Compiling Finite Domain Constraints to SAT with BEE[∗]

AMIT METODI MICHAEL CODISH
Department of Computer Science, Ben-Gurion University, Israel

submitted 25 March 2012; revised 11 June 2012; accepted 18 June 2012

Abstract

We present BEE, a compiler which enables to encode finite domain constraint problems to CNF. Using BEE both eases the encoding process for the user and also performs transformations to simplify constraints and optimize their encoding to CNF. These optimizations are based primarily on equi-propagation and on partial evaluation, and also on the idea that a given constraint may have various possible CNF encodings. Often, the better encoding choice is made after constraint simplification. BEE is written in Prolog and integrates directly with a SAT solver through a suitable Prolog interface. We demonstrate that constraint simplification is often highly beneficial when solving hard finite domain constraint problems. A BEE implementation is available with this paper.

KEYWORDS: SAT encoding, FD constraints, Equi-propagation, partial evaluation.

1 Introduction

In recent years, Boolean SAT solving techniques have improved dramatically. Today's SAT solvers are considerably faster and able to manage larger instances than yesterday's. Moreover, encoding and modeling techniques are better understood and increasingly innovative. SAT is currently applied to solve a wide variety of hard and practical combinatorial problems, often outperforming dedicated algorithms. The general idea is to encode a (typically, NP) hard problem instance, μ, to a Boolean formula, φ_μ, such that the solutions of μ correspond to the satisfying assignments of φ_μ. Given the encoding, a SAT solver is then applied to solve μ.

Tailgating the success of SAT technology are a variety of tools which can be applied to specify and then compile problem instances to corresponding SAT instances. For example, Cadoli and Schaerf (2005) introduce NP-SPEC, a logic-based specification language which allows to specify combinatorial problems in a declarative way. At the core of this system is a compiler which translates specifications to CNF formula. The general objective of such tools is to facilitate the process of providing high-level descriptions of how the (constraint) problem at hand is to be solved. Typically, a constraint based modeling language is introduced and used to model instances. Drawing on the analogy to programming languages, given such a

∗ Partially supported by the Lynn and William Frankel Center for Computer Sciences.

description, a compiler then provides a low-level executable for the underlying machine. Namely, in our context, a formula for the underlying SAT or SMT solver. One obstacle when seeking to optimize CNF encodings derived from high-level descriptions, is that CNF encodings are *"bit-level"* representations and do not maintain *"word-level"* information. For example, from a CNF encoding one cannot know that certain bits originate from the same integer value in the original constraint. This limits the ability to apply optimizations which rely on such word-level information.

We mention two relevant tools. Sugar (Tamura et al. 2009), is a SAT-based constraint solver. To solve a finite domain linear constraint satisfaction problem it is first encoded to a CNF formula by Sugar, and then solved using the MiniSat solver (Eén and Sörensson 2003). BEE is like Sugar, but applies optimizations. Sugar is the first system which demonstrates the advantage in adopting the, so-called, unary order-encoding to represent integers. We follow suite, and introduce additional novel encoding techniques that take advantage of, previously unobserved, properties of the order-encoding. MiniZinc (Nethercote et al. 2007), is a constraint modeling language which is compiled by a variety of solvers to the low-level target language FlatZinc for which there exist many solvers. It creates a standard for the source language (which we follow loosely). BEE is like FlatZinc, but with a focus on a subset of the language relevant for finite domain constraint problems.

We present a tool, BEE (**B**en-Gurion University **E**qui-propagation **E**ncoder) which translates models in a constraint based modeling language, similar to Sugar and FlatZinc, to CNF. Conceptually, BEE maintains two representations for each constraint in a model so that each constraint is also viewed as a Boolean function. Partial evaluation, and other word-level techniques, drive simplification through the constraint part; whereas, equi-propagation (Metodi et al. 2011), and other bit-level techniques, drive simplification through the Boolean part. Finally, an encoding technique is selected for a constraint, depending on its context, to derive a CNF.

The name, "BEE" refers both to the constraint language as well as to its compiler to CNF. BEE is not a constraint solver, but can be applied in combination with a SAT solver to solve finite domain constraint problems. We report on our experience with applications which indicates that using BEE, like any compiler, has two main advantages. On the one hand, it facilitates the process of programming (or modeling). On the other hand, given a program (a model), it simplifies the corresponding CNF which, in many cases, is faster to solve than with other approaches. The tool integrates with SWI Prolog and can be downloaded from (Metodi 2012).

2 Representing Integers

A fundamental design choice when encoding finite domain constraints concerns the representation of integer variables. Gavanelli (2007) surveys several of the possible choices (the *direct-*, *support-* and *log- encodings*) and introduces the *log-support encoding.* We focus in this paper on the use of unary representations and primarily on the, so-called, *order-encoding* (see e.g. (Crawford and Baker 1994; Bailleux and Boufkhad 2003)) which has many nice properties when applied to small finite domains. We describe the setting where all integer variables are represented in the order-encoding except for those involved in a global "all-different" constraint which take a dual representation with channeling between the order-encoding and the *direct encoding*. This choice derives from the observation by Ansótegui et al. (2004) that the direct-encoding is superior when encoding the all-different constraint.

Let bit vector $X = [x_1, \ldots, x_n]$ represent a finite domain integer variable. In the *order-encoding*, X constitutes a monotonic non-increasing Boolean sequence. Bit x_i is interpreted as $X \geq i$. For example, the value 3 in the interval $[0, 5]$ is represented in 5 bits as $[1, 1, 1, 0, 0]$. In the *direct-encoding*, X constitutes a characteristic function (exactly one bit takes value 1) and x_i is interpreted as stating $X = i - 1$. For example, the value 3 in the interval $[0, 5]$ is represented in 6 bits as $[0, 0, 0, 1, 0, 0]$.

An important property of a Boolean representation for finite domain integers is the ability to represent changes in the set of values a variable can take. 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 $x_{b+1} = 0$ (if $b < n$). The variables $x_{a'}$ and $x_{b'}$ for $0 \geq a' > a$ 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 $dom(X) = \{3, 4, 5\}$. This property is exploited in Sugar (Tamura et al. 2009) which also applies the order-encoding.

We observe, and apply in BEE, an additional property of the order-encoding: 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, $X = [x_1, \underline{x_2, x_2}, x_4, \underline{x_5, x_5}, \underline{x_7, x_7}, x_9]$, signifying that $dom(X) = \{0, 1, 3, 4, 6, 8, 9\}$.

The order-encoding has many additional nice features that are exploited in BEE to simplify constraints and their encodings to CNF. To illustrate one, consider a constraint of the form $\mathtt{A} + \mathtt{B} = 5$ where A and B are integer values in the range between 0 and 5 represented in the order-encoding. At the bit level we have: $\mathtt{A} = [\mathtt{a}_1, \ldots, \mathtt{a}_5]$ and $\mathtt{B} = [\mathtt{b}_1, \ldots, \mathtt{b}_5]$. The constraint is satisfied precisely when $\mathtt{B} = [\neg\mathtt{a}_5, \ldots, \neg\mathtt{a}_1]$. Instead of encoding the constraint to CNF, we substitute the bits $\mathtt{b}_1, \ldots, \mathtt{b}_5$ by the literals $\neg\mathtt{a}_5, \ldots, \neg\mathtt{a}_1$, and remove the constraint. In Prolog, this is implemented as a unification and does not generate any clauses in the encoding.

3 Constraints in BEE

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, which may be instantiated when simplifying constraints. Table 1 introduces the syntax for (a simplified subset of) BEE. In the table, X and Xs (possibly with subscripts) denote a literal (a Boolean variable or its negation) and a vector of literals, I (possibly with subscript) denotes an integer

