following:

$$X.\mathrm{s} = 0 \wedge Y.\mathrm{s} = 1 \vee$$
$$X.\mathrm{s} = 1 \wedge Y.\mathrm{s} = 1 \Rightarrow X.\mathrm{m} \leq Y.\mathrm{m} \vee$$
$$X.\mathrm{s} = 0 \wedge Y.\mathrm{s} = 0 \Rightarrow X.\mathrm{m} \geq Y.\mathrm{m}$$

Let $X.\mathrm{m} = <X_{n-1}X_{n-2}\ldots X_1X_0>$ and $Y.\mathrm{m} = <Y_{n-1}Y_{n-2}\ldots Y_1Y_0>$, where X_i and Y_i are Boolean variables, $i = 0, ..., n-1$. The following function returns the CNF formula for $X.\mathrm{m} \geq Y.\mathrm{m}$:

```
ge(<X_{n-1}X_{n-2}...X_1X_0>, <Y_{n-1}Y_{n-2}...Y_1Y_0>):
  if n = 1 then return the CNF formula for X_0 ≥ Y_0;
  return the CNF formula for X_{n-1} > Y_{n-1} ∨
              X_{n-1} = Y_{n-1} ∧ ge(<X_{n-2}...X_1X_0>, <Y_{n-2}...Y_1Y_0>);
```

A reified constraint has the form $B \Leftrightarrow C$, and an implicative constraint has the form $B \Rightarrow C$, where B is a Boolean variable and C is a primitive constraint. The reified constraint $B \Leftrightarrow C$ is equivalent to $B \Rightarrow C$ and $\neg B \Rightarrow \neg C$, where $\neg C$ is the negation of C. Let $C_1 \wedge \ldots \wedge C_n$ be the CNF formula of C after Booleanization. Then $B \Rightarrow C$ can be encoded into $C'_1 \wedge \ldots \wedge C'_n$, where $C'_i = C_i \vee \neg B$ for $i = 1, ..., n$.

4.1 Booleanization of the Addition Constraint

This subsection considers the Booleanization of the constraint $X+Y=Z$, where the operands are log-encoded integers or integer-domain variables. If all of the operands are non-negative (i.e., $X\texttt{.s = 0}$ and $Y\texttt{.s = 0}$), then the constraint can be rewritten into the unsigned addition $X\texttt{.m}+Y\texttt{.m} = Z\texttt{.m}$. Let $X\texttt{.m} = X_{n-1}\ldots X_1X_0$, $Y\texttt{.m} = Y_{n-1}\ldots Y_1Y_0$, and $Z\texttt{.m} = Z_n\ldots Z_1Z_0$. The unsigned addition can be Booleanized by using logic adders as follows:

$$\begin{array}{rcccc} & X_{n-1} & \ldots & X_1 & X_0 \\ + & Y_{n-1} & \ldots & Y_1 & Y_0 \\ \hline Z_n & Z_{n-1} & \ldots & Z_1 & Z_0 \end{array}$$

A half-adder is employed for $X_0 + Y_0 = C_1Z_0$, where C_1 the carry-out. For each other position i ($0 < i \leq n-1$), a full adder is employed for $X_i+Y_i+C_i = C_{i+1}Z_i$. The top-most bit of Z, Z_n, is equal to C_n.

If any of the variables in the addition constraint has both negative and positive values in the domain, then the Booleanization of the constraint is not so straightforward. Although the compiler makes efforts not to create negative-domain variables when flattening constraints into primitive ones, some problem variables may be negative. Picat-SAT adopts the *enumeration* algorithm for the general addition constraint. We also implemented another algorithm, called *transformation*, which replaces negative domain values by non-negative domain values in the addition constraint. This algorithm is given in Appendix A, together with the experimental results that justify the choice of the enumeration algorithm.

