Compiling Finite Domain Constraints to SAT with BEE: the Director’s Cut

Michael Codish  Yoav Fekete  Amit Metodi

Department of Computer Science, Ben-Gurion University, Israel

Abstract

BEE is a compiler which facilitates solving finite domain constraints by encoding them to CNF and applying an underlying SAT solver. In BEE constraints are modeled as Boolean functions which propagate information about equalities between Boolean literals. This information is then applied to simplify the CNF encoding of the constraints. We term this process *equi-propagation*. A key factor is that considering only a small fragment of a constraint model at one time enables to apply stronger, and even complete reasoning to detect equivalent literals in that fragment. Once detected, equivalences propagate to simplify the entire constraint model and facilitate further reasoning on other fragments. BEE is described in several recent papers: [20], [19] and [21]. In this paper, after a quick review of BEE, we elaborate on two undocumented details of the implementation: the hybrid encoding of cardinality constraints and complete equi-propagation. We then describe ongoing work aimed to extend BEE to consider binary representation of numbers.

1 Introduction

BEE (Ben-Gurion Equi-propagation Encoder) is a tool which applies to encode finite domain constraint models to CNF. BEE was first introduced in [19] and is further described in [21]. During the encoding process, BEE performs optimizations based on equi-propagation [20] and partial evaluation to improve the quality of the target CNF. BEE is implemented in (SWI) Prolog and can be applied in conjunction with any SAT solver. It can be downloaded from [18] where one can also find examples of its use. This version of BEE is configured to apply the CryptoMiniSAT solver [24] through a Prolog interface [6]. CryptoMiniSAT offers direct support for xor clauses, and BEE can be configured to take advantage of this feature.

A main design choice of BEE is that integer variables are represented in the unary order-encoding (see, e.g. [9, 3]) which has many nice properties when applied to small finite domains. In the *order-encoding*, an integer variable $X = [x_1, \ldots, x_n]$ in the domain $[0, \ldots, n]$ is represented by a bit vector $X = [x_1, \ldots, x_n]$. Each bit $x_i$ is interpreted as $X \geq i$ implying that $X$ is a monotonic non-increasing Boolean sequence. For example, the value 3 in the interval $[0, 5]$ is represented in 5 bits as $[1, 1, 1, 0, 0]$. It is well-known that the order-encoding facilitates the propagation of bounds. Consider an integer variable $X = [x_1, \ldots, x_n]$ with values in the interval $[0, n]$. To restrict $X$ to take values in the range $[a, b]$ (for $1 \leq a \leq b \leq n$), it is sufficient to assign $x_a = 1$ and $x_{a+1} = 0$ (if $b < n$). The variables $x_a$ and $x_{a'}$ for $1 \leq a' < a$ and $b < b' \leq n$ are then determined *true* and *false*, respectively, by *unit propagation*. For example, given $X = [x_1, \ldots, x_9]$, assigning $x_3 = 1$ and $x_6 = 0$ propagates to give $X = [1, 1, 1, x_4, x_5, 0, 0, 0, 0]$, signifying that $\text{dom}(X) = \{3, 4, 5\}$. We observe an additional property of the order-encoding for $X = [x_1, \ldots, x_n]$: its ability to specify that a variable cannot take a specific value $0 \leq v \leq n$ in its domain by equating two variables: $x_v = x_{v+1}$. This indicates that the order-encoding is well-suited not only to propagate lower and upper bounds, but also to represent integer variables with an arbitrary,
finite set, domain. For example, given $X = \{x_1, \ldots, x_9\}$, equating $x_2 = x_3$ imposes that $X \neq 2$. Likewise $x_5 = x_6$ and $x_7 = x_8$ impose that $X \neq 5$ and $X \neq 7$. Applying these equalities to $X$ gives, $X = \{x_1, x_2, x_4, x_5, x_6, x_7, x_9\}$ (note the repeated literals), signifying that $\text{dom}(X) = \{0, 1, 3, 4, 6, 8, 9\}$.

The order-encoding has many additional nice features that can be exploited to simplify constraints and their encodings to CNF. To illustrate one, consider a constraint of the form $A + B = 5$ where $A$ and $B$ are integer values in the range between 0 and 5 represented in the order-encoding. At the bit level (in the order encoding) we have: $A = [a_1, \ldots, a_5]$ and $B = [b_1, \ldots, b_5]$. The constraint is satisfied precisely when $B = [\lnot a_5, \ldots, \lnot a_1]$. Equi-propagation derives the equations $E = \{b_1 = \lnot a_5, \ldots, b_5 = \lnot a_4\}$ and instead of encoding the constraint to CNF, we apply the substitution indicated by $E$, and remove the constraint which is a tautology given $E$.

2 Compiling Constraints with BEE

BEE is a constraint modeling language similar to the subset of FlatZinc [22] relevant for finite domain constraint problems. The full language is presented in Table 1. Boolean constants “true” and “false” are viewed as (integer) values “1” and “0”. Constraints are represented as (a list of) Prolog terms. Boolean and integer variables are represented as Prolog variables and may be instantiated when simplifying constraints. In Table 1, $X$ and $Xs$ (possibly with subscripts) denote a Boolean literal and a vector of literals, $I$ (possibly with subscript) denotes an integer variable, and $c$ (possibly with subscript) denotes an integer constant. On the right column of the table are brief explanations regarding the constraints. The table introduces 26 constraint templates.

