Ben-Gurion University of the Negev
Department of Computer Science

Compiling Finite Domain Constraints to SAT with Boolean Equi-Propagation

A Thesis for a Magister Degree in the
Department of Computer Science
Faculty of Natural Sciences
Ben-Gurion University of the Negev

by

Amit Metodi

under the supervision of Professor Michael Codish.

September 2012

Subject: **Compiling Finite Domain Constraints to SAT with Boolean Equi-Propagation**

A Thesis for a Magister Degree in the Department of Computer Science, Faculty of Natural Sciences, Ben-Gurion University of the Negev.

Written by: **Amit Metodi**
Advisor: **Prof. Michael Codish**
Department: **Computer Science**
Faculty: **Natural Sciences**

Ben-Gurion University of the Negev

Author signature: _____ Date: _____

Advisor signature: _____ Date: _____

Dept. Committee Chairman signature: _____ Date: _____

Overview

Boolean SAT solving is about solving hard problems by encoding them to Boolean satisfiability. Such encoding a suitable CNF representation and apply a SAT solver to find a solution.

In this thesis we present a new approch to simplifying CNF generated when translating Constraint Satisfaction Problems (CSP) to Boolean satisfaction problems (SAT). The thesis contains two parts:

In **Part 1**, we present an approach to propagation based SAT encoding, Boolean equi-propagation, where 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.

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 apply to simplify the entire constraint model and facilitate further reasoning on other fragments.

Equi-propagation in combination with partial evaluation and constraint simplification provide the foundation for a powerful approach to SAT based finite domain constraint solving.

We introduce BEE (Ben-Gurion Equi-propagation Encoder), a compiler which encodes finite domain constraint models to CNF. BEE performs transformations to simplify constraints and optimize their encoding to CNF. These optimizations are based primarily on equi-propagation and on partial evaluation.

We demonstrate for a variety of benchmarks that using BEE leads to a considerable reduction in the size of CNF encodings and subsequent speed-ups in SAT solving times.

This part of the thesis extends two papers: the first, [46] presented at CP 2011, which introduced equi-progagation, and the second, [45] presented at ICLP 2012, which introduced BEE.

In **Part 2**, we introduce a novel encoding of Model Based Diagnosis (MBD) to Boolean Satisfaction (SAT) focusing on minimal cardinality diagnosis. The encoding is based on a combination of sophisticated MBD preprocessing algorithms and the application of BEE, a SAT compiler introduced in Part 1, which optimizes the encoding to provide more succinct CNF representations than obtained with previous works. Experimental evidence indicates that our approach is superior to all published algorithms for minimal cardinality MBD. In particular, we can determine, for the first time, minimal cardinality diagnoses for the entire standard ISCAS-85 benchmark. Our results open the way to improve the state-of-the-art on a range of similar MBD problems.

This part of the thesis extends the paper [47], which was presented at AAAI 2012.

Table of Contents

List of Figures

List of Tables

Part I

Equi-Propagation and BEE

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 satisfying assignments of φ_μ correspond to the solutions of μ. Given such an encoding, a SAT solver can be 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. 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 description, a compiler can then provide a low-level executable for the underlying machine. Namely, in our context, a formula for the underlying SAT or SMT solver.

For example, [11] introduce NP-SPEC, a logic-based specification language which allows specifying combinatorial problems in a declarative way. At the core of this system is a component which translates specifications to CNF formula. Similarly Sugar [58] is a SAT-based constraint solver. To solve a finite domain constraint satisfaction problem it is first modeled in a constraint language (also called Sugar) and then encoded to a CNF formula and solved using the MiniSAT solver [21]. MiniZinc [49] is a constraint modeling language that is compiled by a variety of solvers to the low-level target language FlatZinc for which there exist many solvers. In particular, FlatZinc instances are solved by `fzntini` [35] by encoding them to CNF and in `fzn2smt` by encoding to SMT-LIB [5].

Simplifying CNF formulae prior to the application of SAT solving is of the utmost importance and there are a wide range of techniques that can be applied to achieve this goal. See for example [12], [20], [34] and [41], and the references therein. All of these techniques exhibit a clear trade-off between the amount of simplification obtained and the time it requires. Moreover, the stronger techniques become prohibitive when the SAT model involves hundreds of thousands of variables and millions of clauses. So in CNF simplification tools, time limits on simplification techniques are imposed and/or approximations are used.

This part of the thesis presents a new approach to CNF simplification. Typically, a CNF is not a random collection of clauses, but rather has a structure derived from an application or specific problem domain. When SAT solving is applied to encode and solve finite domain constraint problems, the original constraint model is a manifest of this structure. Usually, the constraints are discarded once encoded to CNF. We advocate that maintaining the constraints provides important structural information that can be applied to drive the process of CNF simplification. To be specific, the constraints in a model induce a

partitioning of their CNF encoding to a conjunction of subformulae which we call "portions".

The novelty in our approach to CNF simplification is that instead of considering the CNF as a whole, we assume that it is partitioned into a conjunction of smaller portions. Then simplification is repeatedly applied to individual portions. This faciliates a propagation based process because the simplification of one portion propagates information to all of the portions and this information may trigger further simplification in other portions.

Because portions are typically much smaller than the entire CNF we can effectively apply stronger simplification algorithms. We introduce the notion of equi-propagation. Similar to how unit propagation is about inferring unit clauses which can then be applied to simplify CNF formulae, equi-propagation is about inferring equational consequences between literals (and Boolean constants).

There is a wide body of research on CNF simplification that can be applied to implement equi-propagation which is sometimes called equivalent literal substitution, for example in [30]. Techniques typically involve binary clause based simplifications using, among others, hyper binary resolution and binary implication graphs. See for example, [34] and the references therein. The guiding principle in all of these works is that techniques must be simple and efficient because of the prohibitive size of the CNF to which they must apply.

Our approach is different and we focus on far richer forms of inference not even related to the CNF structure of a formula. At one extreme we apply complete equi-propagation which detects all equivalences implied by a formula. Clearly complete equi-propagation is NP-hard. However, complete equi-propagators are feasible as we apply them only to small portions of the formula. When complete equi-propagation is too slow we consider ad-hoc techniques. All of these forms of equi-propagation have in common that they are not driven by the CNF structure (e.g. binary clauses) but rather by the underlying constraint structure from which a CNF was, or is being, generated.

The rest of this part of the thesis is structured as follows. Section 2 introduces a modeling language for finite domain constraints which consists of just 5 constraint constructs and is sufficient to illustrate the contribution of the thesis. We argue that the constraints in a model induce a natural partition of their CNF encoding to smaller portions and that this partition can be used to drive the simplification of the CNF encoding. Section 3 presents equi-propagation which is the first ingredient for our contribution. Equi-propagation is about learning information that will apply to simplify CNF encodings. Section 4 describes a practical basis for implementing equi-propagation. Section 5 introduces the second ingredient: partial evaluation. Given the information derived using equi-propagation, partial evaluation applies to simplify the constraints and in particular to remove Boolean variables from their CNF encodings. Section 6 describes a tool, called BEE [45], that is based on equi-propagation and partial evaluation. We introduce here our full constraint language which is similar to Sugar and to the subset of FlatZinc relevant for finite domain constraint problems. We also spell out the special treatment of the all-different constraint in BEE. Section 7 demon-

straints the application of BEE. Section 8 presents an experimental evaluation. and Finally Section 9 presents our conclusion.

This part of the thesis extends earlier work presented in [46], which first introduced equi-progagation, and also the BEE tool paper presented in [45]. The BEE tool is available as [42].

2 Constraint Based Boolean Modeling

This section provides the basis for our contribution: a constraint based modeling language, together with a Boolean interpretation for each constraint in the language. This enables us to view a constraint model as a conjunction of Boolean formulae and provides a structure which drives the subsequent encoding to CNF.

We first introduce a simple and small fragment of a typical finite domain constraint based modeling language. This serves to illustrate our approach. Later, in Section 4, we show the full language. We then discuss several options for Boolean representation of integers. In this thesis we adopt a particular unary representation, called the order encoding. Our contribution is independent of this choice, although equi-propagation works well with it. Finally we finish the section so that each of the constraints in the language fragment can be viewed as a Boolean formula, and a constraint model as their conjunction.

2.1 Constraint language fragment

We focus on a small fragment of a typical constraint modeling language. This serves to present the main ideas of the thesis.

(1)	$\texttt{new\_int}(\mathtt{I}, \mathtt{c}_1, \mathtt{c}_2)$	$\mathtt{0} \leq \mathtt{c}_1 \leq \mathtt{I} \leq \mathtt{c}_2$
(2)	$\texttt{int\_neq}(\mathtt{I}_1, \mathtt{I}_2)$	$\mathtt{I}_1 \neq \mathtt{I}_2$
(3)	$\texttt{allDiff}([\mathtt{I}_1, \ldots, \mathtt{I}_n])$	$\bigwedge_{\mathtt{i}<\mathtt{j}} \mathtt{I}_\mathtt{i} \neq \mathtt{I}_\mathtt{j}$
(4)	$\texttt{int\_plus}(\mathtt{I}_1, \mathtt{I}_2, \mathtt{I})$	$\mathtt{I}_1 + \mathtt{I}_2 = \mathtt{I}$
(5)	$\texttt{int\_array\_plus}([\mathtt{I}_1, \ldots, \mathtt{I}_n], \mathtt{I})$	$\mathtt{I}_1 + \cdots + \mathtt{I}_n = \mathtt{I}$

Constraint (1) is about declaring finite domain integer variables in the range $[c_1...c_2]$. For simplicity in the presentation we will further assume that $c_1 \geq 0$. Constraints (2–3) are about difference of integer variables, and constraints (4–5) are about sums of integer variables. As syntactic sugar we also allow writing integer constants in constraints. For example, $\texttt{int\_neq}(\mathtt{I}, \mathtt{5})$ which is short for $\texttt{new\_int}(\mathtt{I}', \mathtt{5}, \mathtt{5})$, $\texttt{int\_neq}(\mathtt{I}, \mathtt{I}')$.

2.2 Modeling Kakuro: an example

A Kakuro puzzle is an $n \times m$ board of black and white cells. The black cells contain hints and the white cells are to be filled by numbers between 1 and 9 (the bound 9 is often generalized by a larger value r). The hints specify constraints on the sums of the values in blocks of white cells to the right and/or below the

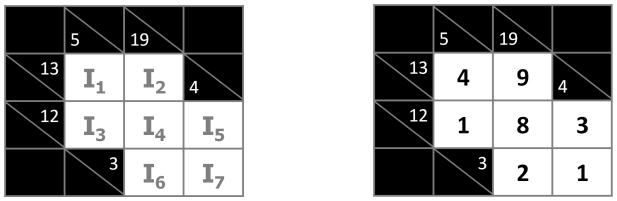


Fig. 1. A 4×4 Kakuro puzzle (right) and its solution (left).

hint. The numbers assigned to the white cells in such a block are required to be "all different". Figure 1 illustrates a 4×4 Kakuro puzzle (left) and its solution (right).

To model a Kakuro puzzle we view it as a set of blocks (of white cells) where each block B is a set of integer variables and is associated with a corresponding integer value, $hint(B)$. Each block B is associated with two constraints: the integers in B must sum to $hint(B)$ and must be all-different. Figure 2 illustrates the constraints corresponding to the Kakuro instance in Figure 1.