Declaring Variables		
(1)	$\mathtt{new\_bool(X)}$	declare Boolean $\mathtt{X}$
(2)	$\mathtt{new\_int}(\mathtt{I}, \mathtt{c}_1, \mathtt{c}_2)$	declare integer $\mathtt{I}$, $\mathtt{c}_1 \leq \mathtt{I} \leq \mathtt{c}_2$
(3)	$\mathtt{ordered}([\mathtt{X}_1, \ldots, \mathtt{X}_n])$	$\mathtt{X}_1 \geq \mathtt{X}_2 \geq \cdots \geq \mathtt{X}_n$ (on Booleans)
Boolean (reified) Statements		$\mathtt{op} \in \{\mathtt{or}, \mathtt{and}, \mathtt{xor}, \mathtt{iff}\}$
(4)	$\mathtt{bool\_eq}(\mathtt{X}_1, \mathtt{X}_2)$ or $\mathtt{bool\_eq}(\mathtt{X}_1, -\mathtt{X}_2)$	$\mathtt{X}_1 = \mathtt{X}_2$ or $\mathtt{X}_1 = \neg\mathtt{X}_2$
(5)	$\mathtt{bool\_array\_op}([\mathtt{X}_1, \ldots, \mathtt{X}_n])$	$\mathtt{X}_1\ \mathtt{op}\ \mathtt{X}_2 \cdots \mathtt{op}\ \mathtt{X}_n$
(6)	$\mathtt{bool\_array\_op\_reif}([\mathtt{X}_1, \ldots, \mathtt{X}_n],\ \mathtt{X})$	$\mathtt{X}_1\ \mathtt{op}\ \mathtt{X}_2 \cdots \mathtt{op}\ \mathtt{X}_n \Leftrightarrow \mathtt{X}$
(7)	$\mathtt{bool\_op\_reif}(\mathtt{X}_1, \mathtt{X}_2,\ \mathtt{X})$	$\mathtt{X}_1\ \mathtt{op}\ \mathtt{X}_2 \Leftrightarrow \mathtt{X}$
(8)	$\mathtt{bool\_array\_lex}(\mathtt{Xs}_1, \mathtt{Xs}_2)$	$\mathtt{Xs}_1$ precedes $\mathtt{Xs}_2$ in the lex order
Integer relations (reified) and arithmetic		$\mathtt{rel} \in \{\mathtt{leq}, \mathtt{geq}, \mathtt{eq}, \mathtt{lt}, \mathtt{gt}, \mathtt{neq}\}$ $\mathtt{op} \in \{\mathtt{plus}, \mathtt{times}, \mathtt{div}, \mathtt{mod}, \mathtt{max}, \mathtt{min}\}$, $\mathtt{op}' \in \{\mathtt{plus}, \mathtt{max}, \mathtt{min}\}$
(9)	$\mathtt{int\_rel}(\mathtt{I}_1, \mathtt{I}_2)$	$\mathtt{I}_1\ \mathtt{rel}\ \mathtt{I}_2$
(10)	$\mathtt{int\_rel\_reif}(\mathtt{I}_1, \mathtt{I}_2,\ \mathtt{X})$	$\mathtt{I}_1\ \mathtt{rel}\ \mathtt{I}_2 \Leftrightarrow \mathtt{X}$
(11)	$\mathtt{int\_op}(\mathtt{I}_1, \mathtt{I}_2,\ \mathtt{I})$	$\mathtt{I}_1\ \mathtt{op}\ \mathtt{I}_2 = \mathtt{I}$
(12)	$\mathtt{int\_array\_op}'([\mathtt{I}_1, \ldots, \mathtt{I}_n],\ \mathtt{I})$	$\mathtt{I}_1\ \mathtt{op}' \cdots \mathtt{op}'\ \mathtt{I}_n = \mathtt{I}$
All Different and cardinality		$\mathtt{rel} \in \{\mathtt{leq}, \mathtt{geq}, \mathtt{eq}, \mathtt{lt}, \mathtt{gt}, \mathtt{neq}\}$
(13)	$\mathtt{allDiff}([\mathtt{I}_1, \ldots, \mathtt{I}_n])$	$\bigwedge_{i<j} \mathtt{I}_i \neq \mathtt{I}_j$
(14)	$\mathtt{bool\_array\_sum\_rel}([\mathtt{X}_1, \ldots, \mathtt{X}_n],\ \mathtt{I})$	$(\Sigma\ \mathtt{X}_i)\ \mathtt{rel}\ \mathtt{I}$
(15)	$\mathtt{comparator}(\mathtt{X}_1, \mathtt{X}_2, \mathtt{X}_3, \mathtt{X}_4)$	$\mathtt{sort}([\mathtt{X}_1, \mathtt{X}_2]) = [\mathtt{X}_3, \mathtt{X}_4]$

Table 1. *Syntax for a subset of BEE.*

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 15 constraint templates. Constraints (1-2) are about variable declarations: Booleans and integers. Constraint (3) signifies that a bit sequence is monotonic non-increasing, and is used to specify that an integer variable is in the order-encoding. Constraints (4-7) are about Boolean (and reified Boolean) statements. The cases for $\mathtt{bool\_array\_or}([\mathtt{X}_1, \ldots, \mathtt{X}_n])$ and $\mathtt{bool\_array\_xor}([\mathtt{X}_1, \ldots, \mathtt{X}_n])$ facilitate the specification of clauses and of xor clauses (supported in the CryptoMiniSAT solver (Soos 2010)). Constraint (8) specifies that two bit-vectors are ordered lexicographically. Constraints (9-12) are about integer relations and operations. Constraints (13-14) are the all-different constraint on integers and the cardinality constraint on Booleans. Constraint (15) specifies that sorting a bit pair $[\mathtt{X}_1, \mathtt{X}_2]$ (decreasing order) results in the pair $[\mathtt{X}_3, \mathtt{X}_4]$. This is a basic building block for the construction of sorting networks (Batcher 1968) used to encode cardinality constraints during compilation as described in (Asín et al. 2011) and in (Codish and Zazon-Ivry 2010).

```
:- use_module(bee_compiler, [compile/2]).
:- use_module(sat_solver, [sat/1]).
solve(Instance, Solution) :-
    encode(Instance, Map, Constraints),
    compile(Constraints, CNF),
    sat(CNF),
    decode(Map, Solution).
```

Fig. 1. A generic application of BEE.

4 An Example BEE Application: magic graph labeling

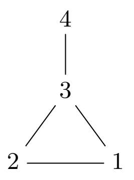
We illustrate the application of BEE to solve a graph labeling problem. A typical BEE application has the form depicted as Figure 1 where the predicate `solve/2` takes a problem `Instance` and provides a `Solution`. The specifics of the application are in the call to `encode/3` which given the `Instance` generates the `Constraints` that solve it together with a `Map` relating instance variables with constraint variables. The calls to `compile/2` and `sat/1` compile the constraints to a `CNF` and solve it applying a SAT solver. If the instance has a solution, the SAT solver binds the constraint variables accordingly. Then, the call to `decode/2`, using the `Map`, provides a `Solution` in terms of the instance variables. The definitions of `encode/3` and `decode/3` are application dependent and provided by the user. The predicates `compile/2` and `sat/1` provide the interface to BEE and the underlying SAT solver.

Graph labeling is about finding an assignment of integers to the vertices and edges of a graph subject to certain conditions. Graph labelings were introduced in the 60's and hundreds of papers on a wide variety of related problems have been published since then. See for example the survey by Gallian (2011) with more than 1200 references. Graph labelings have many applications. For instance in radars, xray crystallography, coding theory, etc.

We focus here on the vertex-magic total labeling (VMTL) problem where one should find for the graph $G = (V, E)$ a labeling that is a one-to-one map $V \cup E \to \{1, 2, \ldots, |V| + |E|\}$ with the property that the sum of the labels of a vertex and its incident edges is a constant K independent of the choice of vertex. A problem instance takes the form $vmtl(G, K)$ specifying the graph G and a constant K. The query `solve(vmtl(G,K),Solution)` poses the question: "Does there exist a vmtl labeling for G with magic constant K?" It binds `Solution` to indicate such a labeling if one exists, or to "unsat" otherwise. Figure 2 illustrates an example problem instance together with the constraints, `Cs` and the map, `M`, generated by the `encode/3` predicate for this instance. The constraints introduce integer variables for the vertices and edges, specify that these variables take "all different" values, and specify that the labels for each vertex with its incident edges sum to `K`. Solving the constraints from Figure 2 for the example VMTL instance binds the Map, `M`, as follows, indicating a solution:

$$
\mathtt{M} = \left[\begin{array}{ll}
((1,2),\ [1,1,1,1,1,1,1,0]), & (1,\ [1,1,1,1,0,0,0,0]),\\
((1,3),\ [1,1,1,0,0,0,0,0]), & (2,\ [1,1,1,1,1,0,0,0]),\\
((2,3),\ [1,1,0,0,0,0,0,0]), & (3,\ [1,0,0,0,0,0,0,0]),\\
((3,4),\ [1,1,1,1,1,1,1,1]), & (4,\ [1,1,1,1,1,1,0,0])
\end{array}\right]
$$

An Instance	The Graph	The Map

`Instance = vmtl(G,K),`
`G = (V,E),`
`V = [1,2,3,4],`
`E = [(1,2),(1,3),`
`(2,3),(3,4)],`
`K = 14`
		$\mathtt{M} = \begin{bmatrix} ((1,2),\ \mathtt{E}_1), & (1,\ \mathtt{V}_1), \\ ((1,3),\ \mathtt{E}_2), & (2,\ \mathtt{V}_2), \\ ((2,3),\ \mathtt{E}_3), & (3,\ \mathtt{V}_3), \\ ((3,4),\ \mathtt{E}_4), & (4,\ \mathtt{V}_4) \end{bmatrix}$

The Constraints

$$\mathtt{Cs} = \begin{bmatrix} \mathtt{new\_int}(\mathtt{V}_1,1,8), & \mathtt{new\_int}(\mathtt{E}_1,1,8), & \mathtt{int\_array\_plus}([\mathtt{V}_1,\mathtt{E}_1,\mathtt{E}_2],\mathtt{K}), \\ \mathtt{new\_int}(\mathtt{V}_2,1,8), & \mathtt{new\_int}(\mathtt{E}_2,1,8), & \mathtt{int\_array\_plus}([\mathtt{V}_2,\mathtt{E}_1,\mathtt{E}_3],\mathtt{K}), \\ \mathtt{new\_int}(\mathtt{V}_3,1,8), & \mathtt{new\_int}(\mathtt{E}_3,1,8), & \mathtt{int\_array\_plus}([\mathtt{V}_3,\mathtt{E}_2,\mathtt{E}_3,\mathtt{E}_4],\mathtt{K}), \\ \mathtt{new\_int}(\mathtt{V}_4,1,8), & \mathtt{new\_int}(\mathtt{E}_4,1,8), & \mathtt{int\_array\_plus}([\mathtt{V}_4,\mathtt{E}_4],\mathtt{K}), \\ \mathtt{new\_int}(\mathtt{K},14,14), & \mathtt{allDiff}([\mathtt{V}_1,\mathtt{V}_2,\mathtt{V}_3,\mathtt{V}_4,\mathtt{E}_1,\mathtt{E}_2,\mathtt{E}_3,\mathtt{E}_4]) & \end{bmatrix}$$

Fig. 2. A VMTL instance with the constraints and map generated by `encode/3`.