Constraints (1-2) are about variable declarations: Booleans and integers. Constraint (3) expresses a Boolean as an integer value. Constraints (4-8) are about Boolean (and reified Boolean) statements. The special cases of Constraint (5) for $\text{bool} \_\text{array} \_\text{or}(X_1, \ldots, X_n)$ and $\text{bool} \_\text{array} \_\text{xor}(X_1, \ldots, X_n)$ facilitate the specification of clauses and of xor clauses (supported directly in the CryptoMiniSAT solver [24]). Constraint (8) specifies that sorting a bit pair $[X_3, X_4]$ (decreasing order) results in the pair $[X_3, X_4]$. This is a basic building block for the construction of sorting networks [4] used to encode cardinality (linear Boolean) constraints during compilation as described in [2] and [5]. Constraints (9-14) are about integer relations and operations. Constraints (15-20) are about linear (Boolean, Pseudo Boolean, and integer) operations. Constraints (21-26) are about lexical orderings of Boolean and integer arrays.

The compilation of a constraint model to a CNF using BEE goes through three phases: (1) Unary bit-blasting: integer variables (and constants) are represented as bit vectors in the order-encoding. (2) Constraint simplification: three types of actions are applied: equi-propagation, partial evaluation, and decomposition of constraints. Simplification is applied repeatedly until no rule is applicable. (3) CNF encoding: the best suited encoding technique is applied to the simplified constraints.

Bit-blasting is implemented through Prolog unification. Each declaration of the form $\text{new} \_\text{int}(I, c_1, c_2)$ triggers a unification $I = [1, \ldots, 1, x_{c_1+1}, \ldots, x_{c_2}]$. To ease presentation we assume that integer variables are represented in a positive interval starting from 0 but there is no such limitation in practice (BEE supports also negative integers). BEE applies ad-hoc equi-propagators. Each constraint is associated with a set of ad-hoc rules. The novelty is that the approach is not based on CNF, as in previous works (for example [15, 10, 13, and 16]), but rather driven by the bit blasted constraints that are to be encoded to CNF. For example, Figure 1 illustrates the rules for the $\text{int} \_\text{plus}$ constraint. For an integer $X = \langle x_1, \ldots, x_n \rangle$, we write: $X \geq i$ to denote the equation $x_i = 1$, $X < i$ to de-
Declaring Variables

(1) `new_bool(X)` declare Boolean X
(2) `new_int(I, c1, c2)` declare integer I, c1 ≤ I ≤ c2
(3) `bool2int[X, I]` (X ⇔ I = 1) ∧ (¬X ⇔ I = 0)

Boolean (reified) Statements

1. `bool_eq(x1, x2)` or `bool_eq(x1, ¬x2)` x1 = x2 or x1 = ¬x2
2. `bool_array_op([x1, ..., x_n])` x_i op x_k ...
3. `bool_array_op_ref([x1, ..., x_n], x)` x_i op x_k ... op x_n ⇒ x
4. `bool_op_reif(X, x2, X)` x_i op x_k ⇒ x
5. `comparator(X, x2, x3) x3 = [x3, x2]

Integer relations (reified)

1. `int_relf(I1, I2)` I1 rel I2
2. `int_relf(I1, I2, X)` I1 rel I2 ⇔ X
3. `int_array_allDiff([I1, ..., I_n])` \( \bigwedge_{1 \leq j \leq 1 \neq I_j} \)
4. `int_abs([I1, 1])` |I| = 1
5. `int_op(I1, I2)` I1 op I2 = I
6. `int_array_op([I1, ..., I_n, I])` I1 op’ ... op’ I_n = I

Linear Constraints

1. `bool_array_sum_rel([X1, ..., X_n], I)` ((X1) rel I)
2. `bool_array_pb_rel([C1, ..., C_n], [X1, ..., X_n], I)` ((C_i * X_j) rel I)
3. `bool_array_sum_modK([X1, ..., X_n], C, I)` ((X1) mod c = I)
4. `bool_array_sum_rel([I1, ..., I_n], I)` ((I1) rel I)
5. `bool_array_lin_rel([C1, ..., C_n], [I1, ..., I_n], I)` ((C1 * I2) rel I)
6. `bool_array_sum_modK([I1, ..., I_n], C, I)` ((I1) mod c = I)

Lexical Order

1. `bool_arrays_lex(Xs1, Xs2)` Xs1 precedes (leq) Xs2 in the lex order
2. `bool_arrays_lex_relf(Xs1, Xs2)` Xs1 precedes (lt) Xs2 in the lex order
3. `bool_array_lexlt_relf(Xs1, Xs2)` X ⇔ Xs1 precedes (lt) Xs2 in the lex order
4. `int_arrays_lex(Is1, Is2)` Is1 precedes (leq) Is2 in the lex order
5. `int_arrays_lexlt(Is1, Is2)` Is1 precedes (lt) Is2 in the lex order

Table 1: Syntax of BEE Constraints.