The *enumeration* algorithm translates the addition constraint $X + Y = Z$ into the following conjunction of conditional constraints, each of which takes care of a combination of signs of the operands.

$$
\begin{array}{l}
X\texttt{.s} = 0 \wedge Y\texttt{.s} = 0 \Rightarrow Z\texttt{.s} = 0 \wedge X\texttt{.m}{+}Y\texttt{.m} = Z\texttt{.m} \\
X\texttt{.s} = 1 \wedge Y\texttt{.s} = 1 \Rightarrow Z\texttt{.s} = 1 \wedge X\texttt{.m}{+}Y\texttt{.m} = Z\texttt{.m} \\
X\texttt{.s = 0} \wedge Y\texttt{.s = 1} \wedge Z\texttt{.s = 1} \Rightarrow X\texttt{.m}{+}Z\texttt{.m} = Y\texttt{.m} \\
X\texttt{.s = 0} \wedge Y\texttt{.s = 1} \wedge Z\texttt{.s = 0} \Rightarrow Y\texttt{.m}{+}Z\texttt{.m} = X\texttt{.m} \\
X\texttt{.s = 1} \wedge Y\texttt{.s = 0} \wedge Z\texttt{.s = 0} \Rightarrow X\texttt{.m}{+}Z\texttt{.m} = Y\texttt{.m} \\
X\texttt{.s = 1} \wedge Y\texttt{.s = 0} \wedge Z\texttt{.s = 1} \Rightarrow Y\texttt{.m}{+}Z\texttt{.m} = X\texttt{.m}
\end{array}
$$

This encoding does not introduce extra new variables, except carry variables used in the Booleanization of the unsigned addition constraints. Since only one combination of signs is possible, one vector of carry variables can be used for the Booleanization of all of the unsigned addition constraints.

4.2 Booleanization of the Multiplication Constraint

This subsection considers the Booleanization of the constraint $X \times Y = Z$, where the operands are log-encoded integers or integer-domain variables. Since the sign of any operand is determined by the signs of the other two operands, this section only considers unsigned multiplication. Many algorithms have been used in the hardware-design community for fast multiplication. Picat-SAT adopts the *shift-and-add* algorithm for multiplication constraints. We also implemented Karatsuba's *divide-and-conquer* algorithm [20]. This algorithm is given in Appendix B, together with the experimental results that justify the choice of the shift-and-add algorithm.

Let $Y\texttt{.m}$ be $Y_{n-1} \ldots Y_1 Y_0$. The *shift-and-add* algorithm generates the following conditional constraints for the multiplication constraint $X \times Y = Z$.

$$
\begin{array}{l}
Y_0 = 0 \Rightarrow S_0 = 0 \\
Y_0 = 1 \Rightarrow S_0 = X \\
Y_1 = 0 \Rightarrow S_1 = S_0 \\
Y_1 = 1 \Rightarrow S_1 = (X << 1) + S_0 \\
\vdots \\
Y_{n-1} = 0 \Rightarrow S_{n-1} = S_{n-2}
\end{array}
$$

$Y_{n-1} = 1 \Rightarrow S_{n-1} = (X << (n-1)) + S_{n-2}$
$Z = S_{n-1}$

The operation $(X << i)$ shifts the binary string of X to left by i positions. Let the length of the binary string of X be m. The length of S_0 is m, that of S_1 is $m+1$, and so on. So the total number of auxiliary Boolean variables that are used in S_i's is $\Sigma_{i=m}^{(n+m-2)} i$. In addition, auxiliary variables are used for carries in the additions. For example, in the addition $S_i = (X << i) + S_{i-1}$, $m+i$ auxiliary Boolean variables are needed.