In Section 9 we report that using BEE enables us to solve interesting instances of the VMTL problem not previously solvable by other techniques.

5 Compiling BEE to CNF

The compilation of a constraint model to a CNF using BEE goes through three phases. In the first phase, (unary) bit blasting, integer variables (and constants) are represented as bit vectors in the order-encoding. Now all constraints are about Boolean variables. The second phase, the main loop of the compiler, is about constraint simplification. Three types of actions are applied: equi-propagation, partial evaluation, and decomposition of constraints. These are specified as a set of transitions which we write in the form $c_1 \overset{\theta}{\longmapsto} c_2$ to specify that constraint c_1 reduces to constraint c_2 generating the (possibly empty) substitution θ. Simplification is applied repeatedly until no rule is applicable. In the third, and final phase, simplified constraints are encoded to CNF. We elaborate below. To simplify the presentation, we assume that integer variables are represented in a positive interval starting from 0. As later detailed in Section 8 there is no such limitation in BEE.

Bit-blasting: Each integer variable declaration $\mathtt{new\_int}(\mathtt{I},\mathtt{c}_1,\mathtt{c}_2)$ triggers a unification $\mathtt{I} = [1,\ldots,1,\mathtt{X}_{\mathtt{c}_1+1},\ldots,\mathtt{X}_{\mathtt{c}_2}]$ and introduces a constraint $\mathtt{ordered}(\mathtt{I})$ to specify that the bits representing `I` are in the order-encoding. To illustrate bit-blasting, consider again the VMTL example detailed in Figure 2. Each variable in the `Map` occurs in a `new_int` declaration. So the following unifications are performed:

$$\begin{array}{ll} \mathtt{V}_1 = [1,\mathtt{V}_{1,2},\mathtt{V}_{1,3},\mathtt{V}_{1,4},\mathtt{V}_{1,5},\mathtt{V}_{1,6},\mathtt{V}_{1,7},\mathtt{V}_{1,8}], & \mathtt{E}_1 = [1,\mathtt{E}_{1,2},\mathtt{E}_{1,3},\mathtt{E}_{1,4},\mathtt{E}_{1,5},\mathtt{E}_{1,6},\mathtt{E}_{1,7},\mathtt{E}_{1,8}], \\ \mathtt{V}_2 = [1,\mathtt{V}_{2,2},\mathtt{V}_{2,3},\mathtt{V}_{2,4},\mathtt{V}_{2,5},\mathtt{V}_{2,6},\mathtt{V}_{2,7},\mathtt{V}_{2,8}], & \mathtt{E}_2 = [1,\mathtt{E}_{2,2},\mathtt{E}_{2,3},\mathtt{E}_{2,4},\mathtt{E}_{2,5},\mathtt{E}_{2,6},\mathtt{E}_{2,7},\mathtt{E}_{2,8}], \\ \mathtt{V}_3 = [1,\mathtt{V}_{3,2},\mathtt{V}_{3,3},\mathtt{V}_{3,4},\mathtt{V}_{3,5},\mathtt{V}_{3,6},\mathtt{V}_{3,7},\mathtt{V}_{3,8}], & \mathtt{E}_3 = [1,\mathtt{E}_{3,2},\mathtt{E}_{3,3},\mathtt{E}_{3,4},\mathtt{E}_{3,5},\mathtt{E}_{3,6},\mathtt{E}_{3,7},\mathtt{E}_{3,8}], \\ \mathtt{V}_4 = [1,\mathtt{V}_{4,2},\mathtt{V}_{4,3},\mathtt{V}_{4,4},\mathtt{V}_{4,5},\mathtt{V}_{4,6},\mathtt{V}_{4,7},\mathtt{V}_{4,8}], & \mathtt{E}_4 = [1,\mathtt{E}_{4,2},\mathtt{E}_{4,3},\mathtt{E}_{4,4},\mathtt{E}_{4,5},\mathtt{E}_{4,6},\mathtt{E}_{4,7},\mathtt{E}_{4,8}], \\ \mathtt{K} = \ [1,1,1,1,1,1,1,1,1,1,1,1,1,1] & \end{array}$$

Integer variables occurring in an `allDiff` constraint are bit-blasted twice: first, in the order-encoding, when declared, as explained above, and second, in the direct encoding, when processing the `allDiff` constraint, as described below.

Equi-propagation is about detecting situations in which a small number of constraints imply an equality of the form $X = L$ where X is a Boolean variable and L is a Boolean literal or constant. In this case X becomes redundant and can be replaced by L in all constraints. In BEE we consider as candidates for equi-propagation, individual constraints together with constraints specifying that their integer variables are in the order-encoding. If $X = L$ is such an equality, then equi-propagation is implemented by unifying X and L. This unification applies to all occurrences of X and in this sense "propagates" to other constraints involving X. Once equi-propagation detects such an equation, this may trigger further equi-propagation from other constraints. For example, consider the constraint $\texttt{int\_neq}(\texttt{I}_1, \texttt{I}_2)$ where $\texttt{I}_1 = [\texttt{x}_1, \texttt{x}_2, \texttt{x}_3, \texttt{x}_4]$ and $\texttt{I}_2 = [1, 1, 0, 0]$. We propagate that $(\texttt{x}_2 = \texttt{x}_3)$ because

$$\left(\begin{array}{l} \texttt{I}_1 = [\texttt{x}_1, \texttt{x}_2, \texttt{x}_3, \texttt{x}_4] \ \wedge \ \texttt{I}_2 = [1,1,0,0] \ \wedge \\ \texttt{int\_neq}(\texttt{I}_1, \texttt{I}_2) \ \wedge \ \texttt{ordered}(\texttt{I}_1) \end{array}\right) \models (\texttt{x}_2 = \texttt{x}_3).$$

To see why, consider that $\texttt{ordered}(\texttt{I}_1)$ implies that $\texttt{x}_2 \geq \texttt{x}_3$. Furthermore, also $\texttt{x}_2 \leq \texttt{x}_3$ as otherwise $\texttt{x}_2 = 1$ and $\texttt{x}_3 = 0$ which implies that $\texttt{I}_1 = [1, 1, 0, 0]$, contradicting $\texttt{int\_neq}(\texttt{I}_1, \texttt{I}_2)$.

In BEE, equi-propagation is implemented by a collection of ad-hoc transition rules for each type of constraint. While this approach is not complete — there are equations implied by a constraint that BEE will not detect — the implementation is fast, and works well in practice. An alternative approach is to implement equi-propagation, using BDD's, as described in (Metodi et al. 2011). This approach, though complete, is slower and not included in the current release of BEE.

The following are two of the simplification (equi-propagation) rules of BEE that apply to `int_neq` constraints:

$\texttt{neq}_1$: applies when one of the (order-encoding) integers in the relation is a constant and $\theta = \{X_1 = X_2\}$:

$$\texttt{int\_neq}\left(\begin{array}{l} [\ldots, X_1, X_2, \ldots] \\ [\ldots,\ 1,\ \ 0,\ \ldots] \end{array}\right) \overset{\theta}{\longmapsto} \texttt{int\_neq}\left(\begin{array}{l} [\ldots, X_1, X_1, \ldots] \\ [\ldots,\ 1,\ \ 0,\ \ldots] \end{array}\right)$$

$\texttt{neq}_2$: applies when the integers share common variables as in the rule template and $\theta = \{X_1 = X_2\}$:

$$\texttt{int\_neq}\left(\begin{array}{l} [\ldots,\ X_1,\ \ X_2,\ \ldots], \\ [\ldots, \neg X_2, \neg X_1, \ldots] \end{array}\right) \overset{\theta}{\longmapsto} \texttt{int\_neq}\left(\begin{array}{l} [\ldots,\ X_1,\ \ X_1,\ \ldots], \\ [\ldots, \neg X_1, \neg X_1, \ldots] \end{array}\right)$$

For the rule $\texttt{neq}_1$, observe that after applying this rule the constraint obtained is a tautology. Hence it is subsequently removed by one of the other "partial evaluation" rules. For the rule $\texttt{neq}_2$, to see why the equation $X_1 = X_2$ is implied by the constraint (on the left side of the rule), consider all possible truth values for the variables X_1 and X_2: (a) If $X_1 = 0$ and $X_2 = 1$ then both integers in the relation take the form $[\ldots, 0, 1, \ldots]$ violating their specification as `ordered`, so this is not possible. (b) If $X_1 = 1$ and $X_2 = 0$ then both numbers take the form $[1, \ldots, 1, 0, \ldots, 0]$

and are equal, violating the `neq` constraint. The only possible bindings for X_1 and X_2 are those where $X_1 = X_2$. The template expressed in rule $\texttt{neq}_2$ is not contrived. It comes up frequently as a result of applying other equi-propagation rules.