Note the equation \( x_i = 0 \), \( X \neq i \) to denote the equation \( x_i = x_{i+1} \), and \( X = i \) to denote the pair of equations \( x_i = 1, x_{i+1} = 0 \). We view \( X = \langle x_1, ..., x_n \rangle \) as if padded with sentinel cells such that all cells “to the left of” \( x_1 \) take value 1 and all cells “to the right of” \( x_n \) take the value 0. This facilitates the specification of the “end cases” in the formalism. The first four rules of Figure 1 capture the standard propagation behavior for interval arithmetic. The last two rules apply when one of the integers in the relation is a constant. There are symmetric cases when replacing the role of \( X \) and \( Y \).

When an equality of the form \( X = L \) (between a variable and a literal or a constant) is detected, then equi-propagation is implemented by unifying \( X \) and \( L \) and applies to all occurrences of \( X \) thus propagating to other constraints involving \( X \).

Decomposition is about replacing complex constraints (for example about arrays) with simpler constraints (for example about array elements). Consider, for instance, the constraint `int_array_plus(As, Sum)`. It is decomposed to a list of `int_plus` constraints applying a straight-
forward divide and conquer recursive definition. At the base case, if \( \text{As} = \{A\} \) then the constraint is replaced by a constraint of the form \( \text{int\_eq}(A, \text{Sum}) \) which equates the bits of \( A \) and \( \text{Sum} \), or if \( \text{As} = \{A_1, A_2\} \) then it is replaced by \( \text{int\_plus}(A_1, A_2, \text{Sum}) \). In the general case \( \text{As} \) is split into two halves, then constraints are generated to sum these halves, and then an additional \( \text{int\_plus} \) constraint is introduced to sum the two sums.

CNF encoding is the last phase in the compilation of a constraint model. Each of the remaining simplified (bit-blasted) constraints is encoded directly to a CNF. These encodings are standard and similar to those applied in various tools such as Sugar [20].

### 3 Cardinality Constraints in BEE

Cardinality Constraints take the form \( \sum\{x_1, \ldots, x_n\} \leq k \) where the \( x_i \) are Boolean literals, \( k \) is a constant, and the relation \( \leq \) might be any of \( \{=, <, >, \leq, \geq\} \). There is a wide body of research on the encoding of cardinality to CNF. We focus on those using sorting networks. For example, the presentations in [11], [5], and [11 [24] describe the use of odd-even sorting networks to encode pseudo Boolean and cardinality constraints to Boolean formula. We observe that for applications of this type, it suffices to apply “selection networks” [14] rather than sorting networks. Selection networks apply to select the \( k \) largest elements from \( n \) inputs. In [14], Knuth shows a simple construction of a selection network with \( O(n \log^2 k) \) size whereas, the corresponding sorting network is of size \( O(n \log^2 n) \). Totalizers [3] are similar to sorting networks except that the merger for two sorted sequences involves a direct encoding with \( O(n^2) \) clauses instead of \( O(n \log n) \) clauses. Totalizers have been shown to give better encodings when cardinality constraints are not excessively large. BEE enables the user to select encodings based on sorting networks, totalizers or a hybrid approach which is further detailed below.

Consider the constraint \( \text{bool\_array\_sum\_eq}(\text{As}, Y) \) in a context where \( \text{As} \) is a list of \( n \) Boolean literals and integer variable \( Y \) defined as \( \text{new\_int}(Y, 0, n) \). BEE applies a divide and conquer strategy. If \( n = 1 \), the constraint is trivial and satisfied by unifying \( Y = \text{As} \). If \( n = 2 \) and \( \text{As} = \{A_1, A_2\} \) then \( \text{Ys} = \{Y_1, Y_2\} \) and the constraint is decomposed to \( \text{comparator}(A_1, A_2, Y_1, Y_2) \). In the general case, where \( n > 2 \), the constraint is decomposed as follows where \( \text{As}_1 \) and \( \text{As}_2 \) are a partitioning of \( \text{As} \) such that \( |\text{As}_1| = n_1, |\text{As}_2| = n_2 \), and \( |n_1 - n_2| \leq 1 \):

\[
\text{bool\_array\_sum\_eq}(\text{As}, Y) \xrightarrow{\text{decompose}} \text{new\_int}(T_1, 0, n_1), \quad \text{bool\_array\_sum\_eq}(\text{As}_1, T_1), \quad \text{new\_int}(T_2, 0, n_2), \quad \text{bool\_array\_sum\_eq}(\text{As}_2, T_2), \quad \text{int\_plus}(T_1, T_2, Y)
\]

This decomposition process continues as long as there remain \( \text{bool\_array\_sum\_eq} \) and when it terminates the model contains only \( \text{comparator} \) and \( \text{int\_plus} \) constraints. The interesting discussion is with regards to the \( \text{int\_plus} \) constraints where BEE offers two options and depending on this choice the original \( \text{bool\_array\_sum\_eq} \) constraint then takes the form either of a sorting network [4] or of a totalizer [3]. So, consider a constraint \( \text{int\_plus}(A, B, C) \) where \( A = [A_1, \ldots, A_m], B = [B_1, \ldots, B_p] \) and \( C = [C_1, \ldots, C_{m+p}] \) represent integer variables in the order encoding. A unary adder leads to a direct encoding of the sum of two unary numbers. It involves \( O(n^2) \) clauses where \( n \) is the size of the inputs and as a circuit it has “depth” 1. The encoding introduces the following clauses where \( (1 \leq i \leq m) \) and \( (1 \leq j \leq p) \):