This algorithm can be improved to reduce the number of new variables when Y is an integer. For the addition $S_i = (X << i) + S_{i-1}$, if S_{i-1} is a 0 vector, then the Boolean variables can be copied from X to S_i, and so no new variables are necessary. Booth's algorithm [3], which examines multiple digits of Y at once, can also be used to further reduce the number of variables if Y is a constant.[6]

4.3 Booleanization of the Table Constraint

In log-encoding, a table constraint can easily be converted into a truth table. Consider the constraint `table_in({X,Y},[{1,2},{1,3},{2,1},{2,2},{3,1}])`. Let `X.m` be <X_1,X_0>, and `Y.m` be <Y_1,Y_0>. Since the sign bits are known to be 0, they are ignored. The above table constraint can be represented as the truth table shown in Figure 1. The task of compiling a table constraint amounts to finding a CNF formula to equivalently represent its truth table.

X_1	X_0	Y_1	Y_0
0	1	1	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1

Fig. 1. The truth table of `table_in({X,Y},[{1,2},{1,3},{2,1},{2,2},{3,1}])`

Just as for variables with small domains, Picat-SAT uses the logic optimizer, Espresso, to find compact SAT encodings for small truth tables. Espresso returns a *disjunctive-normal-form* (DNF) formula for a true table. In order to easily convert the DNF formula into CNF, Picat-SAT feeds Espresso with the complementary truth table, i.e., setting the output of *in* tuples to *0* and the output of *out* tuples to *1*. For example, Espresso returns the following DNF clauses for the complement of the truth table in Figure 1.

[6] Booth's algorithm has not yet been implemented in Picat-SAT.

$\neg \mathtt{X}_0 \wedge \mathtt{Y}_1 \wedge \mathtt{Y}_0$
$\mathtt{X}_1 \wedge \mathtt{X}_0 \wedge \mathtt{Y}_1$
$\neg \mathtt{X}_1 \wedge \neg \mathtt{X}_0$
$\neg \mathtt{X}_1 \wedge \neg \mathtt{Y}_1$
$\neg \mathtt{Y}_1 \wedge \neg \mathtt{Y}_0$

The original truth table is represented as the negation of the DNF formula, which is a conjunction of the following CNF clauses:

$\mathtt{X}_0 \vee \neg \mathtt{Y}_1 \vee \neg \mathtt{Y}_0$
$\neg \mathtt{X}_1 \vee \neg \mathtt{X}_0 \vee \neg \mathtt{Y}_1$
$\mathtt{X}_1 \vee \mathtt{X}_0$
$\mathtt{X}_1 \vee \mathtt{Y}_1$
$\mathtt{Y}_1 \vee \mathtt{Y}_0$

Espresso is slow when the number of variables is large. For a truth table with more than 14 Boolean variables, Picat-SAT breaks the table down into smaller tables by enumerating part of the variables. For example, the Boolean table in Figure 1 can be broken down into two tables, one assuming $\mathtt{X}_1$=1 and the other assuming $\mathtt{X}_1$=0.

4.4 Decomposition of Global Constraints

Picat-SAT decomposes global constraints into primitive ones in a straightforward manner. This subsection briefly overviews the algorithms adopted by Picat-SAT for decomposing several well-used global constraints.

`all_different(`L`)` : Let L be $[E_1, E_2, \ldots, E_n]$. Picat-SAT decomposes this constraint into $E_i \neq E_j$ for $i, j = 1, \ldots, n, i < j$. For an *assignment-type* all-different constraint, in which the number of values in the domains is equal to the number of variables, Picat-SAT also generates the following redundant constraints: each value is assigned to exactly one variable.

`circuit(`L`)` : Picat-SAT first generates `all_different(`L`)` for this constraint, which is guaranteed to be an assignment-type constraint. In order to prevent sub-cycles in the circuit, Picat-SAT uses a new variable, O_i, for each value i, which indicates the ordering number of i in the circuit, assuming the ordering number of value 1 being 1. Let L be $[E_1, E_2, \ldots, E_n]$. If $E_i = j$, then a constraint is generated to ensure that O_j is the successor of O_i, meaning that $O_j = O_i + 1$ if $O_i < n$, and $O_j = 1$ if $O_i = n$. This is a standard decomposition algorithm, which is used by other compilers, such as the `mzn2fzn` compiler [24].