Partial evaluation is about simplifying constraints in view of variables that are (partially) instantiated, either because of information from the constraint model or else due to equi-propagation. Typical cases include constant elimination and elimination of tautologies. The following are some of BEE's partial evaluation rules that apply to `int_neq` constraints (ϵ denotes the empty substitution).

$\texttt{neq}_3$: applies to remove replicated variables:

$$\texttt{int\_neq}\left(\begin{array}{l}[\dots, X_1, X_1, \dots]\\ [\dots, Y_1, Y_1, \dots]\end{array}\right) \overset{\epsilon}{\longmapsto} \texttt{int\_neq}\left(\begin{array}{l}[\dots, X_1, \dots]\\ [\dots, Y_1, \dots]\end{array}\right)$$

$\texttt{neq}_4$: applies to remove leading 1 bits (there is a similar rule for trailing 0's):

$$\texttt{int\_neq}([1, 1, X_3, \dots], [Y_1, Y_2, Y_3, \dots]) \overset{\epsilon}{\longmapsto} \texttt{int\_neq}([1, X_3, \dots], [Y_2, Y_3, \dots])$$

We now detail three of the simplification rules (equi-propagation and partial evaluation) that apply to a constraint of the form `int_plus(A,B,C)` where we assume for simplicity of presentation (the tool supports the general case) that $\texttt{A} = [\texttt{A}_1, \dots, \texttt{A}_\texttt{n}]$, $\texttt{B} = [\texttt{B}_1, \dots, \texttt{B}_\texttt{m}]$, and $\texttt{C} = [\texttt{C}_1, \dots, \texttt{C}_{\texttt{n+m}}]$. We denote by `min(I)` (or `max(I)`) the minimal (or maximal) value that integer variable `I` can take, determined by the number of leading ones (or trailing zeros) in its bit representation.

Rule $\texttt{plus}_1$ is standard propagation for interval arithmetics. Rule $\texttt{plus}_2$ removes redundant bits (assigned values through $\texttt{plus}_1$). Rules $\texttt{plus}_{3(\texttt{a})}$ and $\texttt{plus}_{3(\texttt{b})}$ remove constraints and may seem contrived: `3(a)` assumes that $\texttt{m} = \texttt{0}$ and `3(b)` assumes that $\texttt{n} = \texttt{m}$ and that `C` represents the (same) constant `n`. However, in the general case, when `n`, `m` are arbitrary and constant `C` is represented in $\texttt{m} + \texttt{n}$ bits, then application of the other rules will reduce the constraint to one of these special cases.

$\texttt{plus}_1$: applies to propagate bounds: $\texttt{int\_plus}(\texttt{A}, \texttt{B}, \texttt{C}) \overset{\theta}{\longmapsto} \texttt{int\_plus}(\texttt{A}, \texttt{B}, \texttt{C})$ where

$$\theta = \left\{\begin{array}{ll} C_{max\{min(C), min(A)+min(B)\}} = 1, & C_{min\{max(C), max(A)+max(B)\}+1} = 0, \\ A_{max\{min(A), min(C)-max(B)\}} = 1, & A_{min\{max(A), max(C)-min(B)\}+1} = 0, \\ B_{max\{min(B), min(C)-max(A)\}} = 1, & B_{min\{max(B), max(C)-min(A)\}+1} = 0 \end{array}\right\}$$

$\texttt{plus}_2$: applies to remove leading 1's (there are similar rules for trailing 0's and for the case when the 1's or 0's are on $[\texttt{B}_1, \dots, \texttt{B}_\texttt{m}]$):

$$\texttt{int\_plus}\left(\begin{array}{l}[1, \texttt{A}_2, \dots, \texttt{A}_\texttt{n}],\\ [\texttt{B}_1, \dots, \texttt{B}_\texttt{m}],\\ [1, \texttt{C}_2, \dots, \texttt{C}_{\texttt{n+m}}]\end{array}\right) \overset{\epsilon}{\longmapsto} \texttt{int\_plus}\left(\begin{array}{l}[\texttt{A}_2, \dots, \texttt{A}_\texttt{n}],\\ [\texttt{B}_1, \dots, \texttt{B}_\texttt{m}],\\ [\texttt{C}_2, \dots, \texttt{C}_{\texttt{n+m}}]\end{array}\right)$$

$\texttt{plus}_{3(\texttt{a})}$: applies when `A` or `B` is the empty bit list and $\theta = \left\{ C_i = A_i \;\middle|\; 1 \leq i \leq n \right\}$

$$\texttt{int\_plus}([\texttt{A}_1, \dots, \texttt{A}_\texttt{n}], [\,], [\texttt{C}_1, \dots, \texttt{C}_\texttt{n}]) \overset{\theta}{\longmapsto} none$$

$\texttt{plus}_{3(\texttt{b})}$: applies when `C` is a constant `n` and $\theta = \left\{ A_i = \neg B_{n-i+1} \;\middle|\; 1 \leq i \leq n \right\}$

$$\texttt{int\_plus}([\texttt{A}_1, \dots, \texttt{A}_\texttt{n}], [\texttt{B}_1, \dots, \texttt{B}_\texttt{n}], [1, \dots 1, 0, \dots, 0]) \overset{\theta}{\longmapsto} none$$

We illustrate the simplification of a `int_plus` constraint by the following example.

Example 1 (*simplifying* `int_plus`*: equi-propagation and partial evaluation*)
Consider constraint $\mathtt{int\_plus(A,B,C)}$ where A and B are integer variables with domain [1..8] and C is the constant 14 represented in 16 bits. Constraint simplification follows the steps:

$$\boxed{\begin{array}{l}\mathtt{int\_plus(}\\ \quad [1,\mathtt{A}_2,\mathtt{A}_3,\mathtt{A}_4,\mathtt{A}_5,\mathtt{A}_6,\mathtt{A}_7,\mathtt{A}_8],\\ \quad [1,\mathtt{B}_2,\mathtt{B}_3,\mathtt{B}_4,\mathtt{B}_5,\mathtt{B}_6,\mathtt{B}_7,\mathtt{B}_8],\\ \quad [\underbrace{1,\quad \ldots\ldots\quad ,1}_{\mathtt{14\ times}},0,0]\\ )\end{array}} \xrightarrow{\mathtt{plus}_1} \boxed{\begin{array}{l}\mathtt{int\_plus(}\\ \quad [1,1,1,1,1,1,\mathtt{A}_7,\mathtt{A}_8],\\ \quad [1,1,1,1,1,1,\mathtt{B}_7,\mathtt{B}_8],\\ \quad [\underbrace{1,\quad \ldots\ldots\quad ,1}_{\mathtt{14\ times}},0,0]\\ )\end{array}} \xrightarrow{\mathtt{plus}_2}$$

$$\xrightarrow{\mathtt{plus}_2} \boxed{\mathtt{int\_plus}\left(\begin{array}{l}[\mathtt{A}_7,\mathtt{A}_8],\ [\mathtt{B}_7,\mathtt{B}_8],\\ {[1,1,0,0]}\end{array}\right)} \xrightarrow{\mathtt{plus}_{3(b)}} \boxed{\begin{array}{l}\mathtt{none},\ \textit{binding}:\\ \mathtt{B}_7=\neg\mathtt{A}_8,\ \mathtt{B}_8=\neg\mathtt{A}_7)\end{array}}$$

After constraint simplification variables `A` and `B` take the form: $[1,1,1,1,1,1,\mathtt{A}_7,\mathtt{A}_8]$ and $[1,1,1,1,1,1,\neg\mathtt{A}_8,\neg\mathtt{A}_7]$ (and nothing is left to encode to CNF).

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 straightforward divide and conquer recursive definition. At the base case, if `As=[A]` then the constraint is replaced by `int_eq(A,Sum)`, or if $\mathtt{As} = [\mathtt{A}_1, \mathtt{A}_2]$ then it is replaced by $\mathtt{int\_plus(A_1, A_2, Sum)}$. In the general case `As` is split into two halves, then constraints are generated to sum these halves, and then an additional `int_plus` constraint is introduced to sum the two sums.

As another example, consider the $\mathtt{int\_plus(A_1, A_2, A)}$ constraint. One approach, supported by BEE, decomposes the constraint as an odd-even merger (from the context of odd-even sorting networks) (Batcher 1968). Here, the sorted sequences of bits $\mathtt{A}_1$ and $\mathtt{A}_2$ are merged to obtain their sum `A`. This results in a model with $O(n \log n)$ `comparator` constraints (and later in an encoding with $O(n \log n)$ clauses). Another approach, also supported in BEE, does not decompose the constraint but encodes it directly to a CNF of size $O(n^2)$, as in the context of so-called totalizers (Bailleux and Boufkhad 2003). A hybrid approach, leaves the choice to BEE, depending on the size of the domains of the variables involved. Finally, we note that the user can configure BEE to fix the way it compiles this constraint (and others).

CNF encoding is the last phase and applies to all remaining simplified constraints. The encoding of constraints to CNF is standard and similar to the encodings in Sugar (Tamura et al. 2009).