- \( \bigwedge_i (A_i \rightarrow C_i) \)
- \( \bigwedge_j (B_j \rightarrow C_j) \)
- \( \bigwedge_{i,j} (A_i \land B_j \rightarrow C_{i+j}) \)
- \( \bigwedge_i (\neg A_i \rightarrow \neg C_{p+i}) \)
- \( \bigwedge_j (\neg B_j \rightarrow \neg C_{n+j}) \)
- \( \bigwedge_{i,j} (\neg A_i \land \neg B_j \rightarrow \neg C_{i+j-1}) \)
Figure 2: Relative size of CNF encodings for cardinality: adders, hybrid & mergers. On the left number of clauses, and on the right number of added variables.

An alternative encoding for \(\text{int\_plus}(A, B, C)\) is obtained by means of a recursive decomposition based on the so called odd-even merger from Batcher’s construction [4]. It leads to an encoding with \(O(n \log n)\) clauses where \(n\) is the size of the inputs and as a circuit it has “depth” \(\log n\). The decomposition is as follows (ignoring the base cases) where \(A_o, A_e, B_o, B_e\) are partitions of \(A\) and \(B\) to their odd and even positioned elements, \(C_o, C_e\) are new unary variables defined with the appropriate domains, and where \(\text{combine}(C_o, C_e, C)\) signifies a set of comparator constraints and is defined as \(\bigwedge_i \text{comparator}(C_o + i, C_e + i, C_{2i}, C_{2i} + 1)\):

\[
\text{int\_plus}(A, B, C) \xrightarrow{\text{decompose}} \text{int\_plus}(A_o, B_o, C_o), \quad \text{int\_plus}(A_e, B_e, C_e), \quad \text{combine}(C_o, C_e, C)
\]

In addition to the encodings based on unary adders (direct) and mergers (recursive decomposition), BEE offers a combination of the two which we call “hybrid”. The intuition is simple: in the hybrid approach we perform recursive decomposition as for odd-even mergers, but only so long as the resulting CNF is predetermined to be smaller than the corresponding unary adder. So, it is just like a merger except that the base case is a unary adder. Before each decomposition of \(\text{int\_plus}\), BEE evaluates the benefit (in terms of CNF size) of decomposing the constraint as a merger and takes the smaller of the two.

Figure 2 depicts the size of CNF encodings for the constraint \(\text{int\_plus}(A, B, C)\) where \(|A| = |B| = n\). The left graph illustrates the number of clauses in the three encodings. The unary adder has fewest number of clauses for inputs of size 7 or less. The hybrid encoding is always just slightly smaller than the merger. Each time a merger is decomposed to an adder it is just about of the same number of clauses. In contrast, in the right graph we see that the encoding never introduces fresh variables, and as the size of the input increases so does the benefit of the hybrid approach in number of added variables.

Now let us consider the constraint \(\text{bool\_array\_sum\_leq}(A, k)\) where \(A\) is a list of \(n\) Boolean literals and \(k\) is a constant. Assume as before that \(A_1\) and \(A_2\) are a partitioning of \(A\) such that \(|A_1| = n_1, |A_2| = n_2, \text{ and } |n_1 - n_2| \leq 1\). A naive decomposition might proceed as follows:
But we can do better. In BEE we decompose bool_array_sum_leq(As, k) as follows:

This is correct because the constraint int_plus(T₃, T₂, k) defines \( T₃ = k - T₂ \) and so we have

\[
(T₁ + T₂ \leq k) \leftrightarrow (T₁ \leq T₃) \land (T₂ + T₃ = k)
\]

This encoding is preferable because the int_plus(T₃, T₂, k) constraint is encoded with 0 clauses (due to equi-propagation) and the int_leq(T₁, T₃) constraint in \( O(k) \) clauses. Whereas in the naive version the int_plus(T₁, T₂, Y) is encoded in \( O(n \log(n)) \) or \( O(n^2) \) (sorting network or direct) and the int_leq(Y, k) is encoded with 0 clauses.

### 4 Complete Equi-Propagation in BEE

Equi-propagation is about inferring Boolean Equalities, \( x = \ell \), implied from a given CNF formula \( \varphi \) where \( x \) is a Boolean variable and \( \ell \) a Boolean constant or literal. Complete equi-propagation (CEP) is about inferring all such equalities. Equi-propagation in BEE is based on ad-hoc rules and thus incomplete. However, BEE allows the user to specify, for given sets of constraints in a model, that CEP is to be applied (instead of ad-hoc equi-propagation). CEP generalizes the notion of a backbone. The backbone of a CNF, \( \varphi \), is the set of literals that are true in all models of \( \varphi \), thus corresponding to the subset of equations, \( x = \ell' \) obtained from CEP where \( \ell' \) is a Boolean constant. Backbones prove useful in applications of SAT such as model enumeration, minimal model computation, prime implicant computation, and also in applications which involve optimization (see for example, [17]). Assigning values to backbone variables reduces the size of the search space while maintaining the meaning of the original formula. In exactly the same way, CEP identifies additional variables that can be removed from a formula, to reduce the search space, by equating pairs of literals, as in \( x = y \) or \( x = -y \).

