Compiling Finite Domain Constraints to SAT with BEE

Michael Codish

Department of Computer Science
Ben Gurion University
Beer-Sheva, Israel

Joint work: Yoav Fekete & Amit Metodi

In Collaboration with: Vitaly Lagoon & Peter Stuckey
It is all about: Solving hard problems via SAT encodings

Problem (hard) \rightarrow \text{encoding} \rightarrow \text{CNF} \rightarrow \text{sat solving} \rightarrow \text{SAT 'ing Assignment}

Solution \downarrow \text{direct} \downarrow \text{decoding}

\text{hype!}
was born (*) with two objectives:

- Facilitate the (user) process of **encoding** a (constraint) problem to CNF
- **Compile** constraint models to CNF while applying optimizations in order to generate (usually) smaller and better CNF formulas.

(*) Amit Metodi, Michael Codish, Vitaly Lagoon, Peter J. Stuckey: Boolean Equi-propagation for Optimized SAT Encoding. **CP 2011**: 621-636
Problem (hard) \rightarrow \text{modeling} \rightarrow \text{Constraint Model} \rightarrow \text{Encoding} \rightarrow \text{CNF}

\text{hype!} \rightarrow \text{direct} \rightarrow \text{Solution} \leftarrow \text{decoding} \rightarrow \text{SAT 'ing Assignment}

The language \rightarrow \text{The compiler}
Outline

• Introduction
• BEE in a nutshell
  • Order encoding (representing integers)
  • Equi-propagation (ad-hoc)
• The “new” stuff
  • Complete Equi-Propagation
  • Cardinality Constraints in BEE
  • The binary extension of BEE
Problem (hard) → modeling → Constraint Model → Encoding → CNF

The language
The compiler
Example: encoding Sudoku

Problem (hard)

Constraint Model

new_int(X_{1,1}, 1, 9)

\vdots

new_int(X_{9,9}, 1, 9)

\text{allDiff}([X_{1,1}, \ldots, X_{1,9}])

\vdots

\text{int_eq}(X_{1,1}, 5)

\text{int_eq}(X_{1,2}, 3)

\vdots

encoding

CNF
Encoding

modeling

Problem
(hard)

Constraint
Model

Encoding

CNF

The language

The compiler
The Usual Approach

Constraint Model

\[ C_1 \]
\[ C_2 \]
\[ C_3 \]
\[ C_n \]

encode

encode

encode

encode

CNF

simplification

Tools such as: SatELite, ReVivAl
Based on Unit Propagation and Resolution.
The Usual Approach

C1, C2, C3, ..., Cn: Constraint Model

- Encode
- CNF
- Simplification

Problems:
- Word vrs bit level
- Large CNF
The CNF you want to optimize did not fall out of the sky

Optimize it while generating it

Let the constraint model drive the CNF optimization
The BEE Approach

Constraint Model

C1, C2, C3, \ldots, Cn

encode

CNF

encode

CNF

encode

CNF

encode

CNF

propagation

simplification
The BEE Approach

Constraint Model

C1

encode
update

CNF

C2

encode
update

CNF

C3

encode
update

CNF

Cn

encode
update

CNF

simplification

A=1, D= -E

bool_array_sum_eq([ A,B,C,D,E,F,G], 3)

bool_array_sum_eq([ 1,B,C,-E,E,F,G], 3)

bool_array_sum_eq([ B,C,F,G], 1)
Equi-propagation is the process of inferring equations implied by a “small chunk” of constraints.

1. View each “single” constraint as a Boolean formula.
2. Derive (“all”) implied equalities between literals and constants.
3. Apply them to simplify all constraints.

Equi-propagation is more powerful reasoning but on smaller CNF portions of the form X=L where L is a constant or a literal: X=Y, X= -Y, X=0, X=1.
Representing numbers

Order encoding (unary)

\[ X = [x_1, \ldots, x_i, \ldots, x_n] \]

\[ x_i \leftrightarrow (X \geq i) \]

\((X = 3) = [1,1,1,0,0]\)
Lots of equi-propagation

1. $X \geq i \quad i$

2. $X \neq i \Leftrightarrow u = v$

3. $[x_1, x_2, x_3] + [y_1, y_2, y_3] = 3$

The Encoding to SAT needs NO Clauses. It is obtained by unification:

- $x_1 = -y_3$
- $x_2 = -y_2$
- $x_3 = -y_1$
TWO DESIGN CHOICES

Implementing Equi-Propagation

1. Using BDD’s.
   • Prohibitive for global constraints.
   • Complete

2. Using SAT (on small groups of constraints)
   • In practice, surprisingly, “not slow”
   • Complete

3. Ad-Hoc rules (per constraint type)
   • Fast, precise in practice
   • Incomplete
Ad-Hoc Rules: \texttt{int\_plus}

- **Equi-Propagation**

  \begin{align*}
  c &= \texttt{int\_plus}(X, Y, Z) \text{ where } X = \langle x_1, \ldots, x_n \rangle, \quad Y = \langle y_1, \ldots, y_m \rangle, \text{ and } Z = \langle z_1, \ldots, z_{n+m} \rangle \\
  \text{if in } E &\quad \text{then add in } \mu_c(E) \\
  \begin{array}{ll}
  X \geq i, Y \geq j & Z \geq i + j \\
  X < i, Y < j & Z < i + j - 1 \\
  Z \geq k, X < i & Y \geq k - i \\
  Z < k, X \geq i & Y < k - i \\
  X = i & z_{i+1} = y_1, \ldots, z_{i+m} = y_m \\
  Z = k & x_1 = -y_k, \ldots, x_k = -y_1 \\
  \end{array}
  \end{align*}

- **Partial Evaluation**

  \begin{align*}
  c &= \texttt{int\_plus}(X, Y, Z) \text{ where } X = \langle x_1, \ldots, x_n \rangle, \quad Y = \langle y_1, \ldots, y_m \rangle, \text{ and } Z = \langle z_1, \ldots, z_{n+m} \rangle \\
  \text{if } &\quad \text{then replace with} \\
  \begin{array}{ll}
  X = i & \text{true} \\
  Z = k & \text{true} \\
  X \geq i, Z \geq i & \texttt{int\_plus}([x_{i+1}, \ldots, x_n], Y, [z_{i+1}, \ldots, z_{n+m}]) \\
  X \leq i, Z \leq i + m & \texttt{int\_plus}([x_1, \ldots, x_i], Y, [z_1, \ldots, z_{i+m}]) \\
  \end{array}
  \end{align*}
Introduction

BEE in a nutshell

http://amit.metodi.me/research/bee/

The “new” stuff

- Complete Equi-Propagation
- Cardinality Constraints in BEE
- The binary extension of BEE
designate specific sets of constraints for complete equi-propagation (using a SAT solver)
### Example: Kakuro

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>19</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>I₁</td>
<td>I₂</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>I₆</th>
<th>I₇</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>new_int(I₁, 1, 9)</th>
<th>int_array_plus([I₁, I₂], 13)</th>
<th>allDiff([I₁, I₂])</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>new_int(I₂, 1, 9)</td>
<td>int_array_plus([I₁, I₃], 5)</td>
<td>allDiff([I₁, I₃])</td>
</tr>
<tr>
<td></td>
<td>new_int(I₃, 1, 9)</td>
<td>int_array_plus([I₃, I₄, I₅], 12)</td>
<td>allDiff([I₃, I₄, I₅])</td>
</tr>
<tr>
<td></td>
<td>new_int(I₄, 1, 9)</td>
<td>int_array_plus([I₂, I₄, I₆], 19)</td>
<td>allDiff([I₂, I₄, I₆])</td>
</tr>
<tr>
<td></td>
<td>new_int(I₅, 1, 9)</td>
<td>int_array_plus([I₆, I₇], 3)</td>
<td>allDiff([I₆, I₇])</td>
</tr>
<tr>
<td></td>
<td>new_int(I₆, 1, 9)</td>
<td>int_array_plus([I₅, I₇], 4)</td>
<td>allDiff([I₅, I₇])</td>
</tr>
<tr>
<td></td>
<td>new_int(I₇, 1, 9)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Example: Kakuro

![Kakuro Diagram]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>I₃</td>
<td>I₄</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I₅</td>
<td>I₆</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>I₇</td>
<td></td>
</tr>
</tbody>
</table>

### CEP

<table>
<thead>
<tr>
<th>new_int(I₁, 1, 9)</th>
<th>int_array_plus([I₁, I₂], 13)</th>
<th>allDiff([I₁, I₂])</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_int(I₂, 1, 9)</td>
<td>int_array_plus([I₁, I₃], 5)</td>
<td>allDiff([I₁, I₃])</td>
</tr>
<tr>
<td>new_int(I₃, 1, 9)</td>
<td>int_array_plus([I₃, I₄, I₅], 12)</td>
<td>allDiff([I₃, I₄, I₅])</td>
</tr>
<tr>
<td>new_int(I₄, 1, 9)</td>
<td>int_array_plus([I₂, I₄, I₆], 19)</td>
<td>allDiff([I₂, I₄, I₆])</td>
</tr>
<tr>
<td>new_int(I₅, 1, 9)</td>
<td>int_array_plus([I₆, I₇], 3)</td>
<td>allDiff([I₆, I₇])</td>
</tr>
<tr>
<td>new_int(I₆, 1, 9)</td>
<td>int_array_plus([I₅, I₇], 4)</td>
<td>allDiff([I₅, I₇])</td>
</tr>
<tr>
<td>new_int(I₇, 1, 9)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CEP is similar to Backbones