$\texttt{new\_int}(\texttt{I}_1, 1, 9)$	$\texttt{int\_array\_plus}([\texttt{I}_1, \texttt{I}_2], 13)$	$\texttt{allDiff}([\texttt{I}_1, \texttt{I}_2])$
$\texttt{new\_int}(\texttt{I}_2, 1, 9)$	$\texttt{int\_array\_plus}([\texttt{I}_1, \texttt{I}_3], 5)$	$\texttt{allDiff}([\texttt{I}_1, \texttt{I}_3])$
$\texttt{new\_int}(\texttt{I}_3, 1, 9)$	$\texttt{int\_array\_plus}([\texttt{I}_3, \texttt{I}_4, \texttt{I}_5], 12)$	$\texttt{allDiff}([\texttt{I}_3, \texttt{I}_4, \texttt{I}_5])$
$\texttt{new\_int}(\texttt{I}_4, 1, 9)$	$\texttt{int\_array\_plus}([\texttt{I}_2, \texttt{I}_4, \texttt{I}_6], 19)$	$\texttt{allDiff}([\texttt{I}_2, \texttt{I}_4, \texttt{I}_6])$
$\texttt{new\_int}(\texttt{I}_5, 1, 9)$	$\texttt{int\_array\_plus}([\texttt{I}_6, \texttt{I}_7], 3)$	$\texttt{allDiff}([\texttt{I}_6, \texttt{I}_7])$
$\texttt{new\_int}(\texttt{I}_6, 1, 9)$	$\texttt{int\_array\_plus}([\texttt{I}_5, \texttt{I}_7], 4)$	$\texttt{allDiff}([\texttt{I}_5, \texttt{I}_7])$
$\texttt{new\_int}(\texttt{I}_7, 1, 9)$		

Fig. 2. Constraints for the Kakuro instance of Figure 1.

2.3 Representing Integers

A fundamental design choice when encoding finite domain constraints concerns the representation of integer variables. In [29] Gavanelli surveys several of the possible choices (the *direct-*, *support-* and *log- encodings*) and introduces the *log-support encoding*. Given a choice of representation constraints are bit-blasted and interpreted as Boolean formulae. We focus for now on the use a unary representation, the so-called, *order-encoding* (see e.g. [15, 3]) which has many nice properties when applied to small finite domains.

In the *order-encoding*, an integer variable X 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 $X \geq i$ so in particular the bit sequence X constitutes a monotonic non-increasing Boolean sequence. For example, the value 3 in the interval $[0, 5]$ is represented in 5 bits as $[1, 1, 1, 0, 0]$.

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 $dom(X) = \{3, 4, 5\}$.

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, $X = [x_1, \underline{x_2, x_2}, x_4, \underline{x_5, x_5}, \underline{x_7, x_7}, x_9]$, signifying that $dom(X) = \{0, 1, 3, 4, 6, 8, 9\}$.

The order-encoding has many additional nice features that can be exploited to simplify constraints and their encodings to CNF. To illustrate one, consider a constraint of the form $\mathtt{A} + \mathtt{B} = \mathtt{5}$ where $\mathtt{A}$ and $\mathtt{B}$ 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: $\mathtt{A} = [\mathtt{a_1}, \ldots, \mathtt{a_5}]$ 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 section 3 we formalize this process of discovering equalities between literals implied by a constraint and using them to simplify CNF encodings.

2.4 Bit Blasting

Given a constraint model and the decision on how to represent finite domain integer variables at the bit level (we chose the order encoding), "bit-blasting" is the process of instantiating integer variables by corresponding bit vectors and interpreting constraints as Boolean formulae.

Each integer variable, $\mathtt{I}$, declared by a constraint of the form $\mathtt{new\_int}(\mathtt{I}, \mathtt{c_1}, \mathtt{c_2})$ where $0 \leq c_1 \leq c_2$ is represented as a bit-vector $\mathtt{I} = [\mathtt{1}, \ldots, \mathtt{1}, \mathtt{X}_{\mathtt{c_1}+\mathtt{1}}, \ldots, \mathtt{X}_{\mathtt{c_2}}]$. So, we may view a constraint model as consisting only of Boolean variables and each constraint c corresponds to a Boolean formula denoted as $[\![c]\!]$. The specific definition of $[\![\cdot]\!]$ is not important. Just for illustration, note that one could define

$$[\![\mathtt{new\_int}(\mathtt{I}, \mathtt{c_1}, \mathtt{c_2})]\!] = \bigwedge_{c_1 \leq i < c_2} (x_{i+1} \rightarrow x_i)$$

where $I = [1, \ldots, 1, X_{c_1+1}, \ldots, X_{c_2}]$ as well as

$$[\![\texttt{int\_neq}(\texttt{I}_1, \texttt{I}_2)]\!] = \bigvee_{i=1}^{n} (x_i \texttt{ xor } y_i)$$

where to simplify presentation we assume that $I_1 = [x_1, \ldots, x_n]$ and $I_2 = [y_1, \ldots, y_n]$ are represented in the same number of bits. The mapping $[\![\cdot]\!]$ extends in the natural way to apply to conjunctions of constraints. So, given a constraint model such as the one in Figure 2, integer variables are instantiated to unary (order encoding) bit vectors and each constraint is viewed as a Boolean formula. The constraint model takes a Boolean representation as the conjunction of these formulae.

3 Boolean Equi-Propagation

In this section we present an approach to propagation based SAT encoding, Boolean equi-propagation, which propagates information about equalities between Boolean literals (and constants). We prove that Boolean equi-propagation is stronger than unit propagation as it determines at least as many fixed literals as unit propagation. We demonstrate the power of Equi-Propagation for Kakuro example. and show that it leads to a considerable reduction in the size of the CNF encoding.

3.1 Boolean Equi-Propagation

Let $\mathcal{B}$ be a set of Boolean variables. A *literal* is a Boolean variable $b \in \mathcal{B}$ or its negation $\neg b$. The negation of a literal ℓ, denoted $\neg\ell$, is defined as $\neg b$ if $\ell = b$ and as b if $\ell = \neg b$. The Boolean constants 1 and 0 represent *true* and *false*, respectively. The set of literals is denoted $\mathcal{L}$ and $\mathcal{L}_{0,1} = \mathcal{L} \cup \{0, 1\}$.

An *assignment*, A, is a partial mapping from Boolean variables to constants, often viewed as the set of literals: $\{\ b \,|\, A(b) = 1\ \} \cup \{\ \neg b \,|\, A(b) = 0\ \}$. For a formula φ and $b \in \mathcal{B}$, we denote by $\varphi[b]$ (likewise $\varphi[\neg b]$) the formula obtained by substituting all occurrences of $b \in \mathcal{B}$ in φ by *true* (*false*). This notation extends in the natural way for sets of literals. We say that A satisfies φ if $\varphi[A]$ evaluates to *true*. A *Boolean Satisfiability (SAT) problem* consists of a Boolean formula φ and determines if there exists an assignment which satisfies φ. The set of (free) Boolean variables that appear in a Boolean formula φ is denoted $vars(\varphi)$.

A *Boolean equality* is a constraint $\ell = \ell'$ where $\ell, \ell' \in \mathcal{L}_{0,1}$. An *equi-formula* E is a set of Boolean equalities understood as a conjunction. The set of Boolean equalities is denoted $\mathcal{L}^{eq}_{0,1}$ and the set of equi-formulae is denoted $\mathcal{E}$.

Equi-propagation is a process of inferring equational consequences from a Boolean formula and given equational information. An *equi-propagator* for a formula φ is an extensive function $\mu_\varphi : \mathcal{E} \to \mathcal{E}$ defined such that for all $E \in \mathcal{E}$,

$$E \subseteq \mu_\varphi(E) \subseteq \{\ e \in \mathcal{L}^{eq}_{0,1} \,|\, \varphi \wedge E \models e\ \}$$

That is, a conjunction of equalities, at least as strong as E, made true by $\varphi \wedge E$. We say that an equi-propagator μ_φ is complete if $\mu_\varphi(E) = \left\{\ e \in \mathcal{L}^{eq}_{0,1} \,\middle|\, \varphi \wedge E \models e\ \right\}$. We denote a complete equi-propagator for φ as $\hat{\mu}_\varphi$. We assume that equi-propagators are monotonic: $E_1 \subseteq E_2 \Rightarrow \mu_\varphi(E_1) \subseteq \mu_\varphi(E_2)$. In particular, this follows, by definition, for complete equi-propagators.

Example 1. Consider the constraint

$$C = \texttt{new\_int}(\texttt{X}, 0, 4)\ \wedge\ \texttt{new\_int}(\texttt{Y}, 0, 4)\ \wedge\ \texttt{int\_neq}(\texttt{X}, \texttt{Y})$$

and its corresponding Boolean representation $\varphi = [\![C]\!]$ on the bit representation where $X = \langle x_1, x_2, x_3, x_4\rangle$ and $Y = \langle y_1, y_2, y_3, y_4\rangle$. Assume the setting where $E = \left\{\ y_1 = 1,\ y_2 = 1,\ y_3 = 0,\ y_4 = 0\ \right\}$ signifying that $Y = 2$. Then, $\hat{\mu}_\varphi(E) = E \cup \{x_2 = x_3\}$ indicating that $X \neq 2$. □

The following theorem states that complete equi-propagation is at least as powerful as unit propagation.

Theorem 1. *Let $\hat{\mu}_\varphi$ be a complete equi-propagator for a Boolean formula φ. Then, $\hat{\mu}_\varphi(E)$ determines at least as many fixed literals as unit propagation for any clausal representation of φ using the equations in E.*

Proof. Let φ be a Boolean formula, E an equi-formula, and let C_φ and C_E be any clausal representations of φ and of E respectively. Clearly $\varphi \models C_\varphi$ and $E \models C_E$. Let b be a positive literal determined by unit propagation of $C_\varphi \cup C_E$. Then by correctness of unit propagation, $C_\varphi \cup C_E \models b$. Hence, $\varphi \wedge E \models b$ and thus $\hat{\mu}_\varphi(E) \models b = 1$. The case for a negative literal $\neg b$ is the same, except that we infer $b = 0$.

The following example illustrates that equi-propagation can be more powerful than unit propagation.

Example 2. Consider $\varphi = (x_1 \leftrightarrow x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$. The clausal representation is $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ and no unit propagation is possible, since there are no unit clauses. Equi-propagation (with no additional equational information) gives: $\hat{\mu}_\varphi(\emptyset) = \{x_1 = 1, x_2 = 1, x_3 = 0\}$

Boolean Unifiers: It is sometimes convenient to view an equi-formula E in a generic "solved-form" as a Boolean substitution, θ_E, which is a (most general) unifier for the equations in E. Boolean substitutions generalize assignments in that variables can be bound also to literals. A Boolean *substitution* is an idempotent mapping $\theta : \mathcal{B} \to \mathcal{L}_{0,1}$ such that $dom(\theta) = \left\{\ b \in B \,\middle|\, \theta(b) \neq b\ \right\}$ is finite and for every $b \in \mathcal{B}$, $\theta(b) \neq \neg b$. A Boolean substitution, θ, is viewed as the set $\theta = \left\{\ b \mapsto \theta(b) \,\middle|\, b \in dom(\theta)\ \right\}$. We can apply θ to another substitution θ', to obtain substitution $(\theta \cdot \theta') = \left\{\ b \mapsto \theta(\theta'(b)) \,\middle|\, b \in dom(\theta) \cup dom(\theta')\ \right\}$. A *unifier* for equi-formula E is a substitution θ such that $\models \theta(e)$, for each $e \in E$. A *most-general unifier* for E is a substitution θ such that for any unifier θ' of E, there exists substitution γ where $\theta' = \gamma \cdot \theta$.