`cumulative(`*Starts*`,` *Durations*`,` *Resources*`,` *Limit*`)` : For the `cumulative` constraint, two basic decomposition algorithms exist, namely, *time decomposition* and *task decomposition* [10]. Let n be the number of tasks, S_i be the earliest start time, and E_i to the latest end time of task i, $i \in 1, \ldots, n$. The time-decomposition algorithm generates constraints to ensure the following: for each time point from $\mathtt{min}_{i=1}^{n}(S_i)$ to $\mathtt{max}_{i=1}^{n}(E_i)\mathtt{-1}$, the total amount of

resources consumed by the running tasks cannot exceed the resource limit. The task-decomposition algorithm only generates constraints to ensure that the resource limit is not exceeded at the start and end times of each of the tasks. Picat-SAT adopts the task decomposition algorithm. All of the resource constraints are pseudo-Boolean constraints. As mentioned above, Picat-SAT splits pseudo-Boolean constraints into adders and comparators, as it does for other arithmetic constraints.

`element(I,L,V)` : Let L be $[E_1, E_2, \ldots, E_n]$. If L contains variables, Picat-SAT decomposes this constraint into the following : for each value i in `1..n`, $I = i \Rightarrow V = E_i$; otherwise, if L is ground, Picat-SAT encodes the constraint as a table constraint.

5 Experimental Results

Picat-SAT, which is implemented in Picat, has about 5,000 lines of code, excluding comments. In addition to the encoding algorithms described in the paper, it also incorporates some specialization and optimization techniques. Picat-SAT eliminates common subexpressions in constraints. For example, when a reification constraint $B \Leftrightarrow C$ is generated, Picat-SAT tables it and reuses the variable B, rather than introducing a new variable for the primitive constraint C, when C is encountered again. Picat-SAT has been an entrant in the MiniZinc Challenge three times since 2013, and was the only purely SAT-based CSP solver in 2015. This section experimentally compares Picat-SAT, as implemented in Picat version 1.4, with two other SAT compilers, and reports on the Picat-SAT's performance in MiniZinc Challenge 2015.

Table 1 compares Picat-SAT with two other SAT compilers: Sugar and Azucar.[7] Sugar is a mature SAT compiler based on order encoding [30], and Azucar is a successor to Sugar based on compact order encoding [32]. The benchmarks are from Sugar's package. The `all_different` constraint is used in *golombRuler*, *knightTour*, and *nqueens*, and no other global constraints are used. The table shows the the size of the generated code (the number of variables, #vars, and the number of clauses, #cls) and the CPU time (in seconds) taken to run the generated code using Lingeling version 587f on a Cygwin notebook with 2.4GHz Intel i5 and 4GB RAM.[8]

It is not surprising that Picat-SAT generates more compact code than Sugar and Azucar for most of the benchmarks because log-encoding is known to be more compact than order-encoding. For the *knightTour* and *nqueens* benchmarks, both of which include the assignment-type `all_different` constraint, Picat-SAT uses more Boolean variables than Sugar and Azucar. This is because

[7] Efforts were also made to compare Picat-SAT with FznTini [15], a log-encoding based SAT compiler, and with meSAT [29], which supports the hybrid of order and sparse encodings. FznTini is for an old version of FlatZinc, and the default setting of meSAT did not perform better than Sugar on the benchmarks.

[8] The CPU time does not include the compile time for any of the SAT compilers.

Picat-SAT generates redundant constraints to ensure that each value is assigned to exactly one variable.

Picat-SAT also compares favorably well with Sugar and Azucar in terms of CPU time on most of the benchmarks. Picat-SAT is significantly slower than Sugar and Azucar on *knightTour* and *nqueens*. These results reveal that Picat-SAT's choice algorithm for `all_different` is not the best.