Cardinality constraints are about the cardinality of sets of Boolean variables and are specified by the template $\mathtt{bool\_array\_sum\_rel([X_1, \ldots, X_n],\ I)}$. Cardinality constraints are normalized, see e.g., (Eén and Sörensson 2006), so we only consider $\mathtt{rel} \in \{\mathtt{leq}, \mathtt{eq}\}$. Partial evaluation rules for cardinality constraints are the obvious. For example, in the special case when `I` is a constant:

$$\begin{array}{l}
\texttt{bool\_array\_sum\_leq}([\texttt{X}_1,\texttt{X}_2,1,\texttt{X}_4],\ 3) \mapsto \texttt{bool\_array\_sum\_leq}([\texttt{X}_1,\texttt{X}_2,\texttt{X}_4],\ 2)\\
\texttt{bool\_array\_sum\_leq}([\texttt{X}_1,\texttt{X}_2,0,\texttt{X}_4],\ 3) \mapsto \texttt{bool\_array\_sum\_leq}([\texttt{X}_1,\texttt{X}_2,\texttt{X}_4],\ 3)\\
\texttt{bool\_array\_sum\_leq}([\texttt{X}_1,\texttt{X}_2,-\texttt{X}_1,\texttt{X}_4],\ 3) \mapsto \texttt{bool\_array\_sum\_leq}([\texttt{X}_2,\texttt{X}_4],\ 2)
\end{array}$$

The special case, when `I` is the constant 1 is called the "at-most-one" constraint and it has been studied extensively (for a recent survey see (Frisch and Giannaros 2010)). In BEE, we support two different encodings for this case (the user can choose). The first is the standard "pairwise" encoding which specifies a clause $(\neg x_i \vee \neg x_j)$ for each pair of Boolean variables x_i and x_j. This encoding introduces $O(n^2)$ clauses and is sometimes too large. The second, is a more compact encoding which follows the approach described in (Chen 2010). In the general case (when $\texttt{I} > 1$) the constraint is decomposed, much the same as the `int_array_plus` constraint, to a network of `int_plus` constraints.

The All-different constraint specifies that a set of integer variables take all different values. Although we adopt the order-encoding for integer variables, it is well accepted that for these constraints the direct encoding is superior (Ansótegui et al. 2004). For this reason, in BEE, when processing the constraint, a dual representation is chosen. When integer variable `I`, occurring in an `allDiff` constraint, is declared, it was unified with its unary representation in the order-encoding: $\texttt{I} = [\texttt{x}_1, \ldots, \texttt{x}_n]$. In addition, we associate `I` with a new bit-blast, $[\texttt{d}_0, \ldots, \texttt{d}_n]$, in the direct encoding. We introduce for each such `I` a channeling formula to capture the relation between its two representations.

$$\texttt{channel}([\texttt{x}_1, \ldots, \texttt{x}_n], [\texttt{d}_0, \ldots, \texttt{d}_n]) = \left(\begin{array}{c} d_0 = \neg x_1 \\ \wedge\ d_n = x_n \end{array} \right) \wedge \bigwedge_{\texttt{i}=1}^{\texttt{n}-1} (\texttt{d}_\texttt{i} \leftrightarrow \texttt{x}_\texttt{i} \wedge \neg \texttt{x}_{\texttt{i}+1})$$

During constraint simplification, the $\texttt{allDiff}([\texttt{I}_1, \ldots, \texttt{I}_n])$ constraint is viewed as a bit matrix where each row consists of the bits $[\texttt{d}_{\texttt{i}0}, \ldots, \texttt{d}_{\texttt{im}}]$ for $\texttt{I}_\texttt{i}$ in the direct encoding. The element d_{ij} is true iff I_i takes the value j. The j^{th} column specifies which of the I_i take the value j and hence, at most one variable in a column may take the value true. BEE distinguishes the special case when $[\texttt{I}_1, \ldots, \texttt{I}_n]$ must take precisely n different values. In this case the constraint is about "permutation". We denote this by a flag (*) as in $\texttt{allDiff}^*([\texttt{I}_1, \ldots, \texttt{I}_n])$. In this case, exactly one bit in each column of the representation must take the value true.

To simplify an `allDiff` constraint, BEE applies simplification rules to the implicit cardinality constraints on the columns and also two specific `allDiff` rules. The first is essentially the usual domain consistent propagator (Régin 1994) focusing on Hall sets of size 2. The second rule applies only to an $\texttt{allDiff}^*$ constraint which is about permutation. We denote the values that $\texttt{I}_\texttt{i}$ can take as $\texttt{dom}(\texttt{I}_\texttt{i})$.

$\texttt{allDiff}_1$: when $\texttt{dom}(\texttt{I}_1) = \texttt{dom}(\texttt{I}_2) = \{\texttt{v}_1, \texttt{v}_2\}$:

$$\texttt{allDiff}([\texttt{I}_1, \texttt{I}_2, \texttt{I}_3, \ldots, \texttt{I}_n]) \overset{\theta}{\longmapsto} \texttt{allDiff}([\texttt{I}_3, \ldots, \texttt{I}_n])$$

where $\theta = \bigcup_{3 \leq i \leq n} \{d_{i,v_1} = 0, d_{i,v_2} = 0\} \cup \{d_{1,v_1} = -d_{2,v_1}, d_{1,v_2} = -d_{2,v_2}\}$.

$\texttt{allDiff}_2:$ when $\{\mathtt{v_1},\mathtt{v_2}\} \subseteq \mathtt{dom}(\mathtt{I_1}) \cap \mathtt{dom}(\mathtt{I_2})$, and for $i \geq 3$, $\{\mathtt{v_1},\mathtt{v_2}\} \cap \mathtt{dom}(\mathtt{I_i}) = \emptyset$

$$\texttt{allDiff}^*([\mathtt{I_1},\ldots,\mathtt{I_n}]) \xmapsto{\theta} \texttt{allDiff}^*([\mathtt{I_1},\ldots,\mathtt{I_n}])$$

where $\theta = \bigcup_{j\neq v_1, j\neq v_2}\{d_{1j} = 0, d_{2,j} = 0\}$.

To illustrate the two rules for `allDiff` consider the following.

Example 2
Consider an `allDiff` constraint on 5 integer variables taking values in the interval $[0,7]$ where the first two can take only values 0 and 1. So, they are a Hall set of size two and rule $\texttt{allDiff}_1$ applies. We present the simplification step on the order encoding representation (though it is triggered through the direct encoding representation):

$$\texttt{allDiff}\begin{pmatrix} [X_{1,1}, & 0, & \ldots, & 0] \\ [X_{2,1}, & 0, & \ldots, & 0] \\ [X_{3,1}, & X_{3,2}, & \ldots, & X_{3,7}] \\ [X_{4,1}, & X_{4,2}, & \ldots, & X_{4,7}] \\ [X_{5,1}, & X_{5,2}, & \ldots, & X_{5,7}] \end{pmatrix} \xmapsto{\theta} \texttt{allDiff}\begin{pmatrix} [1, X_{3,2}, \ldots, X_{3,7}] \\ [1, X_{4,2}, \ldots, X_{4,7}] \\ [1, X_{5,2}, \ldots, X_{5,7}] \end{pmatrix}$$

where $\theta = \{X_{1,1} = \neg X_{2,1}, X_{3,1} = 0, X_{4,1} = 0, X_{5,1} = 0\}$.

Now consider a setting where an `allDiff` constraint is about 5 variables that can take 5 values (permutation) and the first two are the only two that can take values 0 and 1. So rule $\texttt{allDiff}_2$ applies. We present the simplification step on the order encoding representation (though it is triggered through the direct encoding representation):

$$\texttt{allDiff}^*\begin{pmatrix} [X_{1,1}, & X_{1,2}, & X_{1,3}, & X_{1,4}] \\ [X_{2,1}, & X_{2,2}, & X_{2,3}, & X_{2,4}] \\ [1, & 1, & X_{3,3}, & X_{3,4}] \\ [1, & 1, & X_{4,3}, & X_{4,4}] \\ [1, & 1, & X_{5,3}, & X_{5,4}] \end{pmatrix} \xmapsto{\theta} \texttt{allDiff}^*\begin{pmatrix} [X_{1,1}, & 0, & 0, & 0] \\ [X_{2,1}, & 0, & 0, & 0] \\ [1, & 1, & X_{3,3}, & X_{3,4}] \\ [1, & 1, & X_{4,3}, & X_{4,4}] \\ [1, & 1, & X_{5,3}, & X_{5,4}] \end{pmatrix}$$

where $\theta = \{X_{1,2},\ldots,X_{1,4} = 0, X_{2,2},\ldots,X_{2,4} = 0\}$.

When no further simplification rules apply the `allDiff` constraint is decomposed to the corresponding cardinality constraints on the columns of its bit matrix representation.

6 Constraint simplification in the VMTL example

Consider again the VMTL example and the constraints from Figure 2. We focus on three constraints and follow the steps made when compiling these (we write "14" as short for $[\underbrace{1,1,\ldots,1}_{14}]$).