Example 3. Consider the equi-formula $E \equiv \{b_1 = \neg b_2, \neg b_3 = \neg b_4, b_5 = b_6, b_6 = b_4, b_7 = 1, b_8 = \neg b_7\}$ then a unifier θ for E is $\{b_2 \mapsto \neg b_1, b_4 \mapsto b_3, b_5 \mapsto b_3, b_6 \mapsto b_3, b_7 \mapsto 1, b_8 \mapsto 0\}$. Note that $\theta(E)$ is the trivially true equi-formula $\{b_1 = \neg\neg b_1, \neg b_3 = \neg b_3, b_3 = b_3, b_3 = b_3, 1 = 1, 0 = \neg 1\}$.

□

Consider the enumeration $\mathcal{L}_{0,1} = \{0, 1, \neg b_1, b_1, \neg b_2, b_2, \ldots\}$ and let $\prec$ be the total (strict) order on $\mathcal{L}_{0,1}$ such that $0 \prec 1 \prec \neg b_1 \prec b_1 \prec \neg b_2 \prec b_2 \cdots$. We define a canonical most-general unifier for any satisfiable equi-formula E: $\texttt{unify}_E = \lambda b.\, \min\left\{\ \ell \in \mathcal{L}_{0,1} \,\middle|\, E \models b = \ell\ \right\}$. We can compute $\texttt{unify}_E$ in almost linear (amortized) time using a variation of the union-find algorithm [59].

Example 4. For the equi-formula E and substitution θ from Example 3 we have that $\texttt{unify}_E = \theta$.

□

The following proposition provides the foundation for equi-propagation based Boolean simplification. It allows us to apply equational information to simplify a given formula. In particular, if E is an equi-formula about literals occurring in φ then $\texttt{unify}_E(\varphi)$ is smaller than φ in that it contains less variables.

Proposition 1. *Let φ be a Boolean formula and $E \in \mathcal{E}$ be a satisfiable equi-formulae. Then,*

a. $\varphi \wedge E \ \leftrightarrow\ \texttt{unify}_E(\varphi) \wedge E$*;*
b. $\varphi \wedge E$ *is satisfiable if and only if* $\texttt{unify}_E(\varphi)$ *is satisfiable; and*
c. if σ *is a satisfying assignment for* $\texttt{unify}_E(\varphi)$ *then* $\sigma \circ \texttt{unify}_E$ *is a satisfying assignment for* $\varphi \wedge E$*.*

Proof. (a) Let $\theta = \texttt{unify}_E$ and assume that σ is a satisfying assignment of E, then we can view σ as a substitution, and as a unifier of E. Hence, since θ is a most general unifier, there exists a substitution γ such that $\sigma = \gamma \cdot \theta$. Clearly $\gamma(b) = \sigma(b)$ for all variables b in the range of θ. Hence, σ and γ agree on all variables in $\theta(\varphi)$ which implies that $\sigma(\theta(\varphi)) = \gamma(\theta(\varphi))$ meaning that $\sigma(\theta(\varphi)) = \sigma(\varphi)$. So, σ is a satisfying assignment of $\theta(\varphi) \wedge E$ if and only if σ is a satisfying assignment of $\varphi \wedge E$. (b) The $(\rightarrow)$ direction follows from (a) and the $(\leftarrow)$ direction from (c). (c) Assume σ is a satisfying assignment of $\texttt{unify}_E(\varphi)$. Clearly $\sigma \cdot \texttt{unify}_E$ satisfies φ by construction. Also $\sigma \cdot \texttt{unify}_E$ satisfies E since $\texttt{unify}_E(E)$ is trivial. Hence $\sigma \cdot \texttt{unify}_E$ is a satisfying assignment of $\varphi \wedge E$. □

The Equi-Propagation Process presented now is a central theme in this thesis: Let $\Phi = \varphi_1 \wedge \cdots \wedge \varphi_n$ be a partitioning of a Boolean formula to n portions, let $\mu_{\varphi_1}, \ldots, \mu_{\varphi_n}$ be corresponding equi-propagators, and take initial $E = \emptyset$. Satisfiability of Φ can be determined as follows:

1. So long as possible, select φ_i such that $\mu_{\varphi_i}(E) \supsetneq E$ and update $E = \mu_{\varphi_i}(E)$.
2. Finally, when the equi-propagators apply no more, check if $\texttt{unify}_E(\Phi)$ is satisfiable.

3. If η is a satisfying assignment for $\mathtt{unify}_E(\Phi)$ then $\mathtt{unify}_E \circ \eta$ is a satisfying assignment for Φ.

We typically apply this equi-propagation theme to the Boolean representation $\Phi = \varphi_1 \wedge \cdots \wedge \varphi_n$ of a constraint model $C = C_1 \wedge \cdots \wedge C_n$ where $\varphi_i = [\![C_i]\!]$. Here we require that each C_i is a "small" conjunction of constraints. Typically, the integer variables referred to in each C_i are also declared in C_i (sometimes this requires duplicating the variable declarations). For an individual constraint c we denote by c^+ the conjunction of constraints including c and the declarations for integer variables it refers to. The specifics of these declarations will be clear from the context.

Example 5. Let C be the following constraint model:

$$C = \begin{pmatrix} \mathtt{new\_int(X,1,3)} \ \wedge \ \mathtt{new\_int(Y,1,3)} \ \wedge \ \mathtt{new\_int(Z,1,3)} \ \wedge \\ \mathtt{int\_plus(X,Y,3)} \ \wedge \ \mathtt{int\_plus(Y,Z,4)} \ \wedge \ \mathtt{int\_neq(Y,Z)} \end{pmatrix}$$

We have

1. $\mathtt{int\_plus}^+(\mathtt{X},\mathtt{Y},3) = \mathtt{int\_plus}(\mathtt{X},\mathtt{Y},3) \wedge \mathtt{new\_int}(\mathtt{X},1,3) \wedge \mathtt{new\_int}(\mathtt{Y},1,3)$,
2. $\mathtt{int\_plus}^+(\mathtt{Y},\mathtt{Z},4) = \mathtt{int\_plus}(\mathtt{Y},\mathtt{Z},4) \wedge \mathtt{new\_int}(\mathtt{Y},1,3) \wedge \mathtt{new\_int}(\mathtt{Z},1,3)$,
3. $\mathtt{int\_neq}^+(\mathtt{Y},\mathtt{Z}) = \mathtt{int\_neq}(\mathtt{Y},\mathtt{Z}) \ \wedge \mathtt{new\_int}(\mathtt{Y},1,3) \ \wedge \ \mathtt{new\_int}(\mathtt{Z},1,3)$.

As a basis for equi-propagation we take $\Phi = \varphi_1 \ \wedge \ \varphi_2 \ \wedge \ \varphi_3$ where $\varphi_1 = [\![\mathtt{int\_plus}^+(\mathtt{X},\mathtt{Y},3)]\!]$, $\varphi_2 = [\![\mathtt{int\_plus}^+(\mathtt{Y},\mathtt{Z},4)]\!]$, and $\varphi_3 = [\![\mathtt{int\_neq}^+(\mathtt{Y},\mathtt{Z})]\!]$. Denoting $X = \langle 1, x_2, x_3\rangle$, $Y = \langle 1, y_2, y_3\rangle$, and $Z = \langle 1, z_2, z_3\rangle$ and applying corresponding complete equi-propagators and starting with $E_0 = \emptyset$ we have:

1. $E_1 = \hat{\mu}_{\varphi_1}(E_0) = E_0 \cup \{x_3 = 0, y_3 = 0, x_2 = \neg y_2\}$;
2. $E_2 = \hat{\mu}_{\varphi_2}(E_1) = E_1 \cup \{z_2 = 1, y_2 = \neg z_3\}$;
3. $E_3 = \hat{\mu}_{\varphi_3}(E_2) = E_2 \cup \{y_2 = 0\}$.

At this point equi-propagation applies no more, and $\mathtt{unify}_{E_3} = \{x_2 \mapsto 1, x_3 \mapsto 0, y_2 \mapsto 0, y_3 \mapsto 0, z_2 \mapsto 1, z_3 \mapsto 1\}$. Now, $\mathtt{unify}_{E_3}(\Phi)$ is a tautology (all of the Boolean variables are determent by equi-propagation).

□

The following theorem clarifies that the order in which equi-propagators are applied in the equi-propagation process does not influence the final result.

Theorem 2. *The equi-propagation process is confluent.*

Proof. Let $\Phi = \varphi_1 \wedge \cdots \wedge \varphi_n$ be a Boolean formula and $\mu_{\varphi_1}, \ldots, \mu_{\varphi_n}$ corresponding equi-propagators. Let $E_1 = \mu_{\varphi_{i_r}}(\mu_{\varphi_{i_{r-1}}}(\ldots \mu_{\varphi_{i_1}}(\emptyset)\ldots))$ and $E_2 = \mu_{\varphi_{j_s}}(\mu_{\varphi_{j_{s-1}}}(\ldots \mu_{\varphi_{j_1}}(\emptyset)\ldots))$ be two different applications of the equi-propagation process. So by construction, for each of the given equi-propagators, we have a property ($\star$): $\mu_{\varphi_i}(E_1) = E_1$ and $\mu_{\varphi_i}(E_2) = E_2$.

Now assume, in contradiction, that $E_1 \subsetneq E_2$. Let us focus on the first step in the equi-propagation process leading to E_2 that introduced an equation $e \in E_2$ not introduced to E_1: So, there exists an $\ell < s$ such that $E =$

$\mu_{\varphi_{j_\ell}}(\mu_{\varphi_{j_{\ell-1}}}(\ldots\mu_{\varphi_{j_1}}(\emptyset)\ldots)) \subseteq E_1$ and $e \in \mu_{\varphi_{\ell+1}}(E)$ but $e \notin E_1$. But, if $E \subseteq E_1$, then by the monotonicity of $\mu_{\varphi_{\ell+1}}$, we have that $\mu_{\varphi_{\ell+1}}(E) \subseteq \mu_{\varphi_{\ell+1}}(E_1)$ and hence $e \in \mu_{\varphi_{\ell+1}}(E_1)$ in contradiction to the construction with property ($\star$).

The following proposition provides an alternative, more efficient to implement, definition for complete equi-propagation.

Proposition 2. *Let φ be a Boolean formula then*

$$\bar{\mu}_\varphi(E) = E \cup \left\{\ e \in \mathcal{L}^{eq}_{0,1} \,\middle|\, \mathtt{unify}_E(\varphi) \models e\ \right\}$$

is such that $\hat{\mu}_\varphi(E) \leftrightarrow \bar{\mu}_\varphi(E)$, that is it implements a complete equi-propagator for φ.