Backbones are often computed by iterating with a SAT solver. In [17], the authors describe and evaluate several such algorithms and present an improved algorithm. This algorithm involves exactly one unsatisfiable call to the sat solver and at most \( n - b \) satisfiable calls, where \( n \) is the number of variables in \( \varphi \) and \( b \) the size of its backbone.

It is straightforward to apply an algorithm that computes the backbone of a CNF, \( \varphi \), to perform CEP (to detect also equations between literals). Enumerating the variables of \( \varphi \) as \( \{ x₁, \ldots, xₙ \} \). One simply defines

\[
\varphi' = \varphi \land \{ e_{ij} \leftrightarrow (xᵢ \leftrightarrow xⱼ) \mid 0 \leq i < j \leq n \}
\]

introducing $\theta(n^2)$ fresh variables $e_{ij}$. If $e_{ij}$ is in the backbone of $\varphi'$ then $x_i = x_j$ is implied by $\varphi$, and if $\neg e_{ij}$ is in the backbone then $x_i = \neg x_j$ is implied. A major obstacle is that computing the backbone of $\varphi$ is at least as hard as testing for the satisfiability of $\varphi$ itself. Hence, for BEE, the importance of the assumption that $\varphi$ is only a small fragment of the CNF of interest. Another obstacle is that the application of CEP for $\varphi$ with $n$ variables involves computing the backbone for $\varphi'$ which has $\theta(n^2)$ variables.

The CEP algorithm applied in BEE is basically the same as that proposed for computing backbones in [17] extending $\varphi$ to $\varphi'$ as prescribed by Equation (1). We prove that iterated SAT solving for CEP using $\varphi'$ involves at most $n + 1$ satisfiable SAT tests, and exactly one unsatisfiable test, in spite of the fact that $\varphi'$ involves a quadratic number of fresh variables.

We first describe the algorithm applied to compute the backbone of a given formula $\varphi$, which we assume is satisfiable. The algorithm maintains a table indicating for each variable $x$ in $\varphi$ for which values of $x$, $\varphi$ can be satisfied: true, false, or both. The algorithm is initialized by calling the SAT solver with $\varphi_0 = \varphi$ and initializing the table with the information relevant to each variable: if the solution for $\varphi_0$ assigns a value to $x$ then that value is tabled for $x$. If it assigns no value to $x$ then both values are tabled for $x$.

The algorithm iterates incrementally. For each step $i > 0$ we add a single clause $C_i$ (detailed below) and reinvoke the same solver instance, maintaining the learned data of the solver. This process terminates with a single unsatisfiable invocation. In words: the clause $C_i$ can be seen as asking the solver if it is possible to flip the value for any of the variables for which we have so far seen only a single value. More formally, at each step of the algorithm, $C_i$ is defined as follows: for each variable $x$, if the table indicates a single value $v$ for $x$ then $C_i$ includes $\neg v$. Otherwise, if the table indicates two values for $x$ then there is no corresponding literal in $C_i$. The SAT solver is then called with $\varphi_i = \varphi_{i-1} \land C_i$. If this call is satisfiable then the table is updated to record new values for variables (there must be at least one new value in the table) and we iterate. Otherwise, the algorithm terminates and the variables remaining with single entries in the table are the backbone of $\varphi$.

Example 1. Figure 3(a) where we assume given a formula, $\varphi$, which has models as indicated below illustrates the backbone algorithm. The first two iterations of the algorithm provide the models, $\theta_1$ and $\theta_2$. The next iteration illustrates that $\varphi$ has no model which satisfies $\varphi$ and flips...
the values of $x_1$ (to false) or of $x_3$ (to true). We conclude that $x_1$ and $x_3$ are the backbone variables of $\varphi$.

Now consider the case where in addition to the backbone we wish to derive also equations between literals which hold in all models of $\varphi$. The CEP algorithm for $\varphi$ is as follows: (1) enhance $\varphi$ to $\varphi'$ as specified in Equation 1 (and 2) apply backbone computation to $\varphi'$. If $\varphi' \models e_{xy}$ then $\varphi \models x = y$ and if $\varphi' \models \neg e_{xy}$ then $\varphi \models x = \neg y$. As an optimization, it is possible to focus in the first two iterations only on the variables of $\varphi$.

Example 2. Consider the same formula $\varphi$ as in Example 1. This time, in the third iteration we ask to either flip the value for one of $\{x_1, x_3\}$ or for one of $\{e_{13}, e_{24}, e_{25}, e_{45}\}$ and there is such a model, $\theta_3$. This is illustrated as Figure 3 (c).

Theorem 1. Let $\varphi$ be a CNF, $X$ a set of $n$ variables, and $\Theta = \{\theta_1, \ldots, \theta_m\}$ the sequence of assignments encountered by the CEP algorithm for $\varphi$ and $X$. Then, $m \leq n + 1$.