(1) $\texttt{int\_array\_plus}([\mathtt{V_4},\mathtt{E_4}],\mathtt{14})$
(2) $\texttt{allDiff}([\mathtt{V_1},\mathtt{V_2},\mathtt{V_3},\mathtt{V_4},\mathtt{E_1},\mathtt{E_2},\mathtt{E_3},\mathtt{E_4}])$,
(3) $\texttt{int\_array\_plus}([\mathtt{V_3},\mathtt{E_2},\mathtt{E_3},\mathtt{E_4}],\mathtt{14})$,

In the first steps, constraint (1) is decomposed to an `int_plus` constraint which has the same form as the constraint in Example 1. So, we have the bindings $\mathtt{V_4} = [\mathtt{1},\mathtt{1},\mathtt{1},\mathtt{1},\mathtt{1},\mathtt{1},\mathtt{V_{4,7}},\mathtt{V_{4,8}}]$ and $\mathtt{E_4} = [\mathtt{1},\mathtt{1},\mathtt{1},\mathtt{1},\mathtt{1},\mathtt{1},\neg\mathtt{V_{4,8}},\neg\mathtt{V_{4,7}}]$. Now, consider the allDiff constraint (2). BEE determines that this constraint is about permutation (8 integer variables with 8 different values in the range [1,8]). The simplification rules for allDiff detect that $\{\mathtt{V}_4, \mathtt{E}_4\}$ must take together the two values 6 and 8 (using a simplification rule similar to $\mathtt{neq}_2$) triggerring the substitution $\{\mathtt{V}_{4,7} = \mathtt{V}_{4,8}\}$. Now rule $\mathtt{allDiff}_1$ detects a Hall set $\{\mathtt{V}_4, \mathtt{E}_4\}$ of size two:

$$\mathtt{allDiff}([\mathtt{V}_1, \mathtt{V}_2, \mathtt{V}_3, \mathtt{V}_4, \mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3, \mathtt{E}_4]) \xrightarrow{\theta} \mathtt{allDiff}([\mathtt{V}_1, \mathtt{V}_2, \mathtt{V}_3, \mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3])$$

where θ is the unification that imposes $\mathtt{V}_1, \mathtt{V}_2, \mathtt{V}_3, \mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3 \neq \mathtt{6}, \mathtt{8}$. So we have the following bindings (where the impact of θ is underlined):

$$\begin{array}{ll}
\mathtt{V}_1 = [\mathtt{1}, \mathtt{V}_{1,2}, \mathtt{V}_{1,3}, \mathtt{V}_{1,4}, \mathtt{V}_{1,5}, \underline{\mathtt{V}_{1,7}, \mathtt{V}_{1,7}}, \mathtt{0}] & \mathtt{E}_1 = [\mathtt{1}, \mathtt{E}_{1,2}, \mathtt{E}_{1,3}, \mathtt{E}_{1,4}, \mathtt{E}_{1,5}, \underline{\mathtt{E}_{1,7}, \mathtt{E}_{1,7}}, \mathtt{0}] \\
\mathtt{V}_2 = [\mathtt{1}, \mathtt{V}_{2,2}, \mathtt{V}_{2,3}, \mathtt{V}_{2,4}, \mathtt{V}_{2,5}, \underline{\mathtt{V}_{2,7}, \mathtt{V}_{2,7}}, \mathtt{0}] & \mathtt{E}_2 = [\mathtt{1}, \mathtt{E}_{2,2}, \mathtt{E}_{2,3}, \mathtt{E}_{2,4}, \mathtt{E}_{2,5}, \underline{\mathtt{E}_{2,7}, \mathtt{E}_{2,7}}, \mathtt{0}] \\
\mathtt{V}_3 = [\mathtt{1}, \mathtt{V}_{3,2}, \mathtt{V}_{3,3}, \mathtt{V}_{3,4}, \mathtt{V}_{3,5}, \underline{\mathtt{V}_{3,7}, \mathtt{V}_{3,7}}, \mathtt{0}] & \mathtt{E}_3 = [\mathtt{1}, \mathtt{E}_{3,2}, \mathtt{E}_{3,3}, \mathtt{E}_{3,4}, \mathtt{E}_{3,5}, \underline{\mathtt{E}_{3,7}, \mathtt{E}_{3,7}}, \mathtt{0}] \\
\mathtt{V}_4 = [\mathtt{1}, \mathtt{1}, \mathtt{1}, \mathtt{1}, \mathtt{1}, \mathtt{1}, \mathtt{V}_{4,7}, \mathtt{V}_{4,7}] & \mathtt{E}_4 = [\mathtt{1}, \mathtt{1}, \mathtt{1}, \mathtt{1}, \mathtt{1}, \mathtt{1}, \neg\mathtt{V}_{4,7}, \neg\mathtt{V}_{4,7}]
\end{array}$$

Consider now the constraint (3). Equi-propagation (because of bounds) dictates that $\mathtt{max}(\mathtt{V}_1) = \mathtt{max}(\mathtt{V}_2) = \mathtt{max}(\mathtt{V}_3) = \mathtt{5}$, so this constraint then simplifies as follows:

$$\boxed{\begin{array}{l}
\mathtt{int\_array\_plus}([\\
\quad [\mathtt{1}, \mathtt{V}_{3,2}, \mathtt{V}_{3,3}, \mathtt{V}_{3,4}, \mathtt{V}_{3,5}, \mathtt{0}, \mathtt{0}, \mathtt{0}], \\
\quad [\mathtt{1}, \mathtt{E}_{2,2}, \mathtt{E}_{2,3}, \mathtt{E}_{2,4}, \mathtt{E}_{2,5}, \mathtt{0}, \mathtt{0}, \mathtt{0}], \\
\quad [\mathtt{1}, \mathtt{E}_{3,2}, \mathtt{E}_{3,3}, \mathtt{E}_{3,4}, \mathtt{E}_{3,5}, \mathtt{0}, \mathtt{0}, \mathtt{0}], \\
\quad [\mathtt{1}, \mathtt{1}, \mathtt{1}, \mathtt{1}, \mathtt{1}, \mathtt{1}, \neg\mathtt{V}_{4,7}, \neg\mathtt{V}_{4,7}],\ 14\ ])
\end{array}} \longmapsto \boxed{\begin{array}{l}
\mathtt{int\_array\_plus}([\\
\quad [\mathtt{V}_{3,2}, \mathtt{V}_{3,3}, \mathtt{V}_{3,4}, \mathtt{V}_{3,5}], \\
\quad [\mathtt{E}_{2,2}, \mathtt{E}_{2,3}, \mathtt{E}_{2,4}, \mathtt{E}_{2,5}], \\
\quad [\mathtt{E}_{3,2}, \mathtt{E}_{3,3}, \mathtt{E}_{3,4}, \mathtt{E}_{3,5}], \\
\quad [\neg\mathtt{V}_{4,7}, \neg\mathtt{V}_{4,7}],\ 5\ ])
\end{array}}$$

After applying simplification and decomposition rules on all the constraints from Figure 2 until no further rules can be applyed, the constraints will be encoded to CNF. The generated CNF contains 301 clauses and 48 Boolean variables. Compiling the same set of constraints from Figure 2 without applying simplification rules generates a larger CNF which contains 642 clauses and 97 Boolean variables.

7 Another Example BEE Application: DNA word problem

The DNA word problem (Problem 033 of CSPLib) seeks the largest parameter n, such that there exists a set S of n eight-letter words over the alphabet $\Sigma = \{A, C, G, T\}$ with the following properties: **(1)** Each word in S has exactly 4 symbols from $\{C, G\}$; **(2)** Each pair of distinct words in S differ in at least 4 positions; and **(3)** For every $x, y \in S$: x^R (the reverse of x) and y^C (the word obtained by replacing each A by T, each C by G, and vice versa) differ in at least 4 positions.

In (Frutos et al. 1997), the authors present a strategy to solve this problem where the four letters are modeled by bit-pairs $\langle t, m\rangle$. Each eight-letter word can then be viewed as the combination of a *"t-part"*, $\langle t_1, \ldots, t_8\rangle$, which is a bit-vector, and a *"m-part"*, $\langle m_1, \ldots, m_8\rangle$, also a bit-vector. Building on the approach described in (Frutos et al. 1997), we pose conditions on sets of *"t-parts"* and *"m-parts"*, T and M, so that their Cartesian product $S = T \times M$ will satisfy the requirements of the original problem. From the three conditions below, T is required to satisfy $(1')$ and $(2')$, and M is required to satisfy $(2')$ and $(3')$. For a set of bit-vectors V, the conditions are: **(1′)** Each bit-vector in V sums to 4; **(2′)** Each pair of distinct bit-vectors in V differ in at least 4 positions; and **(3′)** For each pair of bit-vectors (not necessarily distinct) $u, v \in V$, u^R (the reverse of u) and v^C (the complement of v) differ in at least 4 positions. This is equivalent to requiring that $(u^r)^c$ differs from v in at least 4 positions.

It is this strategy that we model in our BEE encoding. An instance takes the form $\mathtt{dna}(\mathtt{n}_1, \mathtt{n}_2)$ signifying the numbers of bit-vectors, n_1 and n_2 in the sets T and M. Without loss of generality, we impose, to remove symmetries, that T and M are lexicographically ordered. A solution is the Cartesian product $S = T \times M$. In Section 9 we report that using BEE enables us to solve interesting instances of the problem not previously solvable by other techniques.

8 Implementation

BEE is implemented in (SWI) Prolog and can be applied in conjunction with the CryptoMiniSAT solver (Soos 2010) through a Prolog interface (Codish et al. 2008). BEE can be downloaded from (Metodi 2012) where one can find also the examples from this paper and others. The distribution includes also a solver, which we call BumbleBEE, which enables to specify a BEE model as an input file and solve it. The output is a set of bindings to the declared variables in the model.