Proof. For the first direction, ($\Rightarrow$): By definition, we have that $\hat{\mu}_\varphi(E) \rightarrow E$. We also have $\hat{\mu}_\varphi(E) \rightarrow \left\{\ e \,\middle|\, \mathtt{unify}_E(\varphi) \models e\ \right\}$ because by Proposition 1(a) $\varphi \wedge E \models \mathtt{unify}_E(\varphi)$. So, $\hat{\mu}_\varphi(E) \rightarrow \bar{\mu}_\varphi(E)$. For the other direction, ($\Leftarrow$): Let $e \in \bar{\mu}_\varphi(E)$. If $e \in E$ then the proof is straightforward. Otherwise, let $\mathtt{unify}_E(\varphi) \models e$ and assume in contrary that $e \notin \hat{\mu}_\varphi(E)$, or in other words that $\varphi \wedge E \not\models e$. This means that there exists an assignment σ that satisfies $\varphi \wedge E$ but does not satisfy e. By Lemma 1(a), σ also satisfies $\mathtt{unify}_E(\varphi) \wedge E$ and in particular σ satisfies $\mathtt{unify}_E(\varphi)$. From our assumption that $\mathtt{unify}_E(\varphi) \models e$ we now have that σ satisfies e. Contradiction. □

Computing $\bar{\mu}_\varphi$ is considerably more efficient than $\hat{\mu}_\varphi$ since we can simply examine the formula φ after the application of $\mathtt{unify}_E$ to determine new Boolean equality consequences.

3.2 The Power Of Equi-Propagation

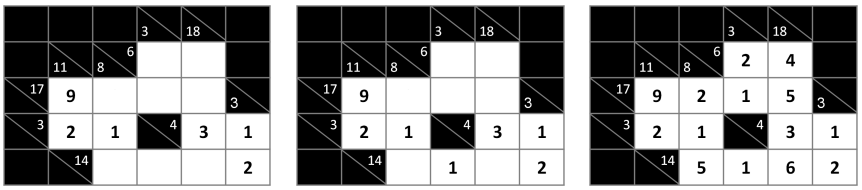
To illustrate the impact of Equi-Propagation we come back to the Kakuro example from Section 2.2. In fact solving such puzzles via SAT encodings is quite easy, with and without equi-propagation. So the example should only be viewed as illustrating the impact of equi-propagation on the size of the encoding.

We consider, as a baseline for this discussion, the following Boolean representation derived from a constraint model where the declarations which are not specified explicitly are of the form $\mathtt{new\_int}(I, 1, h)$ where h is the smallest hint for a block that includes I or 9 if that is smaller.

$$\Phi_1 = \bigwedge_{\substack{\{I_1,\ldots,I_k\} \in Blocks \\ 1 \le i < j \le k}} [\![\mathtt{int\_neq}^+(I_i, I_j)]\!] \ \wedge \bigwedge_{B \in Blocks} [\![\mathtt{int\_array\_sum}^+(B, hint(B))]\!]$$

Notice that there is one "`int_neq`" conjunct for each pair of white cells in the same block, and one "`int_array_sum`" conjunct for each block.

Applying the equi-propagation process to Φ_1 with complete equi-propagators determines six integer values as depicted in Figure 3(a).



a. EP scheme #1 b. EP scheme #2 c. EP scheme #3

Fig. 3. Applying Complete Equi-Propagation to a Kakuro Instance

Figure 3(b) illustrates the impact of applying the equi-propagation process where the equi-propagators are for `allDiff` constraints instead of for the individual `int_neq` constraints. This determines seven integer variables and is formalized taking the following Boolean representation of the constraint model (and introducing an equi-propagator for each conjunct).

$$\Phi_2 = \bigwedge_{B \,\in\, Blocks} [\![\texttt{allDiff}^+(B)]\!] \quad \wedge \quad \bigwedge_{B \,\in\, Blocks} [\![\texttt{int\_array\_sum}^+(B, hint(B))]\!]$$

Figure 3(c) illustrates the impact of applying the equi-propagation process where the equi-propagators are for pairs, each consisting of an `allDiff` constraint together with its corresponding sum constraint. This form of equi-propagation is most powerful. It fixes integer values for all of the white cells (in this example). We stress that equi-propagation reasons only about equalities between Boolean literals and constants. Here we take the model as:

$$\Phi_3 = \bigwedge_{B \,\in\, Blocks} \left([\![\texttt{allDiff}^+(B)]\!] \;\wedge\; [\![\texttt{int\_array\_sum}^+(B, hint(B))]\!] \right)$$

To further demonstrate the impact of equi-propagation, Table 1 provides data for 15 additional instances,[1] categorized as: "easy", "medium" and "hard". The first two columns in the table indicate the instance category and ID. From the five columns headed "Integer Variables", the first four specify the number of unassigned white cells in the initial stage and after each of the three complete equi-propagation processes described above. From the five columns headed "Boolean variables", the first four indicate the corresponding information regarding the number of Boolean variables in the bit representations of the integers. So, the smaller the number in the table, the more variables have been removed due to equi-propagation. In particular, the EP_3 scheme completely solves 9 of the 15 instances. The two columns titled BEE show the corresponding information obtained using a weaker form of equi-propagation that is described in Section 4 below. The last row of the table indicates the time it takes to perform equi-propagation (in seconds) using each of the three schemes. We will come back to discuss this later after detailing how equi-propagation is performed. The results

[1] Instances available from `http://4c.ucc.ie/~hcambaza/page1/page7/page7.html` (generated by Helmut Simonis).

in the table indicate the clear benefit in performing equi-propagation based on coarser portions of the model.

	ID	Integer Variables					Boolean Variables				
		init	EP_1	EP_2	EP_3	BEE	init	EP_1	EP_2	EP_3	BEE
easy	168	484	439	280	0	385	3872	1440	843	0	1170
	169	467	456	440	0	440	3736	1823	1682	0	1692
	170	494	485	469	0	469	3952	1961	1798	0	1805
	171	490	406	393	0	422	3920	1280	1148	0	1341
	172	506	495	484	0	492	4048	1676	1573	0	1634
medium	188	476	461	455	0	461	3808	1939	1915	0	1934
	189	472	437	425	62	449	3776	2017	1911	81	1976
	190	492	481	480	0	480	3936	1998	1920	0	1936
	191	478	452	448	161	448	3824	1864	1821	197	1828
	192	499	481	478	136	478	3992	2455	2417	214	2420
hard	183	490	365	345	0	371	3920	1151	1059	0	1168
	184	506	489	484	23	486	4048	1613	1495	21	1545
	185	482	482	455	206	467	3856	2181	2111	220	2144
	186	472	466	454	0	466	3776	2115	2062	0	2086
	187	492	475	473	69	473	3936	1991	1959	48	1960
	Avg. comp. time in sec.						3.739	2.981	0.916	0.477	

Table 1. Applying complete Equi-Propagation on Kakuro encoding

3.3 Implementing Equi-Propagators

To implement complete equi-propagators we need to infer Boolean equalities implied by a given Boolean formula, φ, and equi-formula, E. Based on Proposition 2, it is sufficient to test for the condition

$$\texttt{unify}_E(\varphi) \models (\ell_1 \leftrightarrow \ell_2) \tag{1}$$

We consider three techniques: using a SAT solver, using BDD's, and using ad-hoc rules applied to the Boolean representations of individual constraints.

It is straightforward to implement a complete equi-propagator using a SAT solver. To test Condition (1) we consider the formula $\psi = \varphi \wedge (\ell_1 \not\leftrightarrow \ell_2)$. If ψ is not satisfiable, then Condition (1) holds. In this way, Condition (1) can be checked for all relevant equations involving variables from $\texttt{unify}_E(\varphi)$ (and constants 0,1). A major obstacle with this SAT-based approach is that testing for a single equivalence, $\ell_1 \leftrightarrow \ell_2$, is at least as hard as testing for the satisfiability of φ. In fact testing for unsatisfiability is typically more expensive. Hence the importance of our assumption that φ is only a small fragment of the CNF of interest. In practice SAT based equi-propagation is surprisingly fast. For illustration, in the last row of Table 1 the average times for SAT-based complete equi-propagation are indicated in the columns EP_1, EP_2, and EP_3. It is interesting to observe

that the strongest technique, using EP_3, is the fastest. This is because there are fewer (but larger) conjuncts and hence less queries to the SAT solver.

When implementing complete equi-propagator using binary decision diagrams (BDDs) we test Condition(1) using a standard BDD containment test e.g., "`bddLeq`" in [55]. Just as in the SAT-based approach, this test is performed for all relevant equations involving variables from $\texttt{unify}_E(\varphi)$ (and constants 0,1).

Example 6. Consider the BDD shown in Figure 4(a) which represents the formula: $\varphi \equiv \texttt{new\_int}(A, 0, 3) \wedge \texttt{new\_int}(B, 0, 3) \wedge \texttt{int\_neq}(A, B)$. Figure 4(b) depicts the The BDD for $\texttt{unify}_E(\varphi)$ where $E = \{B_1 = 1,\ B_2 = 1,\ B_3 = 0\ \}$. Here it is easy to see that equi-propagation determines that $A_2 = A_3$, and we denote $E' = E \cup \{A_2 = A_3\}$. Figure 4(c) shows the simplified BDD for $\texttt{unify}_{E'}(\varphi)$.

A major obstacle with this BDD-based approach concerns the size of the for the formula $\texttt{unify}_E(\varphi)$. For some constraints, the corresponding BDD is guaranteed to be polynomial (in the size of the constraint). The following result holds for an arbitrary constraint φ, so it also holds for $\texttt{unify}_E(\varphi)$.

Proposition 3. *Let c be a constraint about k integer variables each represented with n bits in the order encoding. Then, the number of nodes in the BDD representing $[\![c]\!]$ is bound by $O(n^k)$.*

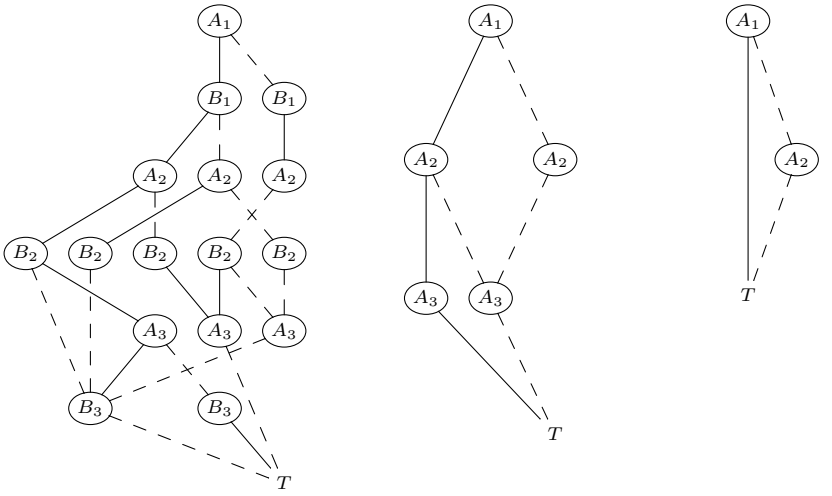
Proof. (Sketch) There are only $n + 1$ legitimate states for each n bit unary variable, and the BDD cannot have more nodes than possible states. □

Constraints like `new_int`, `int_neq`, and `int_plus` involve at most 3 integer variables and hence their BDD-based complete equi-propagators are polynomially bounded. However, this is not the case for global constraints such as `allDiff` and `int_array_plus` where the arity is not fixed. Moreover, it is well known that the `allDiff` constraint does not have a polynomial sized BDD [8].