Before presenting a proof of Theorem 1, we introduce some terminology. Assume a set of Boolean variables $X$ and a sequence $\Theta = \{\theta_1, \ldots, \theta_m\}$ of models. Denote $\hat{X} = X \cup \{1\}$ and let $x, y \in \hat{X}$. If $\theta(x) = \theta(y)$ for all $\theta \in \Theta$ or if $\theta(x) \neq \theta(y)$ for all $\theta \in \Theta$, then we say that $\Theta$ determines the equation $x = y$. Otherwise, we say that $\Theta$ disqualifies $x = y$, intuitively meaning that $\Theta$ disqualifies $x = y$ from being determined. More formally, $\Theta$ determines $x = y$ if and only if $\models (x = y)$ or $\models (x = \neg y)$, and otherwise $\Theta$ disqualifies $x = y$.

The CEP algorithm for a formula $\varphi$ and set of $n$ variables $X$ applies so that each iteration results in a satisfying assignment for $\varphi$ which disqualifies at least one additional equation between elements of $\hat{X}$. Although there are a quadratic number of equations to be considered, we prove that the CEP algorithm terminates after at most $n + 1$ iterations.

Proof. (of Theorem 1) For each value $i \leq m$, $\Theta_i = \{\theta_1, \ldots, \theta_i\}$ induces a partitioning, $\Pi_i$ of $\hat{X}$ to disjoint and non-empty sets, defined such that for each $x, y \in \hat{X}$, $x$ and $y$ are in the same partition $P \in \Pi_i$ if and only if $\Theta_i$ determines the equation $x = y$. So, if $x, y \in P \in \Pi_i$ then the equation $x = y$ takes the same value in all assignments of $\Theta_i$. The partitioning is well defined because if in all assignments of $\Theta_i$ both $x = y$ takes the same value and $y = z$ takes the same value, then clearly also $x = z$ takes the same value, implying that $x, y, z$ are in the same partition of $\Pi_i$. Finally, note that each iteration $1 < i \leq m$ of the CEP algorithm disqualifies at least one equation $x = y$ that was determined by $\Theta_i - 1$. This implies that at least one partition of $\Pi_i - 1$ is split into two smaller (non-empty) partitions of $\Pi_i$. As there are a total of $n + 1$ elements in $\hat{X}$, there can be at most $n + 1$ iterations to the algorithm.

Example 3. Consider the same formula $\varphi$ as in Examples 1 and 2. Figure 3 (b) illustrates the run of the algorithm in terms of the partitioning $\Pi$ from the proof of Theorem 1.

We illustrate the impact of CEP with an application where the goal is to find the largest number of edges in a simple graph with $n$ nodes such that any cycle (length) is larger than 4. The graph is represented as a Boolean adjacency matrix $A$ and there are two types of constraints: (1) constraints about cycles in the graph: $\forall_{i \neq j,k}. A[i,j] + A[j,k] + A[k,i] < 3$, and $\forall_{i \neq j,k,l}. A[i,j] + A[j,k] + A[k,l] + A[l,i] < 4$; and (2) constraints about symmetries: in addition to the obvious $\forall_{i \neq j \leq n}. (A[i,j] \equiv A[j,i])$ and $A[i,i] \equiv false$, we constrain the rows of the adjacency matrix to be sorted lexicographically (justified in [7]), and we impose lower and upper bounds on the degrees of the graph nodes as described in [12].

Table 2 illustrates results, running BEE with and without CEP. Here, we focus on finding a graph with the prescribed number of graph nodes with the known maximal number of edges (all
<table>
<thead>
<tr>
<th></th>
<th>with CEP</th>
<th>without CEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes edges</td>
<td>comp.</td>
<td>clauses</td>
</tr>
<tr>
<td>15 26</td>
<td>0.24</td>
<td>13421</td>
</tr>
<tr>
<td>16 28</td>
<td>0.26</td>
<td>18339</td>
</tr>
<tr>
<td>17 31</td>
<td>0.39</td>
<td>21495</td>
</tr>
<tr>
<td>18 34</td>
<td>0.49</td>
<td>26765</td>
</tr>
<tr>
<td>19 38</td>
<td>0.46</td>
<td>30626</td>
</tr>
<tr>
<td>20 41</td>
<td>0.55</td>
<td>43336</td>
</tr>
<tr>
<td>21 44</td>
<td>0.77</td>
<td>52187</td>
</tr>
<tr>
<td>22 47</td>
<td>0.88</td>
<td>61611</td>
</tr>
<tr>
<td>23 50</td>
<td>1.10</td>
<td>73147</td>
</tr>
<tr>
<td>24 54</td>
<td>2.02</td>
<td>81634</td>
</tr>
<tr>
<td>25 57</td>
<td>1.40</td>
<td>99027</td>
</tr>
<tr>
<td>26 61</td>
<td>4.58</td>
<td>110240</td>
</tr>
<tr>
<td>27 65</td>
<td>2.16</td>
<td>127230</td>
</tr>
</tbody>
</table>

Table 2: Search for graphs with no cycles of size 4 or less (comp. & solve times in sec.)