In BEE, Boolean variables are represented as Prolog variables. The negation of `X` is represented as `-X`. The truth values, *true* and *false*, are denoted `1` and `-1`. Integer variables (including negative range values) are represented in the order-encoding. When processing (bit-blasting) a declaration $\mathtt{new\_int(I, Min, Max)}$, Prolog variable `I` is unified with the tuple `(Min,Max,Bits,LastBit)` where `Min` and `Max` are constants indicating the interval domain of `I`, `Bits` is a list of $(\mathtt{Max} - \mathtt{Min})$ variables, and `LastBit` is the last variable of `Bits`. This representation is more concise than the one assumed for simplicity in the previous sections and it also supports negative numbers. Maintaining direct access to the last bit in the representation (we already can access the first bit through the list `Bits`) facilitates a (constant time) check if the lower and upper bound values of a variable has changed. This way we can more efficiently determine when (certain) simplification rules apply. We make a few notes: (1) Integer variables must be declared before use; (2) BEE allows the use of constants in constraints instead of declaring them as integer variables (for example $\mathtt{int\_gt(I, 5)}$ represents a declaration $\mathtt{new\_int(I', 5, 5)}$ together with the constraint $\mathtt{int\_gt(I, I')}$); (3) integer variables can be negated.

BEE maintains constraints as a Prolog list (of terms). Each type of constraint is associated with corresponding rules for simplification, decomposition, and encoding to CNF. After bit-blasting, constraints are first simplified (equi-propagation and partial evaluation) using these rules until no further rules apply. During this process, if a pair of literals is equated (e.g. as in `X=Y, X=-Y, X=1, X=-1`), then they are unified, thus propagating the effect to other constraints. After constraint simplification, some constraints are decomposed, and this process repeats. We end up with a set of “basic” constraints (which cannot be further decomposed or simplified). These are then encoded to CNF.

instance		BEE (dual encoding)				BEE (order encoding)				Sugar		
		comp	clauses	vars	sat	comp	clauses	vars	sat	clauses	vars	sat
25-264-0	sat	0.23	6509	1317	0.33	0.36	33224	887	8.95	126733	10770	34.20
25-264-1	sat	0.20	7475	1508	3.29	0.30	34323	917	97.50	127222	10798	13.93
25-264-2	sat	0.21	6531	1329	0.07	0.30	35238	905	2.46	127062	10787	8.06
25-264-3	sat	0.21	6819	1374	0.83	0.29	32457	899	18.52	127757	10827	44.03
25-264-4	sat	0.21	7082	1431	0.34	0.29	32825	897	19.08	126777	10779	85.92
25-264-5	sat	0.21	7055	1431	3.12	0.30	33590	897	46.15	126973	10784	41.04
25-264-6	sat	0.21	7712	1551	0.34	0.33	39015	932	69.81	128354	10850	12.67
25-264-7	sat	0.21	7428	1496	0.13	0.30	36580	937	19.93	127106	10794	7.01
25-264-8	sat	0.21	6603	1335	0.18	0.27	31561	896	10.32	124153	10687	9.69
25-264-9	sat	0.21	6784	1350	0.19	0.27	35404	903	34.08	128423	10853	38.80
25-264-10	unsat	0.21	6491	1296	0.04	0.30	33321	930	10.92	126999	10785	57.75
25-264-11	unsat	0.12	1	0	0.00	0.28	37912	955	0.09	125373	10744	0.47
25-264-12	unsat	0.16	1	0	0.00	0.29	39135	984	0.08	127539	10815	0.57
25-264-13	unsat	0.12	1	0	0.00	0.29	35048	944	0.09	127026	10786	0.56
25-264-14	unsat	0.23	5984	1210	0.07	0.28	31093	885	11.60	126628	10771	15.93
Total					8.93				349.58			370.63

Table 2. *QCP results for* 25 × 25 *instances with 264 holes*

9 Experiments

We report on our experience in applying BEE. To appreciate the ease in its use, and for further details, the reader is encouraged to view the example encodings available with the tool (Metodi 2012). All experiments run on an Intel Core 2 Duo E8400 3.00GHz CPU with 4GB memory under Linux (Ubuntu lucid, kernel 2.6.32-24-generic). BEE is written in Prolog and run using SWI Prolog v6.0.2 64-bits. Comparisons with Sugar (v1.15.0) are based on the use of identical constraint models, apply the same SAT solver (CryptoMiniSat v2.5.1), and run on the same machine. For all of the tables describing experiments, columns indicate:

`comp`:	compile time (seconds)	`vars`:	number of CNF variables
`clauses`:	number of CNF clauses	`sat`:	SAT solving time (seconds)

We first focus on the impact of the dual representation for `allDiff` constraints. We report on the application of BEE to Quasi-group completion problems (QCP), proposed by Gomes et al. (1997) as a constraint satisfaction benchmark, where the model is a conjunction of `allDiff` constraints.

Quasi-group completion: We consider 15 instances from the 2008 CSP competition[1]. Table 2 considers three settings: BEE with its dual encoding for `allDiff` constraints, BEE using only the order encoding (equivalent to using `int_neq` constraints instead of `allDiff`), and Sugar. The results indicate that: (1) Application of BEE using the dual representation for `allDiff` is 38 times faster and produces 20 times less clauses (in average) than when using the order-encoding alone (despite the need to maintain two encodings); (2) Without the dual representation, solving encodings generated by BEE is only slightly faster but BEE generates CNF

[1] `http://www.cril.univ-artois.fr/CPAI08/`

K_8	k	BEE	Sugar	K_{10}	k	BEE	Sugar	k	BEE	Sugar
	143	1.26	2.87		277	5.31	9.25	267	88.51	175.70
	142	10.14	1.62		276	7.11	9.91	266	229.80	247.56
	141	7.64	2.94		275	13.57	19.63	265	1335.31	259.45
	140	14.68	6.46		274	4.93	9.24	264	486.09	513.61
	139	25.60	6.67		273	45.94	9.03	263	236.68	648.43
	138	12.99	2.80		272	22.74	86.45	262	1843.70	6429.25
	137	22.91	298.58		271	7.35	9.49	261	2771.60	7872.76
	136	14.46	251.82		270	6.03	55.94	260	4873.99	∞
	135	298.54	182.90		269	5.20	11.05	259	∞	∞
	134	331.80	∞		268	94.44	424.89	258	∞	∞
Average								Average		
clauses $\times$**1000**		248	402				clauses $\times$**1000**		1229	1966
vars		5688	9370				vars		15529	25688

Table 3. *VMTL results for K_8 and K_{10} (times are in seconds)*

encodings 4 times smaller (on average) than those generated by Sugar. Observe that 3 instances are found unsatisfiable by BEE (indicated by a CNF with a single clause and no variables). We comment that Sugar preprocessing times are higher than those of BEE and not indicated in the table.

To further appreciate the impact of the tool we describe results for three additional applications which shift the state-of-the-art with respect to what could previously be solved. The experiments clearly illustrate that BEE decreases the size of CNF encodings as well as the subsequent SAT solving time.

Magic labels: In (MacDougall et al. 2002) the authors conjecture that the n vertex complete graph, K_n, for $n \geq 5$ has a vertex magic total labeling with magic constants for specific range of values of k, determined by n. This conjecture is proved correct for all odd n and verified by brute force for $n = 6$. We address the cases for $n = 8$ and $n = 10$ which involve 15 instances (different values of K) for $n = 8$, and 23 (different values of K) for $n = 10$. Starting from the simple constraint model (illustrated by the example in Figure 2), we add additional constraints to exploit that the graphs are symmetric: (1) We assume that the edge with the smallest label is $e_{1,2}$; (2) We assume that the labels of the edges incident to v_1 are ordered and hence introduce constraints $e_{1,2} < e_{1,3} < \cdots < e_{1,n}$; (3) We assume that the label of edge $e_{1,3}$ is smaller than the labels of the edges incident to v_2 (except $e_{1,2}$) and introduce constraints accordingly. In this setting BEE can solve all except 2 instances with a 4 hour timeout and Sugar can solve all except 4.

Table 3 depicts results for the 10 hardest instances for K_8 and the 20 hardest for K_{10} with a 4 hour time-out. BEE compilation times are on the order of 0.5 sec/instance for K_8 and 2.5 sec/instance for K_{10}. Sugar encoding times are slightly larger. The instances are indicated by the magic constant, k; the columns for BEE and Sugar indicate SAT solving times (in seconds). The bottom two lines indicate average encoding sizes (numbers of clauses and variables).

The results indicate that the Sugar encodings are (in average) about 60% larger, while the average SAT solving time for the BEE encodings is about 2 times faster (average excluding instances where Sugar times-out).

To address the two VMTL instances not solvable using the BEE models described above (K_{10} with magic labels 259 and 258), we partition the problem fixing the values of $e_{1,2}$ and $e_{1,3}$ and maintaining all of the other constraints. Analysis of the symmetry breaking constraints indicates that this results in 198 new instances for each of the two cases. The original VMTL instance is solved if any one of of these 198 instances is solved. So, we solve them in parallel. Fixing $e_{1,2}$ and $e_{1,3}$ "fuels" the compiler so the encodings are considerably smaller. The instance for $k = 259$ is solved in 1379.50 seconds where $e_{1,2} = 1$ and $e_{1,3} = 6$. The compilation time is 2.09 seconds and the encoding consists in 1056107 clauses and 14143 variables.

To the best of our knowledge, the hard instances from this suite are beyond the reach of all previous approaches to program the search for magic labels. The SAT based approach presented in (Jäger 2010) cannot handle these.[2] The comparison with Sugar indicates the impact of the compiler.