Table 1. A comparison of three SAT compilers

Benchmark	Picat			Sugar			Azucar		
	#vars	#cls	time	#vars	#cls	time	#vars	#cls	time
golombRuler-10-55	6438	27194	**36.50**	4584	190594	92.53	5635	44937	60.32
golombRuler-8	3024	12698	**0.22**	5111	422046	3.04	2988	35847	0.44
jss-ft10	13613	68880	5.90	76617	703313	6.24	8227	119126	**1.50**
knightTour-5	3051	21401	8.19	1296	20510	**0.83**	1667	9123	2.36
knightTour-7	11970	114818	374.36	4944	141818	99.68	5919	44059	**85.53**
magicSquare-5x5	2742	20302	1.68	3504	73717	0.72	2052	16239	**0.29**
magicSquare-9x9	33606	178328	**48.58**	60960	5463881	167.24	19405	317128	57.36
nqueens-200	1003112	3997099	1925.52	159196	18646098	**131.96**	284400	4004662	397.82
oss-gp03-01	785	3535	**0.19**	11126	46075	0.69	786	6231	2.13
oss-gp10-01	25540	139168	**2.90**	343189	6514888	40.40	17216	544880	5.79
socialGolfer-3-2-5	315	3360	0.12	810	2355	0.31	1680	5250	**0.10**
socialGolfer-6-3-7	9702	163863	**13.19**	22545	79236	13.78	46248	184980	14.04
tdsp-C1-1	6361	25187	0.67	1226	17631	**0.26**	4586	36997	0.64
tdsp-C2-1	16079	67032	5.09	2720	61572	**0.72**	11746	112436	7.12

Table 2 and Table 3 show the points scored by Picat-SAT and the top three CP solvers in MiniZinc Challenge 2015. Table 2 shows the scores for the benchmarks that do not include global constraints,[9] and Table 3 shows the scores for the benchmarks that include global constraints. MiniZinc Challenge uses a scoring procedure based on the Borda count voting system; the higher a score is, the better. Despite the 0 score from the *project-planning* benchmark, Picat-SAT performed quite well on the benchmarks that do not include global constraints: it scored higher than OR-Tools, and could have performed better than iZplus had the points from *project-planning* been counted in. Picat-SAT is less competitive with the winners on the benchmarks that use global constraints. Picat-SAT scored 0 points on *is(Min)*, which uses `circuit` and `table`, on *largeschedul-ing*, which uses `cumulative`, and on *tdsp(Min)*, which uses `inverse`. Overall, Picat-SAT had the highest scores on 5 of 20 benchmarks.

6 Concluding Remarks

This paper has described Picat-SAT, which employs the sign-and-magnitude log encoding for domain variables. Log-encoding of domain variables resembles the

[9] Picat-SAT scored 0 on *project-planning* because `mzn2fzn` failed to specialize the `element` constraint, which prevented Picat's FlatZinc interpreter from functioning. It was understood that `mzn2fzn` would specialize generic global constraints into specific ones once the types of the arguments are known.

Table 2. MiniZinc Challenge scores on benchmarks that do not use global constraints

Benchmark	OR-Tools	Opturion CPX	iZplus	Picat-SAT
freepizza(MIN)	4.03	10.96	**12.00**	3.01
grid-colouring(MIN)	1.91	11.27	5.28	**11.54**
nmseq(SAT)	**13.46**	10.75	3.19	0.60
project-planning(MIN)	5.00	**14.67**	10.33	0.00
radiation(MIN)	0.00	**11.04**	8.56	10.40
triangular(MAX)	7.75	1.00	9.00	**12.25**
zephyrus(MIN)	**9.00**	7.50	7.50	6.00
(TOTAL)	41.16	67.19	55.86	43.79