instances are satisfiable), and CEP is applied to the set of clauses derived from the symmetry constraints (2) detailed above. The table indicates the number of nodes, and for each CEP choice: the BEE compilation time, the number of clauses and variables, and the subsequent sat solving time. The table indicates that CEP increases the compilation time (within reason), reduces the CNF size (considerably), and (for the most) improves SAT solving time.²

5 Enhancing BEE for Binary Number Representation

This section describes an extension of BEE to support binary numbers. A naive extension is straightforward. There is a wide body of research specifying the bit-blasting of finite domain constraints for binary arithmetic. So, that is not the topic of this section. The interesting aspect of this exercise is how to obtain the constraint encodings together with support for equi-propagation on their bit representations. In the presentation we refer to the current version of BEE as the unary core, and to the extension for binary numbers as the binary extension. There are several possible approaches to define the binary extension:

1. **CEP**: A straightforward approach is to specify standard encodings for each of the new constraints in the binary extension and then to flag each of them (individually) as candidates for complete equi-propagation. In this way, as described in the previous section, BEE will infer at compile time all equi-propagations and perform the corresponding simplifications. However, the implementation of CEP involves calling a SAT solver and its application should be limited.

2. **Ad-hoc rules**: Another option is to introduce ad-hoc equi-propagation rules for each binary constraint similar to those already in BEE for the unary constraints (recall the example of Figure 1). However, besides being tedious, for the constraints of the binary extension there are very few relevant ad-hoc rules.

3. **Decomposition to the unary kernel**: In this approach we design encodings for binary constraints in terms of decompositions to unary constraints for which equi-propagation rules

---

²Experiments are performed on a single core of an Intel(R) Core(TM) i5-2400 3.10GHz CPU with 4GB memory under Linux (Ubuntu lucid, kernel 2.6.32-24-generic).
are already defined. For example, encoding the multiplication of two \( n \)-bit binary numbers decomposes to involve unary sums of at most \( 2n \) bits each. The unary core then performs equi-propagation on the decomposed constraints.

We describe encodings using the third approach for two constraints on binary representations: \texttt{binary\_array\_sum\_eq} and \texttt{binary\_times}. We also consider the special case where multiplication is applied to specify that \( Z = X^2 \) and demonstrate ad-hoc rules for that case.

**Summing:** Consider a constraint \texttt{binary\_array\_sum\_eq(As,Sum)} where \( As \) is an array of binary numbers and \( Sum \) is the binary number representing their sum. In this context, we view \( As \) as a binary matrix. The rows correspond to binary numbers, and the columns to so-called buckets which are sets of bits with the same “weight” or position. The number of rows is typically not large so that it is reasonable to sum the columns using unary arithmetic. In this way the decomposition of the constraint on binary numbers relies on the underlying unary core of BEE. Assume that \( As \) consists of more than a single number, otherwise the decomposition is trivial. The decomposition proceeds as follows: (1) apply \texttt{transpose(As,Bs)} which transposes the binary numbers in \( As \) to a bucket representation \( Bs \) (assume least significant bucket first). (2) introduce unary-core constraints \texttt{bool\_array\_sum\_eq(Bi,Ui)} which sum the buckets to an array \( Us \) of unary numbers. (3) the recursively defined \texttt{buckets2binary([U|Us],C,[S|Sum])} finishes the task and is defined as follows.

\[
\text{buckets2binary([U|Us],C,[B|Sum])} \xrightarrow{\text{decompose}} \text{int\_plus(U,C,U'),}\quad \text{int\_div(U',2,C'),}\quad \text{int\_mod(U',2,B)},\quad \text{buckets2binary(Us,C',Sum)}
\]

Here: \( U \) is the least significant (unary) bucket, \( C \) is a carry variable (unary integer, initially 0), and \( B \) is the least significant bit of the (binary) sum. When, eventually, the buckets are exhausted, decomposition proceeds as follows.

\[
\text{buckets2binary([],C,[B|Sum])} \xrightarrow{\text{decompose}} \text{int\_div(C,2,C'),}\quad \text{int\_mod(C,2,B)},\quad \text{buckets2binary([],C',Sum)}
\]

Observe that, if applied without any buckets, \texttt{buckets2binary([],Unary,Binary)} defines the channeling between unary and binary representations. We also note that for unary numbers, the encoding of division and modulo by 2 are efficient. Division (by 2) simply collects the even positioned bits, and modulo (2) takes advantage of the fact that the representation is “sorted”.

Below we evaluate our proposed encoding in BEE, but first let us introduce the encoding of binary multiplication.