Given the potential exponential runtime when performing SAT-based equi-propagation, and the potential exponential size of BDD-based equi-propagators, we consider a third approach where we implement equi-propagation 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 are not detected — the implementation is fast, and works well in practice. This is the topic of the next section.

4 Ad-hoc equi-propagation

We consider a rule based approach to define equi-propagators. The definition is given as a set of ad-hoc rules specified for each type of constraint. While this approach is not complete — there are Boolean equalities implied by a constraint that will not be detected — the approach leads to a fast implementation, and works well in practice. The novelty is that the approach is not CNF based, as in previous works, but rather driven by the bit blasted constraints that are to



(a) BDD for $\texttt{int\_neq}_u(A, B)$ **(b)** Simpl'd wrt B=$[1, 1, 0]$ **(c)** Simpl'd wrt A_2=A_3

Fig. 4. BDDs for (a) $\varphi \equiv \texttt{new\_int}_3(A, [0, 3]) \wedge \texttt{new\_int}_3(B, [0, 3]) \wedge \texttt{int\_neq}(A, B)$ (b) $\texttt{unify}_E(\varphi)$ where $E = \{B_1{=}1, B_2{=}1, B_3{=}0\}$ and (c) $\texttt{unify}_{E'}(\varphi)$ where $E' = E \cup \{A_2{=}A_3\}$. Full (dashed) lines correspond to true (false) edges. Target "F" node is omitted for brevity.

be encoded to CNF. Our presentation focuses on the case where finite domain integers are represented in the order encoding. For an integer $X = \langle x_1, \ldots, x_n \rangle$, we often write: $X \geq i$ to denote the equation $x_i = 1$, $X < i$ to denote 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 $x_i = 1, x_{i+1} = 0$. Moreover, to simplify notation when specifying the rules below, we view $X = \langle x_1, \ldots, x_n \rangle$ as a larger vector 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. Basically this facilitates the specification of the "end cases" in our formalism. We now consider each of the 5 constraints in the language fragment presented in Section 2.

(1) The two rules, on the right, derive from the monotonicity in the order encoding representation. These basically correspond to unit propagation, but at the constraint level.

$c = \texttt{new\_int}(\langle x_1, \ldots, x_n \rangle, 0, n)$	
if in $\texttt{E}$	then add in $\mu_\texttt{c}(\texttt{E})$
$x_i = 1$	$x_1 = 1, \ldots, x_{i-1} = 1$
$x_i = 0$	$x_{i+1} = 0, \ldots, x_n = 0$

(2) The first rule on the right consider cases when X is a constant (for the symmetric case replace X and Y). The other two rules capture templates that commonly arise in the equi-propagation process. To

$c = \texttt{int\_neq}(X, Y)$ where $X = \langle x_1, \ldots, x_n \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$	
if in $\texttt{E}$	then add in $\mu_\texttt{c}(\texttt{E})$
$X = i$	$Y \neq i$
$x_i = y_{i+1}$, $y_i = x_{i+1}$	$X \neq i, Y \neq i$
$x_i = \neg y_{i+1}$, $y_i = \neg x_{i+1}$	$X \neq i, Y \neq i$

illustrate the justification of the third rule consider all possible truth values for the variables x_i and x_{i+1}: (a) If $x_i = 0$ and $x_{i+1} = 1$ 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 $x_i = 1$ and $x_{i+1} = 0$ 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_i and x_{i+1} are those where $x_i = x_{i+1}$.

(3) On the right we illustrate a single rule for the `allDiff` constraint which considers Hall sets of size 2. Here each Z_i represents an integer in the order encoding and we focus on the case when Z_1 and Z_2 are restricted by the equations in E to take only two possible values, i or j. This can be expressed in E because $\langle x_1, \ldots, x_n \rangle \in \{i, j\}$ (for $i < j$) means that $x_k = 1$ for $k < i$, $x_k = x_{k+1}$ for $i \leq k < j$, and $x_k = 0$ for $j < k \leq n$. $Z_1 \neq Z_2$ then means adding the single equation $x_i = \neg y_i$ (because Z_1 and Z_2 can take only two values). In addition to this rule, we apply the rules for $\texttt{int\_neq}(Z_i, Z_j)$ for each pair of integers Z_i and Z_j in the constraint.

$c = \texttt{allDiff}([Z_1, Z_2, Z_3, \ldots, Z_n])$	
if in `E`	then add in $\mu_{\texttt{c}}(\texttt{E})$
$Z_1, Z_2 \in \{i, j\}$	$Z_1 \neq Z_2,\ Z_k \neq i$ $Z_k \neq j\ \ (k > 2)$

(4) The first four rules on the right capture the standard propagation behavior for interval arithmetics. 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.

$c = \texttt{int\_plus}(X, Y, Z)$ where $X = \langle x_1, \ldots, x_n \rangle$, $Y = \langle y_1, \ldots, y_m \rangle$, and $Z = \langle z_1, \ldots, z_{n+m} \rangle$	
if in `E`	then add in $\mu_{\texttt{c}}(\texttt{E})$
$X \geq i,\ Y \geq j$	$Z \geq i + j$
$X < i,\ Y < j$	$Z < i + j - 1$
$Z \geq k,\ X < i$	$Y \geq k - i$
$Z < k,\ X \geq i$	$Y < k - i$
$X = i$	$z_{i+1} = y_1, \ldots, z_{i+m} = y_m$
$Z = k$	$x_1 = \neg y_k, \ldots, x_k = \neg y_1$

(5) There are no special ad-hoc rules for equi-propagation of an `int_array_plus` constraint. These are simply viewed as their decomposition to a set of `int_plus` constraints. Then simplification is performed at that level using the rules for `int_plus`.

Example 7 (ad-hoc equi-propagation). Consider the following constraint model[2] where we denote $\texttt{X} = [\texttt{x}_1, \ldots, \texttt{x}_9]$, $\texttt{Y} = [\texttt{y}_1, \ldots, \texttt{y}_9]$ and $\texttt{Z} = [\texttt{z}_1, \ldots, \texttt{z}_{18}]$ and assume an initial equi-formula E_0 to specify that integer variable $Z = 4$:

$$C = \begin{pmatrix} \texttt{new\_int}(\texttt{X}, 0, 9) \ \wedge\ \texttt{new\_int}(\texttt{Y}, 0, 9) \ \wedge\ \texttt{new\_int}(\texttt{Z}, 0, 18) \ \wedge \\ \texttt{int\_plus}(\texttt{X}, \texttt{Y}, \texttt{Z}) \ \wedge\ \texttt{int\_neq}(\texttt{X}, \texttt{Y}) \end{pmatrix}$$

Figure 5 illustrates, step-by-step, the equi-propagation process on C using the ad-hoc rules defined above. Each step corresponds to the application of one of the above defined adhoc equi-propagation rules as indicated by the label on the transition. At each stage we illustrate the derived equations (top part) and their application (as a unifier) to the state variables `X`, `Y` and `Z` (lower part). □

[2] From the context of the Kakuro example of Section 2.2.

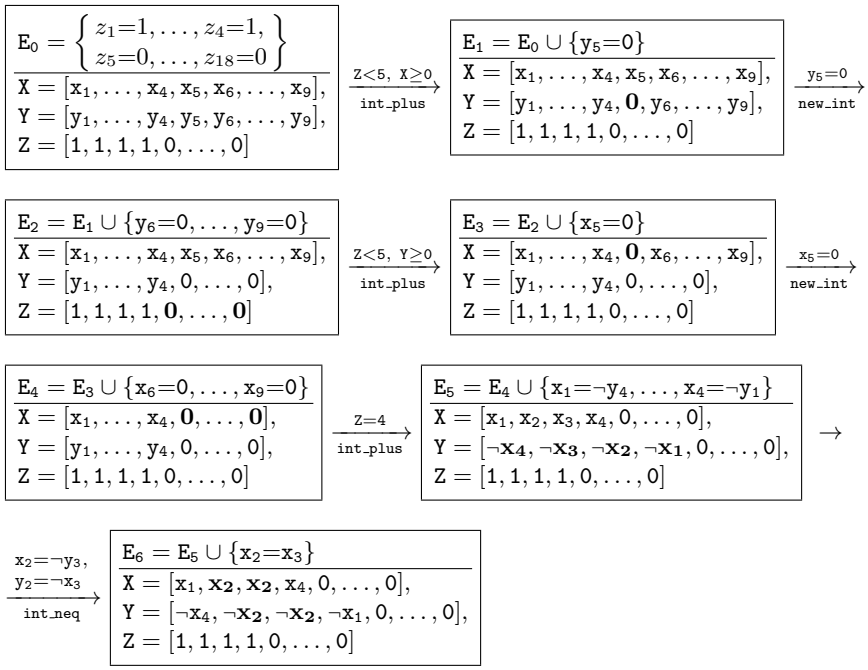


Fig. 5. Ad-hoc equi-propagation described in Example 7

To summarize, let us come back Table 1. The numbers presented in the two columns headed "BEE" specify the number of variables remaining after application of ad-hoc equi-propagation. We also observe that our definition of ad-hoc equi-propagation is trivially monotonic and hence also confluent.

5 Constraint Model Partial Evaluation

Partial evaluation, together with equi-propagation, is the second important component in our approach to compile constraint models to CNF. Partial evaluation is about simplifying a given constraint model in view of information that becomes available due to equi-propagation. Typically, in the constraint simplification process, we apply alternating steps of equi-propagation and partial evaluation. Examples of partial evaluation include constant elimination and removing constraints which are tautologies. In this section we detail the partial evaluation rules that apply for the five constraint types defined in the language fragment presented in Section 2.

(1) A $\texttt{new\_int}(\texttt{I}, \texttt{c}_1, \texttt{c}_2)$ constraint specifies that an integer $\texttt{I} = [\texttt{x}_1, \ldots, \texttt{x}_\texttt{n}]$ is represented in the order encoding and in particular that the corresponding bit sequence is sorted (not increasing). We denote this as $\texttt{ordered}([\texttt{x}_1, \ldots, \texttt{x}_\texttt{n}])$. Partial evaluation focuses on this aspect of the constraint and ignores the bounds $\texttt{c}_1$, $\texttt{c}_2$ specified in the constraint. The table on the right specifies four simplification rules that apply. The first rule identifies tautologies, the second and third rules remove leading ones and tailing zeros, and the fourth removes (one of two) equated bits.

$c = \texttt{ordered}([x_1, \ldots, x_n])$ (new_int)	
if	then replace with
$n \leq 1$	true
$x_1 = 1$	$\texttt{ordered}([\cancel{1}, x_2 \ldots, x_n])$
$x_n = 0$	$\texttt{ordered}([x_1, \ldots, x_{n-1}, \cancel{0}])$
$x_i = x_{i+1}$	$\texttt{ordered}([x_1, \ldots, x_i, \cancel{x_{i+1}}, \ldots, x_n])$

(2) These simplification rules on the right are symmetric when replacing the role of X and Y. The first two rules identify tautologies. The third rule is about X and Y which have an equal bit at position i. The last two rules are about removing leading ones and trailing zeroes and are illustrated by the following example.