Table 3. MiniZinc Challenge scores on benchmarks that use global constraints

Benchmark	OR-Tools	Opturion CPX	iZplus	Picat-SAT
costas-array(SAT)	**7.35**	4.86	5.06	0.73
cvrp(MIN)	7.07	8.59	**12.30**	2.04
gfd-schedule(MIN)	2.48	10.53	**11.99**	5.00
is(MIN) TOTAL	7.56	7.88	**12.56**	0.00
largescheduling(MIN)	**13.00**	10.00	0.00	0.00
mapping(MIN)	2.00	**14.91**	7.00	5.09
multi-knapsack(MAX)	4.18	**10.75**	4.26	9.80
opd(MIN)	8.07	2.50	7.00	**12.43**
open_stacks(MIN)	10.10	5.19	2.00	**12.71**
p1f(MIN)	**14.53**	3.95	2.01	3.51
roster(MIN)	**11.72**	4.44	11.40	2.44
spot5(MAX)	8.00	6.68	2.34	**12.98**
tdtsp(MIN)	8.24	**11.26**	10.50	0.00
(TOTAL)	104.31	101.54	88.43	66.72

binary representation of numbers used in computer hardware. It is a very compact encoding, and a large repository of algorithms has been developed by the hardware design community. Interestingly, few SAT compilers for general constraints have been developed that are based on this encoding. This is partly because of the negative research results reported on log-encoding's failure to maintain arc consistency, and partly because of the engineering effort required to implement the encoding. This paper has given the adapted algorithms employed by Picat-SAT, and has presented preliminary, but quite encouraging, experimental results.

One direction for future work is to examine more encoding algorithms and optimization techniques for Booleanizing constraints, especially global and pseudo-Boolean constraints. SMT solvers have been shown to be successful in handling various types of constraints, including arithmetic constraints. Another direction for future work is to compare Picat-SAT with SMT solvers on SMT-LIB benchmarks.

References

1. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability.* IOS Press, 2009.
2. Magnus Bjork. Successful SAT encoding techniques. JSAT Addendum, 2009.

3. Andrew D. Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, IV, 1951.
4. Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Comput. Surv.*, 38(4):1–54, 2006.
5. Robert King Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.
6. Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.
7. Jingchao Chen. A new SAT encoding of the at-most-one constraint. In *Proc. of the 9th Int. Workshop of Constraint Modeling and Reformulation*, 2010.
8. James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI*, pages 1092–1097, 1994.
9. Jean-Louis Boulanger (Editor). *Formal Methods Applied to Industrial Complex Systems: Implementation of the B Method*. Wiley, 2014.
10. Kathryn Glenn Francis and Peter J. Stuckey. Explaining circuit propagation. *Constraints*, 19(1):1–29, 2014.
11. Marco Gavanelli. The log-support encoding of CSP into SAT. In *CP*, pages 815–822, 2007.
12. Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *IJCAI*, pages 386–, 2007.
13. Ian P. Gent. Arc consistency in SAT. In *ECAI*, pages 121–125, 2002.
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. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.
18. Ethan K. Jackson. A module system for domain-specific languages. *Theory and Practice of Logic Programming*, 14(4-5):771–785, 2014.
19. Peter Jeavons and Justyna Petke. Local consistency and SAT-solvers. *JAIR*, 43:329–351, 2012.
20. Anatoly Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Proc. the USSR Academy of Sciences*, 145:293–294, 1962.
21. Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.
22. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2004.
23. Sharad Malik and Lintao Zhang. Boolean satisfiability: from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, 2009.
24. Kim Marriott, Peter J. Stuckey, Leslie De Koninck, and Horst Samulowitz. A MiniZinc tutorial. http://www.minizinc.org/downloads/doc-latest/minizinc-tute.pdf.
25. Amit Metodi and Michael Codish. Compiling finite domain constraints to SAT with BEE. *Theory and Practice of Logic Programming*, 12(4-5):465–483, 2012.