**Multiplying:** Consider a constraint \texttt{binary\_times(A,B,C)} specifying that \( C = A \times B \). It is implemented in BEE as follows. Assume that \( A = [A_0 \ldots A_n] \) and \( B = [B_0 \ldots B_n] \) are the binary representations of \( A \) and \( B \). Decomposition for this constraint introduces clauses defining

\[
\bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq m} Z_{ij} \leftrightarrow A_i \land B_j
\]  

(2)

and an additional constraint \texttt{binary\_array\_sum\_eq([Z_1 \ldots Z_n],C)} where for \( 1 \leq j \leq m \), \( Z_j \) is the binary number with bits

\[
Z_{nj} \ldots Z_{ij} 0 \ldots 0
\]

\[ j - 1 \]
redundant variables. The result of this is illustrated in Figure 4(b). In Figure 4(c) we reorder

\[\sum_{i=1}^{n} x_i = 1\]

such that each digit value is used between 1 and \([n/3]\) times. Table 3 depicts experimental

Results comparing two encodings of binary_array_sum_eq(As, Sum) and binary_times(A, B, C), we consider the application of BEE to model and solve the n-fractions problem, also known as CSPLIB 041.3 Here, one should find digit values \((1 - 9)\) for the variables in

\[\text{Squaring: } \text{Consider the special case of multiplication binary_times(A, A, C) specifying that } A^2 = C \text{ where we introduce two additional optimizations. First, consider the variables } Z_{ij} \text{ introduced in Equation } [2], \text{ we have } Z_{ij} = Z_{ji} \text{ and hence equi-propagation applies to remove the redundant variables. The result of this is illustrated in Figure } 4(b). \text{ In Figure } 4(c) \text{ we reorder}\]

---

Table 3: Comparison of encodings for the $n$-fractions problem (comp. and sat times in sec. with 4 hour timeout marked as $\infty$)

<table>
<thead>
<tr>
<th>$n$</th>
<th>comp. clauses vars sat</th>
<th>comp. clauses vars sat</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.05 25492 4354 2.72</td>
<td>0.26 23793 4556 1.39</td>
</tr>
<tr>
<td>4</td>
<td>0.13 56125 9556 11.19</td>
<td>0.50 47743 9078 0.56</td>
</tr>
<tr>
<td>5</td>
<td>0.23 98712 16551 59.4</td>
<td>0.77 78607 14703 55.65</td>
</tr>
<tr>
<td>6</td>
<td>0.38 164908 27283 844.91</td>
<td>1.01 118850 21977 5.13</td>
</tr>
<tr>
<td>7</td>
<td>0.76 247082 \infty \infty</td>
<td>1.87 164451 30125 36.83</td>
</tr>
<tr>
<td>8</td>
<td>1.29 363323 59183 \infty</td>
<td>2.14 221262 40196 2653.68</td>
</tr>
</tbody>
</table>

the bits in the columns, as if, letting the bits drop down to the baseline. Second, consider the “columns” in the binary_array_sum_eq([Z1,...,Zn],C) constraint. Each variable of the form $Z_{ij}$ with $i \neq j$ in a column occurs twice. So, both can be removed and one inserted back in the column to the left. This is illustrated in Figure 4(d) where we highlight the move of the two $z_{02}$ instances. These optimizations reduce the size of the CNF and are applied both in the binary and in the unary encodings.

To evaluate the encoding of binary_times(A,B,C) for the special case when $A = B$, we consider the application of BEE to model and solve the number partitioning problem, also known as CSPLIB 049.4 Here, one should finding a partition of numbers $\{1, \ldots, n\}$ into two sets $A$ and $B$ such that: $A$ and $B$ have the same cardinality, the sum of numbers in $A$ equals the sum of numbers in $B$, and the sum of the squares of the numbers in $A$ equals the sum of the squares of the numbers in $B$.

Figure 5 depicts our results. We consider four settings. The first two are the binary and unary approaches described above where buckets of bits of the same binary weight are summed using full adders or sorting networks respectively. In the other two settings, we apply complete equi-propagation per individual constraint (on binary numbers), on top of the ad-hoc rules implemented in BEE. Figure 5(a) illustrates the size of the encodings (number of CNF variables) for each of the four settings in terms of the instance size. The two top curves coincide and correspond to the unary encodings which create slightly larger CNFs. However note that the unary core of BEE with its ad-hoc (and more efficient) implementation of equi-propagation, detects all of the available equi-propagation. There is no need to apply CEP. The bottom two curves correspond to the binary encodings and illustrate that CEP detects further optimizations beyond what is detected using BEE.

Figure 5(b) details the SAT solving times. Here we ignore the compilation times (which are high when using CEP) and focus on the quality of the obtained CNF. The graph indicates a clear advantage to the unary approach (where CEP is not even required). The average solving time using the unary approach (without CEP) is 270 (sec) vs 1503 (sec) using the binary approach (with CEP). This is in spite of the fact that unary approach involves larger CNF sizes.

Figures 5(c) and (d) further detail the effect of CEP in the binary and unary encodings depicting the numbers of clauses and of variables reduced by CEP in both techniques. The smaller this number, the more equi-propagation performed ad-hoc by BEE. In both graphs the lower curve corresponds to the encodings based on the unary core indicating that this is the one of better quality. See footnote 2 for details on machine.

---

6 Conclusion

We have detailed two features of BEE not described in previous publications. These concern the hybrid approach to encode cardinality constraints and the procedure for applying complete equi-propagation. We have also described our approach to enhance the unary kernel of BEE for binary numbers. Our approach is to rely as much as possible on the implementation of equi-propagation on unary numbers to build the task of equi-propagation for binary numbers. We have illustrated the power of this approach when encoding binary number multiplication. The extension of BEE for binary numbers is ongoing and still requires a thorough experimentation to evaluate its design.
References