Backbones are about detecting variables which take fixed values in all solutions

CEP is also about detecting equations between variables which take fixed values in all solutions

\[ \varphi \models x = 1 \]
\[ \varphi \models x = 0 \]
\[ \varphi \models x = 1 \]
\[ \varphi \models x = 0 \]
\[ \varphi \models x = y \]
\[ \varphi \models x = -y \]
Backbones using SAT

Assume $\varphi$ with $n=5$ variables

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\theta_1$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$\theta_2$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\theta_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\varphi_0 = \varphi$
$\varphi_1 = \varphi_0 \land \neg \theta_1$
$\varphi_2 = \varphi_1 \land (\neg x_1 \lor x_3)$

At most $n$ sat calls; Incremental; Only the last call is unsat.

iteration #1: sat($\varphi$)
iteration #2: sat($\varphi$) (& diff)
iteration #3: sat($\varphi$) (and flip at least one that didn't flip yet)
Backbones for Equality (CEP)

Essentially the same; Define

\[ \varphi' = \varphi \land \left\{ e_{ij} \leftrightarrow (x_i \leftrightarrow x_j) \mid 0 \leq i < j \leq n \right\} \]

and then apply a backbone algorithm

But, we have added \( O(n^2) \) new variables (???)
Backbones for Equality (CEP)

iteration #1 and #2: sat(\(\varphi\))
(two different assignments)

\[ \varphi_2 = \varphi_1 \land \left( \neg x_1 \lor x_3 \lor e_{13} \lor \right. \]
\[ \left. \neg e_{24} \lor \neg e_{25} \lor e_{45} \right) \]

iteration #3: sat(\(\varphi\))
(and flip at least one that didn’t flip yet)

\[ \varphi_3 = \varphi_2 \land \left( \neg x_1 \lor x_3 \lor e_{13} \lor e_{45} \right) \]

iteration #4: sat(\(\varphi\))
(and flip at least one that didn’t flip yet)
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, $m \leq n + 1$. 
Outline

• Introduction
• BEE in a nutshell

http://amit.metodi.me/research/bee/

• The “new” stuff
  • Complete Equi-Propagation
  • Cardinality Constraints in BEE
  • The binary extension of BEE
Cardinality Constraints

1. BDD like structure (symbolic)
2. Sorting networks (unary)
3. Network of adders (binary)

Better propagation → Smaller CNF size
Sat encoding - cardinality constraints

\[
\sum x_i \leq 4 \quad \sum x_i \geq 4
\]

\[\varphi \land \neg y_5 \quad \varphi \land y_4\]
Many adapt this approach applying Batcher's Odd Even Sorting Network defined recursively; so it is all in the merger. Another option is Parberry's “pairwise” sorting networks.

The odd-even merger is basically a unary adder and consists of $O(n \log n)$ “comparators”. 
Totalizers (same but with different merger)

Totalizers: define the merger with a direct encoding $O(n^2)$ clauses

\[
\begin{align*}
A \geq i & \land B \geq j \rightarrow C \geq i+j \\
A \leq i & \land B \leq j \rightarrow C \leq i+j \\
i, j
\end{align*}
\]

(direct) adders are larger than mergers but have better propagation properties
Hybrid

(addirect) adders are larger than mergers but have better propagation properties

But, for small n, adders are actually smaller than mergers

Anyway, the size penalty can pay off (if under control)

While constructing, first use mergers. Then, as things get smaller, introduce adders
Experiments illustrating the advantage of the hybrid approach:

Ignasi Abio, Robert Nieuwenhuis, Albert Oliveras, Enric Rodriguez-Carbonell; A parametric approach for smaller and better encodings of cardinality constraints; CP 2013
bSettings.pl (for cardinality constraints)

/*
   Name: 'unaryAdderType'
   Constraint: 'int_plus'
   Possible values:
   'uadder' - (default) use O(N^2) encoding
   'merger' - decompose to comparators O(NlogN) encoding
   'hybrid' - hybrid approach:
      BEE will decide if to decompose like merger or
      encode like uadder - based on the generated CNF size.
*/
:- defineSetting(unaryAdderType,uadder).