26. 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.
27. Jussi Rintanen. Planning as satisfiability: Heuristics. *Artif. Intell.*, 193:45–86, 2012.
28. Rolf Drechsler Stephan Eggersgl. *High Quality Test Pattern Generation and Boolean Satisfiability.* Springer, 2012.
29. Mirko Stojadinovic and Filip Maric. meSAT: multiple encodings of CSP to SAT. *Constraints*, 19(4):380–403, 2014.
30. Sugar. bach.istc.kobe-u.ac.jp/sugar/.
31. Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
32. Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. Azucar: A SAT-based CSP solver using compact order encoding. In *SAT*, pages 456–462, 2012.
33. Toby Walsh. SAT v CSP. In *CP*, pages 441–456, 2000.
34. Neng-Fa Zhou and Jonathan Fruhman. A User's Guide to Picat. http://picat-lang.org.
35. Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint Solving and Planning with Picat.* Springer, 2015.

A The Transformation Algorithm for $X + Y = Z$

The *transformation* algorithm replaces negative domain values by non-negative domain values in the addition constraint. For the constraint $X + Y = Z$, if any of the operands has a domain with negative values, this method finds the smallest integer k such that $\texttt{abs}(X) < 2^{k-1}$, $\texttt{abs}(Y) < 2^{k-1}$, and $\texttt{abs}(Z) < 2^k$, and transforms the original addition constraint to $Z' = X' + Y'$, where $X' = X + 2^{k-1}$, $Y' = Y + 2^{k-1}$, and $Z' = Z + 2^k$. The newly introduced variables X', Y', and Z' are guaranteed not to include negative values in their domains, and therefore the constraint $X' + Y' = Z'$ can be Booleanized as $X'.\texttt{m} + Y'.\texttt{m} = Z'.\texttt{m}$.

The question is how to Booleanize the newly introduced constraints. Consider the constraint $X' = X + 2^{k-1}$. Let $X.\texttt{m} = X_{k-1} \ldots X_n X_{n-1} \ldots X_1 X_0$, and $X'.\texttt{m} = X'_{k-1} \ldots X'_1 X'_0$ ($k > n$, $X_i = 0$ for $i = k-1, \ldots, n$). Note that 0s are added in the high end of $X.\texttt{m}$ in order for it to have the same length as $X'.\texttt{m}$. The constraint $X' = X + 2^{k-1}$ can be translated to a conjunction of the following two conditional constraints:

$$X.\texttt{s} = 0 \Rightarrow X_0 = X'_0 \ \wedge \ \ldots \ \wedge \ X_{k-2} = X'_{k-2} \ \wedge \ X'_{k-1} = 1$$
$$X.\texttt{s} = 1 \Rightarrow X.\texttt{m} + X'.\texttt{m} = 2^{k-1}$$

Note the unsigned addition $X.\texttt{m} + X'.\texttt{m} = 2^{k-1}$ can be Booleanized using a much simpler logic than for the general addition. In particular, no new variables are necessary for carries in the addition. In order for the constraint to hold, the following formula must hold:

$$X_0 = X'_0$$
$$\texttt{for } i = 1..k-2$$

$$
\begin{aligned}
&X_{i-1} = 0 \ \wedge \ X'_{i-1} = 0 \Rightarrow X_i = X'_i \\
&X_{i-1} = 1 \ \wedge \ X'_{i-1} = 1 \Rightarrow X_i \neq X'_i \\
&X_{i-1} \neq X'_{i-1} \Rightarrow X_i \neq X'_i \\
X'_{k-1} &= 0
\end{aligned}
$$

The two Boolean variables at the lowest end, X_0 and X'_0, must be equal. Otherwise, the lowest bit of the sum cannot be 0. The intuition of the logic of the `for` loop is that if there is an 1 at some position, then there must be exactly one 1 at each of the higher positions in order for the sum to be equal to 2^{k-1}. Since no negative zero is allowed, at least one of X_is is 1. In order for the sum to be equal to 2^{k-1}, X'_{k-1} must be 0.