DNA word problem: Mancini et al. (2008) provide a comparison of several state-of-the-art solvers applied to the DNA word problem with a variety of encoding techniques. Their best reported result is a solution with 87 DNA words, obtained in 554 seconds, using an OPL (van Hentenryck 1999) model with lexicographic order to break symmetry. In (Frutos et al. 1997), the authors report a solution composed from two pairs of (t-part and m-part) sets $\langle T_1, M_1\rangle$ and $\langle T_2, M_2\rangle$ where $|T_1| = 6$, $|M_1| = 16$, $|T_2| = 2$, $|M_2| = 6$. This forms a set S with $(6 \times 16) + (2 \times 6) = 108$ DNA words. Marc van Dongen reports a larger solution with 112 words.[3] Using BEE, we find, in a fraction of a second, a template of size 14 and a map of size 8. This provides a solution of size $14 \times 8 = 112$ to the DNA word problem. Running Comet (v2.0.1) we find a 112 word solution in about 10 seconds using a model by Håkan Kjellerstrand.[4] We also prove that there does not exist a template of size 15 (0.15 seconds), nor a map of size 9 (4.47 seconds). These facts were unknown prior to BEE. Proving that there is no solution to the DNA word problem with more than 112 words, not via the two part t-m strategy, is still an open problem.

Model Based Diagnostics (MBD) is an artificial intelligence based approach that aims to cope with the, so-called, diagnosis problem (e.g. (Reiter 1987)). In (Metodi et al. 2012), we (with other researchers) focus on a notion of minimal cardinality MBD and apply BEE to model and solve the instances of a standard MBD benchmark. Experimental evidence (see (Metodi et al. 2012)), indicates that our approach is superior to all existing algorithms for minimal cardinality MBD. We determine, for the first time, minimal cardinality diagnoses for the entire standard benchmark. Prior attempts to apply SAT for MBD (for example, by Smith et al. (2005) and Feldman et al. (2010) where a MaxSAT solver is used) indicate that SAT solvers perform poorly on the standard benchmarks. So, BEE really makes the difference.

[2] Personal communication (Gerold Jäger), March 2012.
[3] See `http://www.cs.st-andrews.ac.uk/~ianm/CSPLib/`.
[4] See `http://www.hakank.org/comet/word_design_dna1.co`.

10 Conclusion

We introduce BEE, a compiler to encode finite domain constraints to CNF. A key design point is to apply bit-level techniques, locally as prescribed by the word-level constraints in a model. Optimizations are based on equi-propagation and partial evaluation. Implemented in Prolog, compilation times are typically small (measured in seconds) even for instances which result in several millions of CNF clauses. On the other hand, the reduction in SAT solving time can be larger in orders of magnitude.

It is well-understood that making a CNF smaller is not the ultimate goal: often smaller CNF's are harder to solve. Indeed, one often introduces redundancies to improve SAT encodings: so removing them is counter productive. Our experience is that BEE reduces the size of an encoding in a way that is productive for the subsequent SAT solving. In particular, by removing variables that can be determined "at compile time" to be definitely equal (or definitely different) in any solution.

The simplification rules illustrated in Section 5 apply standard constraint programming techniques (i.e. to reduce variable domains). However, equi-propagation is more powerful. It focuses, in general, in specializing the bit-level representation of the constraints in view of equations implied by the constraints. In this way it captures many of the well-known constraint programming preprocessing techniques, and more.

Future work will investigate: how to strengthen the implementation of equi-propagation using BDD's and SAT solving techniques, how to improve the compiler implementation using better data-structures for the constraint store (for example applying a CHR based approach for the simplification rules), and how to enhance the underlying constraint language.

References

Ansótegui, C., del Val, A., Dotú, I., Fernández, C., and Manyà, F. 2004. Modeling choices in quasigroup completion: SAT vs. CSP. In *AAAI*, D. L. McGuinness and G. Ferguson, Eds. AAAI Press / The MIT Press, San Jose, California, USA, 137–142.

Asín, R., Nieuwenhuis, R., Oliveras, A., and Rodríguez-Carbonell, E. 2011. Cardinality networks: a theoretical and empirical study. *Constraints 16,* 2, 195–221.

Bailleux, O. and Boufkhad, Y. 2003. Efficient CNF encoding of Boolean cardinality constraints. In *CP*, F. Rossi, Ed. LNCS, vol. 2833. Springer, Kinsale, Ireland, 108–122.

Batcher, K. E. 1968. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*. AFIPS Conference Proceedings, vol. 32. Thomson Book Company, Washington D.C., Atlantic City, NJ, USA, 307–314.

Cadoli, M. and Schaerf, A. 2005. Compiling problem specifications into SAT. *Artificial Intelligence 162,* 1-2, 89–120.

Chen, J. 2010. A new sat encoding of the at-most-one constraint. Workshop on Constraint Modeling and Reformulation (ModRef 2010) `http://www.it.uu.se/research/group/astra/ModRef10/programme.html`.

Codish, M., Lagoon, V., and Stuckey, P. J. 2008. Logic programming with satisfiability. *TPLP 8,* 1, 121–128.

Codish, M. and Zazon-Ivry, M. 2010. Pairwise cardinality networks. In *LPAR (Dakar)*, E. M. Clarke and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 6355. Springer, 154–172.

Crawford, J. M. and Baker, A. B. 1994. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI*, B. Hayes-Roth and R. E. Korf, Eds. Vol. 2. AAAI Press / The MIT Press, Seattle, WA, USA, 1092–1097.

Eén, N. and Sörensson, N. 2003. An extensible SAT-solver. In *SAT*, E. Giunchiglia and A. Tacchella, Eds. LNCS, vol. 2919. Springer, Santa Margherita Ligure, Italy, 502–518.

Eén, N. and Sörensson, N. 2006. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT) 2,* 1-4, 1–26.

Feldman, A., Provan, G., de Kleer, J., Robert, S., and van Gemund, A. 2010. Solving model-based diagnosis problems with max-sat solvers and vice versa. `https://www.phmsociety.org/events/workshop/dx/10/proceedings`.

Frisch, A. M. and Giannaros, P. A. 2010. SAT encodings of the at-most-k constraint: Some old, some new, some fast, some slow. Workshop on Constraint Modeling and Reformulation (ModRef 2010) `http://www.it.uu.se/research/group/astra/ModRef10/programme.html`.

Frutos, A. G., Liu, Q., Thiel, A. J., Sanner, A. M. W., Condon, A. E., Smith, L. M., and Corn, R. M. 1997. Demonstration of a word design strategy for DNA computing on surfaces. *Journal of Nucleic Acids Research 25,* 23, 4748–4757.

Gallian, J. A. 2011. A dynamic survey of graph labeling. *The Electronic Journal of Combinatorics 18.*

Gavanelli, M. 2007. The log-support encoding of csp into sat. In *CP*, C. Bessiere, Ed. LNCS, vol. 4741. Springer, Providence, RI, USA, 815–822.

Gomes, C. P., Selman, B., and Crato, N. 1997. Heavy-tailed distributions in combinatorial search. In *CP*, G. Smolka, Ed. LNCS, vol. 1330. Springer, 121–135.

Jäger, G. 2010. An effective sat encoding for magic labeling. In *CTW*, U. Faigle, R. Schrader, and D. Herrmann, Eds. 97–100.

MacDougall, J., Miller, M., Slamin, M., and Wallis, W. 2002. Vertex-magic total labelings of graphs. *Utilitas Mathematica 61*, 3–21.

Mancini, T., Micaletto, D., Patrizi, F., and Cadoli, M. 2008. Evaluating ASP and commercial solvers on the CSPLib. *Constraints 13,* 4, 407–436.

Metodi, A. 2012. BEE. `http://amit.metodi.me/research/bee/`.

Metodi, A., Codish, M., Lagoon, V., and Stuckey, P. J. 2011. Boolean equi-propagation for optimized SAT encoding. In *CP*, J. H.-M. Lee, Ed. LNCS, vol. 6876. Springer, 621–636.

Metodi, A., Stern, R., Kalech, M., and Codish, M. 2012. Compiling model-based diagnosis to boolean satisfaction. Tech. rep., Department of Computer Science, Ben-Gurion University. `www.cs.bgu.ac.il/~mcodish/Papers/Sources/satMBD.pdf`.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. 2007. MiniZinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming*, C. Bessiere, Ed. LNCS, vol. 4741. Springer-Verlag, 529–543.

Régin, J.-C. 1994. A filtering algorithm for constraints of difference in csps. In *AAAI*, B. Hayes-Roth and R. E. Korf, Eds. AAAI Press / The MIT Press, 362–367.

Reiter, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence 32,* 1, 57–96.

Smith, A., Veneris, A., Ali, M. F., and Viglas, A. 2005. Fault diagnosis and logic debugging using Boolean satisfiability. *IEEE TRANS. ON CAD 24*, 1606–1621.

Soos, M. 2010. CryptoMiniSAT, v2.5.1. `http://www.msoos.org/cryptominisat2`.

Tamura, N., Taga, A., Kitagawa, S., and Banbara, M. 2009. Compiling finite linear CSP into SAT. *Constraints 14,* 2, 254–272.

van Hentenryck, P. 1999. *The OPL Optimization Programming Language.* MIT Press.