$c = \texttt{int\_neq}(X, Y)$ where $X = \langle x_1, \ldots, x_n \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$	
if	then replace with
$X = i, Y \neq i$	true
$x_i = \neg y_i$	true
$x_i = y_i$	$\texttt{int\_neq}([x_1, \ldots, \cancel{x_i}, \ldots, x_n], [y_1, \ldots, \cancel{y_i}, \ldots, y_n])$
$X \geq i \geq 2$	$\texttt{int\_neq}([1, x_{i+1}, \ldots, x_n], [y_i, y_{i+1}, \ldots, y_n])$
$X \leq i$	$\texttt{int\_neq}([x_1, \ldots, x_i, 0], [y_1, \ldots, y_i, y_{i+1}])$

Example 8. The following are two steps of partial evaluation, for a `int_neq` constraint, first removing leading ones, then removing trailing zeroes.

$$\left[\begin{array}{l}\texttt{int\_neq}(\\ \quad[\texttt{x}_1, \ldots, \texttt{x}_4, 0, 0, 0],\\ \quad[1, 1, 1, \texttt{y}_4, \ldots, \texttt{y}_7])\end{array}\right] \xrightarrow[\texttt{int\_neq}]{P.E} \left[\begin{array}{l}\texttt{int\_neq}(\\ \quad[\texttt{x}_3, \texttt{x}_4, 0, 0, 0],\\ \quad[1, \texttt{y}_4, \ldots, \texttt{y}_7])\end{array}\right] \xrightarrow[\texttt{int\_neq}]{P.E} \left[\begin{array}{l}\texttt{int\_neq}(\\ \quad[\texttt{x}_3, \texttt{x}_4, 0],\\ \quad[1, \texttt{y}_4, \texttt{y}_5])\end{array}\right]$$

(3) Four rules for simplifying `allDiff` constraints are illustrated on the right. The first, is about detecting tautologies. The second, identifies cases when one of the integers in the constraint (assume Z_1) has a domain disjoint from all of the others. This rule also captures the case when Z_1 is a constant. The third rule removes a Hall set of size 2 (assume $\{Z_1, Z_2\}$) from the constraint. Note that the corresponding equi-propagation rule detects that the values of $Z_3, \ldots, Z_n$ are different from the values of $\{Z_1, Z_2\}$ and then the next fourth rule applies. The fourth rule is for the case when none of the integers in the constraint can take a certain value i. This rule also captures

$c = \texttt{allDiff}([Z_1, \ldots, Z_n])$ where $Z_i = [z_{i,1}, \ldots, z_{i,m}]$ $(1 \leq i \leq n)$	
if	then replace with
$n \leq 1$	true
$\bigwedge_{k>1} \begin{pmatrix} dom(Z_1) \cap \\ dom(Z_k) = \emptyset \end{pmatrix}$	$\texttt{allDiff}([Z_2, \ldots, Z_n])$
$\lvert \bigcup_{i \in \{1,2\}} dom(Z_i) \rvert = 2$	$\texttt{allDiff}([Z_3, \ldots, Z_n])$
$\bigwedge_k Z_k \neq i$	$\texttt{allDiff}([z_{1,1}, \ldots, \cancel{z_{1,i+1}}, \ldots, z_{1,m}] \ldots [z_{n,1}, \ldots, \cancel{z_{n,i+1}}, \ldots, z_{n,m}])$

the case when all of the numbers have leading ones or trailing zeroes. The last two rules are illustrated in Example 10.

(4 & 5) These simplification rules on the right are symmetric when replacing the role of X and Y. The first two apply where (at least) one of X, Y and Z is a constant. Because we have already applied equi-propagation to the constraint, it is a tautology. See Example 9. The last two rules apply to remove leading ones and trailing zeroes. The simplification rules of an `int_array_plus` constraint are straightforward generalizations of the ones for `int_plus`. See Example 11.

$c = \texttt{int\_plus}(X, Y, Z)$ where $X = \langle x_1, \ldots, x_n \rangle$, $Y = \langle y_1, \ldots, y_m \rangle$, and $Z = \langle z_1, \ldots, z_{n+m} \rangle$	
if	then replace with
$X = i$	`true`
$Z = k$	`true`
$X \geq i$, $Z \geq i$	$\texttt{int\_plus}([x_{i+1}, \ldots, x_n], Y, [z_{i+1}, \ldots, z_{n+m}])$
$X \leq i$, $Z \leq i + m$	$\texttt{int\_plus}([x_1, \ldots, x_i], Y, [z_1, \ldots, z_{i+m}])$

To summarise the rule based approach to apply equi-propagation and partial evaluation we present the following sequence of three examples which focus on the simplification of the three constraints given as Figure 6 where the integer variables $\mathtt{I}_1, \ldots, \mathtt{I}_8$ are defined in the range between 1 and 8 and where $K = 14$.

(a) $\texttt{int\_plus}(\mathtt{I}_1, \mathtt{I}_2, \mathtt{K})$
(b) $\texttt{allDiff}([\mathtt{I}_1, \mathtt{I}_2, \mathtt{I}_3, \mathtt{I}_4, \mathtt{I}_5, \mathtt{I}_6, \mathtt{I}_7, \mathtt{I}_8])$
(c) $\texttt{int\_array\_plus}([\mathtt{I}_2, \mathtt{I}_3, \mathtt{I}_4, \mathtt{I}_5], \mathtt{K})$

Fig. 6. Constraint Model for Examples 9–11

Example 9. Consider equi-propagation of constraint (a) from Figure 6 where E_0 specifies that $K = 14$:

$$\begin{array}{|l} \mathtt{E}_0 = \left\{ \begin{array}{l} K_1{=}1, \ldots, K_{14}{=}1 \\ K_{15} = 0, K_{16} = 0 \end{array} \right\} \\ \hline \mathtt{I}_1 = [1, \mathtt{I}_{1,2}, \ldots, \mathtt{I}_{1,8}], \\ \mathtt{I}_2 = [1, \mathtt{I}_{2,2}, \ldots, \mathtt{I}_{2,8}], \\ \mathtt{K} = [\underbrace{1, 1, \ldots, 1}_{14}, 0, 0] \end{array} \xrightarrow[\texttt{int\_plus}]{\mathtt{K}=14} \begin{array}{|l} \mathtt{E}_1 = \mathtt{E}_0 \cup \left\{ \begin{array}{l} I_{1,2}{=}1, \ldots, I_{1,6}{=}1, \\ I_{2,2}{=}1, \ldots, I_{2,6}{=}1, \\ I_{1,7}{=}\neg I_{2,8}, I_{1,8}{=}\neg I_{2,7} \end{array} \right\} \\ \hline \mathtt{I}_1 = [1, 1, 1, 1, 1, 1, \mathtt{I}_{1,7}, \mathtt{I}_{1,8}], \\ \mathtt{I}_2 = [1, 1, 1, 1, 1, 1, \neg\mathtt{I}_{1,8}, \neg\mathtt{I}_{1,7}], \\ \mathtt{K} = [\underbrace{1, 1, \ldots, 1}_{14}, 0, 0] \end{array}$$

Given E_1, the constraint is a tautology and removed by partial evaluation:

$$\left[\begin{array}{l} \texttt{int\_plus}(\\ \quad [1, 1, 1, 1, 1, 1, 1, \mathtt{I}_{1,7}, \mathtt{I}_{1,8}], \\ \quad [1, 1, 1, 1, 1, 1, 1, \neg\mathtt{I}_{1,8}, \neg\mathtt{I}_{1,7}],\ 14) \end{array} \right] \xrightarrow[\texttt{int\_plus}]{P.E} [\ ]$$

□

Example 10. Consider equi-propagation of constraint (b) from Figure 6 given E_1 from Example 9:

$$\begin{array}{|l} \mathtt{E}_1 \\ \hline \mathtt{I}_1 = [1, 1, 1, 1, 1, 1, 1, \mathtt{I}_{1,7}, \mathtt{I}_{1,8}], \\ \mathtt{I}_1 = [1, 1, 1, 1, 1, 1, 1, \neg\mathtt{I}_{1,8}, \neg\mathtt{I}_{1,7}] \end{array} \xrightarrow[\texttt{int\_neq}]{\mathtt{I}_{1,7}=\neg\mathtt{I}_{2,8},\ \mathtt{I}_{2,7}=\neg\mathtt{I}_{1,8}} \begin{array}{|l} \mathtt{E}_2 = \mathtt{E}_1 \cup \{\mathtt{I}_{1,7}{=}\mathtt{I}_{1,8}\} \\ \hline \mathtt{I}_1 = [1, 1, 1, 1, 1, 1, 1, \mathtt{I}_{1,7}, \mathtt{I}_{1,7}], \\ \mathtt{I}_2 = [1, 1, 1, 1, 1, 1, 1, \neg\mathtt{I}_{1,7}, \neg\mathtt{I}_{1,7}] \end{array}$$

Given E_2, the equi-propagation rule for `allDiff` detects that $\{\mathtt{I}_1, \mathtt{I}_2\}$ is a Hall set (where the two variables take values 6 and 8). and adds to E_2 the set of equations, E', that specify that $\mathtt{I}_3, \mathtt{I}_4, \mathtt{I}_5, \mathtt{I}_6, \mathtt{I}_7, \mathtt{I}_8 \neq 6, 8$. The result is $E_3 = E_2 \cup E'$ and the result of this step gives the following bindings (where the impact of E' is underlined):

$$\begin{array}{ll}
\mathtt{I}_1 = [1, 1, 1, 1, 1, 1, 1, \mathtt{I}_{1,7}, \mathtt{I}_{1,7}] & \mathtt{I}_5 = [1, \mathtt{I}_{5,2}, \mathtt{I}_{5,3}, \mathtt{I}_{5,4}, \mathtt{I}_{5,5}, \underline{\mathtt{I}_{5,7}, \mathtt{I}_{5,7}, 0}] \\
\mathtt{I}_2 = [1, 1, 1, 1, 1, 1, \neg\mathtt{I}_{1,7}, \neg\mathtt{I}_{1,7}] & \mathtt{I}_6 = [1, \mathtt{I}_{6,2}, \mathtt{I}_{6,3}, \mathtt{I}_{6,4}, \mathtt{I}_{6,5}, \underline{\mathtt{I}_{6,7}, \mathtt{I}_{6,7}, 0}] \\
\mathtt{I}_3 = [1, \mathtt{I}_{3,2}, \mathtt{I}_{3,3}, \mathtt{I}_{3,4}, \mathtt{I}_{3,5}, \underline{\mathtt{I}_{3,7}, \mathtt{I}_{3,7}, 0}] & \mathtt{I}_7 = [1, \mathtt{I}_{7,2}, \mathtt{I}_{7,3}, \mathtt{I}_{7,4}, \mathtt{I}_{7,5}, \underline{\mathtt{I}_{7,7}, \mathtt{I}_{7,7}, 0}] \\
\mathtt{I}_4 = [1, \mathtt{I}_{4,2}, \mathtt{I}_{4,3}, \mathtt{I}_{4,4}, \mathtt{I}_{4,5}, \underline{\mathtt{I}_{4,7}, \mathtt{I}_{4,7}, 0}] & \mathtt{I}_8 = [1, \mathtt{I}_{8,2}, \mathtt{I}_{8,3}, \mathtt{I}_{8,4}, \mathtt{I}_{8,5}, \underline{\mathtt{I}_{8,7}, \mathtt{I}_{8,7}, 0}]
\end{array}$$