The idea of removing negative domain values through transformation is well known in linear programming. Unlike in linear programming, however, the transformation is only performed locally on negative-domain variables in addition constraints, and all other constraints, such as `abs(`X`) =` Y, are not affected.

For a negative-domain variable X that occurs in multiple addition constraints, an optimization can be employed to avoid introducing new variables unnecessarily. The compiler memorizes the transformation $X' = X + 2^{k-1}$ in a table. In case the same transformation is required by another addition constraint, then the compiler fetches X' from the table, rather than introducing new variables.

Table 4 compares the enumeration (enum) and transformation (trans) algorithms, using the constraint $Z = X+Y$, where X and Y are in $-N..N$. The code size is not informative on which encoding is better: the enum encoding uses fewer variables but more clauses than the trans encoding.[10] The time shows that the enum encoding is favorable. This result led to the adoption the enum encoding by Picat-SAT for addition constraints.

Table 4. A comparison of two encodings for $Z = X + Y$

N	enum			trans		
	#vars	#cls	time(s)	#vars	#cls	time(s)
5000	56	1121	0.058	100	632	0.070
10000	60	1211	0.060	107	682	0.060
15000	60	1199	0.052	107	670	0.080
20000	64	1301	0.060	114	732	0.070
25000	64	1295	0.059	114	726	0.080

B Karatsuba's Divide and Conquer Algorithm for $X \times Y = Z$

Karatsuba's algorithm [20] is a well-known algorithm for multiplying big integers. It can be applied to numbers in any base. The basic idea of the algorithm is

[10] Note that the code size sometimes decreases with N because of the domain constraints.

to divide and conquer, splitting large numbers into smaller numbers, until the numbers are small enough.

Let X and Y be two N-digit binary strings. If N is small enough, then the multiplication constraint is Booleanized using shift-and-add. If N is big, then this algorithm selects an integer M ($1 \leq M \leq N-1$) and splits both X and Y into two parts as follows:

$$X = X_h \times 2^M + X_l$$
$$Y = Y_h \times 2^M + Y_l$$

where X_l and Y_l are less than 2^M. The product $X \times Y$ is then

$$Z = X \times Y = Z_2 \times 2^{2M} + Z_1 \times 2^M + Z_0$$

where

$$Z_0 = X_l \times Y_l$$
$$Z_1 = (X_h + X_l)(Y_h + Y_l) - Z_2 - Z_0$$
$$Z_2 = X_h \times Y_h$$

The resulting formulas require three multiplications, plus some additions, subtractions, and shifts.

Table 5 compares the shift-and-add (saa) and Karatsuba (kara) algorithms, using the constraint $Z = X \times Y$, where X and Y are in $-N..N$. The base-case size for the Karatsuba algorithm is set to 3 (i.e., $X.\mathtt{m} \leq 7$ or $Y.\mathtt{m} \leq 7$).

Two observations can be made about the results: first, the code sizes only grow slightly with `N` no matter which algorithm is used; and second, kara uses more variables and generates more clauses than saa. The Karatsuba algorithm is well used for multiplying big integers and its advantage cannot be witnessed unless the operands are really big. Nevertheless, big domains rarely occur in CSPs. This experimental comparison naturally resulted in the adoption of the shift-and-add algorithm by Picat-SAT for multiplication constraints.

Table 5. A comparison of two encodings for $Z = X \times Y$

N	saa			kara		
	#vars	#cls	time(s)	#vars	#cls	time(s)
5000	443	2880	0.236	1380	6872	0.382
10000	512	3357	0.270	1594	7931	0.425
15000	513	3353	0.271	1595	7927	0.442
20000	585	3831	0.226	1845	9175	0.481
25000	585	3827	0.223	1845	9171	0.491