/*
   Name: 'sumBitsDecompose'
   Constraint: 'bool_array_sum_op' / 'bool_array_pb_op'
   Possible values:
   'simple' - (default) divide and conquer technique
   'buckets' - split to buckets, sum each bucket
      and use linear constraints to sum buckets
   'pairwise' - pairwise sorting network
*/
:- defineSetting(sumBitsDecompose,simple).
Outline

• Introduction
• BEE in a nutshell

http://amit.metodi.me/research/bee/

• The “new” stuff
  • Complete Equi-Propagation
  • Cardinality Constraints in BEE
  • The binary extension of BEE
Binary Extension of BEE

Bit Blasting is obvious; But it is more about how the simplifications work

Where possible, blast into the unary core
Binary Multiplication

\[
\begin{array}{c}
\times \\
\hline
x_4 & x_3 & x_2 & x_1 & x_0 \\
y_4 & y_3 & y_2 & y_1 & y_0 \\
\hline
z_{04} & z_{03} & z_{02} & z_{01} & z_{00} \\
z_{14} & z_{13} & z_{12} & z_{11} & z_{10} \\
z_{24} & z_{23} & z_{22} & z_{21} & z_{20} \\
z_{34} & z_{33} & z_{32} & z_{31} & z_{30} \\
\hline
z_{44} & z_{43} & z_{42} & z_{41} & z_{40} \\
\end{array}
\]

\[z_{ij} \leftrightarrow x_i \land y_j\]

unary sums
Binary Multiplication (square)

\[
\begin{array}{cccccc}
\times & x_4 & x_3 & x_2 & x_1 & x_0 \\
& x_4 & x_3 & x_2 & x_1 & x_0 \\
\hline
& \_04 & \_03 & \_02 & \_01 & \_00 \\
& \_14 & \_13 & \_12 & \_11 & {\color{red} \_01} \\
& \_24 & \_23 & \_22 & \_12 & \_02 \\
& \_34 & \_33 & \_23 & \_13 & \_03 \\
+ & \_44 & \_34 & \_24 & \_14 & \_04 \\
\end{array}
\]

**equi propagation:** \( \_{ij} = \_{ji} \)
## Binary Multiplication (square)

<table>
<thead>
<tr>
<th></th>
<th>$x_4$</th>
<th>$x_3$</th>
<th>$x_2$</th>
<th>$x_1$</th>
<th>$x_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\times$</td>
<td>$x_4$</td>
<td>$x_3$</td>
<td>$x_2$</td>
<td>$x_1$</td>
<td>$x_0$</td>
</tr>
<tr>
<td>$z_{04}$</td>
<td>$z_{03}$</td>
<td>$z_{02}$</td>
<td>$z_{01}$</td>
<td>$z_{00}$</td>
<td></td>
</tr>
<tr>
<td>$z_{14}$</td>
<td>$z_{13}$</td>
<td>$z_{12}$</td>
<td>$z_{11}$</td>
<td>$z_{10}$</td>
<td></td>
</tr>
<tr>
<td>$z_{24}$</td>
<td>$z_{23}$</td>
<td>$z_{22}$</td>
<td>$z_{21}$</td>
<td>$z_{20}$</td>
<td></td>
</tr>
<tr>
<td>$z_{34}$</td>
<td>$z_{33}$</td>
<td>$z_{32}$</td>
<td>$z_{31}$</td>
<td>$z_{30}$</td>
<td></td>
</tr>
<tr>
<td>$z_{44}$</td>
<td>$z_{43}$</td>
<td>$z_{42}$</td>
<td>$z_{41}$</td>
<td>$z_{40}$</td>
<td></td>
</tr>
</tbody>
</table>

$z_{04}$ $z_{03}$ $z_{02}$ $z_{01}$ $z_{00}$
$z_{14}$ $z_{13}$ $z_{12}$ $z_{11}$ $z_{10}$
$z_{24}$ $z_{23}$ $z_{22}$ $z_{21}$ $z_{20}$
$z_{34}$ $z_{33}$ $z_{32}$ $z_{31}$ $z_{30}$
$z_{44}$ $z_{43}$ $z_{42}$ $z_{41}$ $z_{40}$

$z_{11}$ $z_{00}$

$z_{23}$ $z_{12}$
$z_{34}$ $z_{14}$ $z_{13}$ $z_{03}$
$z_{44}$ $z_{24}$ $z_{33}$ $z_{04}$ $z_{22}$ $z_{02}$

$z_{11}$ $0$ $z_{00}$
Conclusion

The "new" stuff

- Complete Equi-Propagation
- Cardinality Constraints in BEE
- The binary extension of BEE