Given E_3, partial evaluation of the constraint first removes the Hall set:

$$[\mathtt{allDiff}([\mathtt{I}_1, \mathtt{I}_2, \mathtt{I}_3, \mathtt{I}_4, \mathtt{I}_5, \mathtt{I}_6, \mathtt{I}_7, \mathtt{I}_8])] \xrightarrow[\mathtt{allDiff}]{P.E} [\mathtt{allDiff}([\mathtt{I}_3, \mathtt{I}_4, \mathtt{I}_5, \mathtt{I}_6, \mathtt{I}_7, \mathtt{I}_8])]$$

and then applies to remove three redundant bits in the underlying representation of each remaining integer (which is not equal to $0, 6, 8$):

$$\left[\begin{array}{l}
\mathtt{allDiff}([\\
\quad [1, \mathtt{I}_{3,2}, \mathtt{I}_{3,3}, \mathtt{I}_{3,4}, \mathtt{I}_{3,5}, \mathtt{I}_{3,7}, \mathtt{I}_{3,7}, 0], \\
\quad [1, \mathtt{I}_{4,2}, \mathtt{I}_{4,3}, \mathtt{I}_{4,4}, \mathtt{I}_{4,5}, \mathtt{I}_{4,7}, \mathtt{I}_{4,7}, 0], \\
\quad [1, \mathtt{I}_{5,2}, \mathtt{I}_{5,3}, \mathtt{I}_{5,4}, \mathtt{I}_{5,5}, \mathtt{I}_{5,7}, \mathtt{I}_{5,7}, 0], \\
\quad [1, \mathtt{I}_{6,2}, \mathtt{I}_{6,3}, \mathtt{I}_{6,4}, \mathtt{I}_{6,5}, \mathtt{I}_{6,7}, \mathtt{I}_{6,7}, 0], \\
\quad [1, \mathtt{I}_{7,2}, \mathtt{I}_{7,3}, \mathtt{I}_{7,4}, \mathtt{I}_{7,5}, \mathtt{I}_{7,7}, \mathtt{I}_{7,7}, 0], \\
\quad [1, \mathtt{I}_{8,2}, \mathtt{I}_{8,3}, \mathtt{I}_{8,4}, \mathtt{I}_{8,5}, \mathtt{I}_{8,7}, \mathtt{I}_{8,7}, 0]])
\end{array}\right] \xrightarrow[\mathtt{allDifferent}]{P.E} \left[\begin{array}{l}
\mathtt{allDiff}([\\
\quad [\mathtt{I}_{3,2}, \mathtt{I}_{3,3}, \mathtt{I}_{3,4}, \mathtt{I}_{3,5}, \mathtt{I}_{3,7}], \\
\quad [\mathtt{I}_{4,2}, \mathtt{I}_{4,3}, \mathtt{I}_{4,4}, \mathtt{I}_{4,5}, \mathtt{I}_{4,7}], \\
\quad [\mathtt{I}_{5,2}, \mathtt{I}_{5,3}, \mathtt{I}_{5,4}, \mathtt{I}_{5,5}, \mathtt{I}_{5,7}], \\
\quad [\mathtt{I}_{6,2}, \mathtt{I}_{6,3}, \mathtt{I}_{6,4}, \mathtt{I}_{6,5}, \mathtt{I}_{6,7}], \\
\quad [\mathtt{I}_{7,2}, \mathtt{I}_{7,3}, \mathtt{I}_{7,4}, \mathtt{I}_{7,5}, \mathtt{I}_{7,7}], \\
\quad [\mathtt{I}_{8,2}, \mathtt{I}_{8,3}, \mathtt{I}_{8,4}, \mathtt{I}_{8,5}, \mathtt{I}_{8,7}]])
\end{array}\right]$$

□

Example 11. Consider equi-propagation of constraint (c) from Figure 6 given E_3 from Example 10. The rules that apply derive from the decomposition of the `int_array_plus` constraint to it `int_plus` parts. These dictate that $\mathtt{I}_3, \mathtt{I}_4, \mathtt{I}_5 \leq 5$:

$$\begin{array}{|l|}
\hline
\underline{\mathtt{E}_3} \\
\mathtt{I}_2 = [1, 1, 1, 1, 1, 1, \neg\mathtt{I}_{1,7}, \neg\mathtt{I}_{1,7}], \\
\mathtt{I}_3 = [1, \mathtt{I}_{3,2}, \mathtt{I}_{3,3}, \mathtt{I}_{3,4}, \mathtt{I}_{3,5}, \mathtt{I}_{3,7}, \mathtt{I}_{3,7}, 0], \\
\mathtt{I}_4 = [1, \mathtt{I}_{4,2}, \mathtt{I}_{4,3}, \mathtt{I}_{4,4}, \mathtt{I}_{4,5}, \mathtt{I}_{4,7}, \mathtt{I}_{4,7}, 0], \\
\mathtt{I}_5 = [1, \mathtt{I}_{5,2}, \mathtt{I}_{5,3}, \mathtt{I}_{5,4}, \mathtt{I}_{5,5}, \mathtt{I}_{5,7}, \mathtt{I}_{5,7}, 0] \\
\hline
\end{array} \xrightarrow[\mathtt{int_plus}]{\mathtt{int_array}} \begin{array}{|l|}
\hline
\underline{\mathtt{E}_4 = \mathtt{E}_3 \cup \{\mathtt{I}_{3,7}{=}0, \mathtt{I}_{4,7}{=}0, \mathtt{I}_{5,7}{=}0\}} \\
\mathtt{I}_2 = [1, 1, 1, 1, 1, 1, \neg\mathtt{I}_{1,7}, \neg\mathtt{I}_{1,7}], \\
\mathtt{I}_3 = [1, \mathtt{I}_{3,2}, \mathtt{I}_{3,3}, \mathtt{I}_{3,4}, \mathtt{I}_{3,5}, 0, 0, 0], \\
\mathtt{I}_4 = [1, \mathtt{I}_{4,2}, \mathtt{I}_{4,3}, \mathtt{I}_{4,4}, \mathtt{I}_{4,5}, 0, 0, 0], \\
\mathtt{I}_5 = [1, \mathtt{I}_{5,2}, \mathtt{I}_{5,3}, \mathtt{I}_{5,4}, \mathtt{I}_{5,5}, 0, 0, 0] \\
\hline
\end{array}$$

Applying partial evaluation simplifies the constraint as follows:

$$\left[\begin{array}{l}
\mathtt{int_array_plus}([\\
\quad [1, 1, 1, 1, 1, 1, 1, \neg\mathtt{I}_{1,7}, \neg\mathtt{I}_{1,7}], \\
\quad [1, \mathtt{I}_{3,2}, \mathtt{I}_{3,3}, \mathtt{I}_{3,4}, \mathtt{I}_{3,5}, 0, 0, 0], \\
\quad [1, \mathtt{I}_{4,2}, \mathtt{I}_{4,3}, \mathtt{I}_{4,4}, \mathtt{I}_{4,5}, 0, 0, 0], \\
\quad [1, \mathtt{I}_{5,2}, \mathtt{I}_{5,3}, \mathtt{I}_{5,4}, \mathtt{I}_{5,5}, 0, 0, 0]],\ 14\ )
\end{array}\right] \xrightarrow[\mathtt{int_array_plus}]{P.E} \left[\begin{array}{l}
\mathtt{int_array_plus}([\\
\quad [\neg\mathtt{I}_{1,7}, \neg\mathtt{I}_{1,7}], \\
\quad [\mathtt{I}_{3,2}, \mathtt{I}_{3,3}, \mathtt{I}_{3,4}, \mathtt{I}_{3,5}], \\
\quad [\mathtt{I}_{4,2}, \mathtt{I}_{4,3}, \mathtt{I}_{4,4}, \mathtt{I}_{4,5}], \\
\quad [\mathtt{I}_{5,2}, \mathtt{I}_{5,3}, \mathtt{I}_{5,4}, \mathtt{I}_{5,5}]],\ 5\ )
\end{array}\right]$$

□

To summarize Examples 9–11 observe that in the initial constraint model 3 constraints about 8 integers are represented in 56 bits. After constraint simplification 2 constraints remain and the 8 integers are represented using 32 bits:

$$\begin{array}{ll}
\mathtt{I}_1 = [1, 1, 1, 1, 1, 1, 1, \mathtt{I}_{1,7}, \mathtt{I}_{1,7}] & \mathtt{I}_5 = [1, \mathtt{I}_{5,2}, \mathtt{I}_{5,3}, \mathtt{I}_{5,4}, \mathtt{I}_{5,5}, 0, 0, 0] \\
\mathtt{I}_2 = [1, 1, 1, 1, 1, 1, 1, \neg\mathtt{I}_{1,7}, \neg\mathtt{I}_{1,7}] & \mathtt{I}_6 = [1, \mathtt{I}_{6,2}, \mathtt{I}_{6,3}, \mathtt{I}_{6,4}, \mathtt{I}_{6,5}, \mathtt{I}_{6,7}, \mathtt{I}_{6,7}, 0] \\
\mathtt{I}_3 = [1, \mathtt{I}_{3,2}, \mathtt{I}_{3,3}, \mathtt{I}_{3,4}, \mathtt{I}_{3,5}, 0, 0, 0] & \mathtt{I}_7 = [1, \mathtt{I}_{7,2}, \mathtt{I}_{7,3}, \mathtt{I}_{7,4}, \mathtt{I}_{7,5}, \mathtt{I}_{7,7}, \mathtt{I}_{7,7}, 0] \\
\mathtt{I}_4 = [1, \mathtt{I}_{4,2}, \mathtt{I}_{4,3}, \mathtt{I}_{4,4}, \mathtt{I}_{4,5}, 0, 0, 0] & \mathtt{I}_8 = [1, \mathtt{I}_{8,2}, \mathtt{I}_{8,3}, \mathtt{I}_{8,4}, \mathtt{I}_{8,5}, \mathtt{I}_{8,7}, \mathtt{I}_{8,7}, 0]
\end{array}$$

6 Compiling Constraints with BEE

BEE (Ben-Gurion Equi-propagation Encoder) is a tool which applies to encode finite domain constraint models to CNF. BEE was first introduced in [45]. During the encoding process, BEE performs optimizations based on equi-propagation and partial evaluation to improve the quality of the target CNF. BEE is implemented in (SWI) Prolog.

The source language for the BEE compiler is also called BEE. It is a constraint modeling language similar to FlatZinc [49], but with a focus on a subset of the language relevant for finite domain constraint problems. Five of the constraint constructs in the BEE language are those introduced in Section 2.1. The full language is presented in Table 2.

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. in Table 2, `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 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 25 constraint templates.

A main design choice of BEE is that all integer variables are represented in the order-encoding. So, BEE is suitable for problems in which the integer variables take small or medium sized values. 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 and equi-propagation in BEE follow the general descriptions from Sections 2.4 and 3.1. Bit-blasting is implemented through Prolog unification. Each declaration of the form $\texttt{new\_int}(\texttt{I}, \texttt{c}_1, \texttt{c}_2)$ triggers a unification $\texttt{I} = [\texttt{1}, \ldots, \texttt{1}, \texttt{X}_{\texttt{c}_1+\texttt{1}}, \ldots, \texttt{X}_{\texttt{c}_2}]$ (to ease presentation we assume that integer variables are represented in a positive interval starting from 0 but there is no such limitation as BEE also supports negatives integers). BEE applies ad-hoc equi-propagators as described in Section 4. 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. This unification applies to all occurrences of X and in this sense "propagates" 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

Declaring Variables	
(1) $\texttt{new\_bool}(\texttt{X})$	declare Boolean $\texttt{X}$
(2) $\texttt{new\_int}(\texttt{I}, \texttt{c}_1, \texttt{c}_2)$	declare integer $\texttt{I}$, $\texttt{c}_1 \leq \texttt{I} \leq \texttt{c}_2$
(3) $\texttt{bool2int}(\texttt{X}, \texttt{I})$	$(\texttt{X} \Leftrightarrow \texttt{I} = 1) \wedge (\neg\texttt{X} \Leftrightarrow \texttt{I} = 0)$
Boolean (reified) Statements	$\texttt{op} \in \{\texttt{or}, \texttt{and}, \texttt{xor}, \texttt{iff}\}$
(4) $\texttt{bool\_eq}(\texttt{X}_1, \texttt{X}_2)$ or $\texttt{bool\_eq}(\texttt{X}_1, -\texttt{X}_2)$	$\texttt{X}_1 = \texttt{X}_2$ or $\texttt{X}_1 = -\texttt{X}_2$
(5) $\texttt{bool\_array\_op}([\texttt{X}_1, \ldots, \texttt{X}_n])$	$\texttt{X}_1\ \texttt{op}\ \texttt{X}_2 \cdots \texttt{op}\ \texttt{X}_n$
(6) $\texttt{bool\_array\_op\_reif}([\texttt{X}_1, \ldots, \texttt{X}_n], \texttt{X})$	$\texttt{X}_1\ \texttt{op}\ \texttt{X}_2 \cdots \texttt{op}\ \texttt{X}_n \Leftrightarrow \texttt{X}$
(7) $\texttt{bool\_op\_reif}(\texttt{X}_1, \texttt{X}_2, \texttt{X})$	$\texttt{X}_1\ \texttt{op}\ \texttt{X}_2 \Leftrightarrow \texttt{X}$
Integer relations (reified) and arithmetic	$\texttt{rel} \in \{\texttt{leq}, \texttt{geq}, \texttt{eq}, \texttt{lt}, \texttt{gt}, \texttt{neq}\}$, $\texttt{op} \in \{\texttt{plus}, \texttt{times}, \texttt{div}, \texttt{mod}, \texttt{max}, \texttt{min}\}$, $\texttt{op}' \in \{\texttt{plus}, \texttt{max}, \texttt{min}\}$
(8) $\texttt{int\_rel}(\texttt{I}_1, \texttt{I}_2)$	$\texttt{I}_1\ \texttt{rel}\ \texttt{I}_2$
(9) $\texttt{int\_rel\_reif}(\texttt{I}_1, \texttt{I}_2, \texttt{X})$	$\texttt{I}_1\ \texttt{rel}\ \texttt{I}_2 \Leftrightarrow \texttt{X}$
(10) $\texttt{int\_array\_allDiff}([\texttt{I}_1, \ldots, \texttt{I}_n])$	$\bigwedge_{\texttt{i}<\texttt{j}} \texttt{I}_\texttt{i} \neq \texttt{I}_\texttt{j}$
(11) $\texttt{int\_abs}(\texttt{I}_1, \texttt{I})$	$\lvert\texttt{I}_1\rvert = \texttt{I}$
(12) $\texttt{int\_op}(\texttt{I}_1, \texttt{I}_2, \texttt{I})$	$\texttt{I}_1\ \texttt{op}\ \texttt{I}_2 = \texttt{I}$
(13) $\texttt{int\_array\_op}'([\texttt{I}_1, \ldots, \texttt{I}_n], \texttt{I})$	$\texttt{I}_1\ \texttt{op}' \cdots \texttt{op}'\ \texttt{I}_n = \texttt{I}$
Linear Constraints	$\texttt{rel} \in \{\texttt{leq}, \texttt{geq}, \texttt{eq}, \texttt{lt}, \texttt{gt}\}$
(14) $\texttt{bool\_array\_sum\_rel}([\texttt{X}_1, \ldots, \texttt{X}_n], \texttt{I})$	$(\Sigma\ \texttt{X}_\texttt{i})\ \texttt{rel}\ \texttt{I}$
(15) $\texttt{bool\_array\_pb\_rel}([\texttt{c}_1, \ldots, \texttt{c}_n], [\texttt{X}_1, \ldots, \texttt{X}_n], \texttt{I})$	$(\Sigma\ \texttt{c}_\texttt{i} * \texttt{X}_\texttt{i})\ \texttt{rel}\ \texttt{I}$
(16) $\texttt{bool\_array\_sum\_modK}([\texttt{X}_1, \ldots, \texttt{X}_n], \texttt{c}, \texttt{I})$	$((\Sigma\ \texttt{X}_\texttt{i})\ \texttt{mod}\ \texttt{c}) = \texttt{I}$
(17) $\texttt{int\_array\_sum\_rel}([\texttt{I}_1, \ldots, \texttt{I}_n], \texttt{I})$	$(\Sigma\ \texttt{I}_\texttt{i})\ \texttt{rel}\ \texttt{I}$
(18) $\texttt{int\_array\_lin\_rel}([\texttt{c}_1, \ldots, \texttt{c}_n], [\texttt{I}_1, \ldots, \texttt{I}_n], \texttt{I})$	$(\Sigma\ \texttt{c}_\texttt{i} * \texttt{I}_\texttt{i})\ \texttt{rel}\ \texttt{I}$
(19) $\texttt{int\_array\_sum\_modK}([\texttt{I}_1, \ldots, \texttt{I}_n], \texttt{c}, \texttt{I})$	$((\Sigma\ \texttt{I}_\texttt{i})\ \texttt{mod}\ \texttt{c}) = \texttt{I}$
Lexical Order	
(20) $\texttt{bool\_arrays\_lex}(\texttt{Xs}_1, \texttt{Xs}_2)$	$\texttt{Xs}_1$ precedes (leq) $\texttt{Xs}_2$
(21) $\texttt{bool\_arrays\_lexLt}(\texttt{Xs}_1, \texttt{Xs}_2)$	$\texttt{Xs}_1$ precedes (lt) $\texttt{Xs}_2$
(22) $\texttt{bool\_arrays\_lex\_reif}(\texttt{Xs}_1, \texttt{Xs}_2, \texttt{X})$	$\texttt{X} \Leftrightarrow \texttt{Xs}_1$ precedes (leq)
(23) $\texttt{bool\_arrays\_lexLt\_reif}(\texttt{Xs}_1, \texttt{Xs}_2, \texttt{X})$	$\texttt{X} \Leftrightarrow \texttt{Xs}_1$ precedes (lt) $\texttt{Xs}_2$
(24) $\texttt{int\_arrays\_lex}(\texttt{Is}_1, \texttt{Is}_2)$	$\texttt{Is}_1$ precedes (leq) $\texttt{Is}_2$
(25) $\texttt{int\_arrays\_lexLt}(\texttt{Is}_1, \texttt{Is}_2)$	$\texttt{Is}_1$ precedes (lt) $\texttt{Is}_2$

Table 2. Syntax of BEE Constraints.

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 a constraint of the form `int_eq(A,Sum)` which equates the bits of `A` and `Sum`, or if $\texttt{As} = [\texttt{A}_1, \texttt{A}_2]$ then it is replaced by $\texttt{int\_plus}(\texttt{A}_1, \texttt{A}_2, \texttt{Sum})$. 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 $\texttt{int\_plus}(\texttt{A}_1, \texttt{A}_2, \texttt{A})$ constraint. One approach, supported by BEE, decomposes the constraint as an odd-even merger (from the context of odd-even sorting networks) [6]. Here, the sorted sequences of bits $\texttt{A}_1$ and $\texttt{A}_2$ 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 [3]. 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 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. The BEE encodings are similar to those applied in Sugar [58].

6.1 The All-different constraint in BEE

The all-different constraint specifies that a set of integer variables take all different values from their specified domains. This constraint has received much attention in the literature (see for example the survey [62]). BEE provides special treatment for this constraint.

In many applications, all-different constraints are applied to model the special case when the constraint is about "permutation". Namely, when $[\texttt{I}_1, \ldots, \texttt{I}_\texttt{n}]$ are all different but may take precisely n different values. BEE identifies this special case and applies two additional ad-hoc equi-propagation rules for this case. The table on the right illustrates these rules. We annotate the constraint with a "*" to emphasize that it has been detected that it is about permutation. The first rule is about the case when only one integer (assume $\texttt{Z}_1$) can take the value i. The second rule is about the case where all variables except two, assume $\texttt{Z}_1$, $\texttt{Z}_2$, can not take two values, assume $\texttt{i}, \texttt{j}$. Now, because the constraint is about permutation, we can determine that $\texttt{Z}_1$ and $\texttt{Z}_2$ must take the two values `i` and `j`. To illustrate the second rule consider the following example.

$c = \texttt{allDiff}^*([Z_1, Z_2, Z_3, \ldots, Z_n])$	
if in E	then add in $\mu_c(\texttt{E})$
$i \in dom(Z_1)$ $i \notin dom(Z_k)$ $(k > 1)$	$Z_1 = i$
$\{i, j\} \cap dom(Z_k) = \emptyset$ $(k > 2)$	$dom(Z_1) \subseteq \{i, j\}$ $dom(Z_2) \subseteq \{i, j\}$ $Z_1 \neq Z_2$ $Z_k \neq i, Z_k \neq j$ $(k > 2)$

Example 12. Consider a constraint $\texttt{allDiff}(\texttt{I}_1, \ldots, \texttt{I}_5)$ on 5 integer variables taking values in the interval $[0, 4]$ (exactly 5 values) where E_0 specifies that $\texttt{I}_3$, $\texttt{I}_4$ and $\texttt{I}_5$ cannot take the values 0 and 1. Therfore we introduce equations which restrict $\texttt{I}_1$ and $\texttt{I}_2$ to take the values 0 and 1.

and the corresponding ad-hoc rule for permutation applies:

$$\begin{array}{|l|}\hline \texttt{E}_0 = \left\{\begin{array}{l} x_{3,1}{=}1, x_{4,1}{=}1, \\ x_{5,1}{=}1, x_{3,2}{=}1, \\ x_{4,2}{=}1, x_{5,2}{=}1 \end{array}\right\} \\ \hline \texttt{I}_1 = [\texttt{x}_{1,1}, \ldots, \texttt{x}_{1,4}], \\ \texttt{I}_2 = [\texttt{x}_{2,1}, \ldots, \texttt{x}_{2,4}], \\ \texttt{I}_3 = [1, 1, \texttt{x}_{3,3}, \texttt{x}_{3,4}], \\ \texttt{I}_4 = [1, 1, \texttt{x}_{4,3}, \texttt{x}_{4,4}], \\ \texttt{I}_5 = [1, 1, \texttt{x}_{5,3}, \texttt{x}_{5,4}] \\ \hline \end{array} \xrightarrow[\texttt{allDiff}^*]{\begin{array}{l}\texttt{dom}(\texttt{I}_\texttt{k}) \cap \{0,1\}{=}\emptyset \\ \texttt{k} > 2\end{array}} \begin{array}{|l|}\hline \texttt{E}_1 = \texttt{E}_0 \cup \left\{\begin{array}{l} x_{1,2}{=}0, \ldots, x_{1,4}{=}0, \\ x_{2,2}{=}0, \ldots, x_{2,4}{=}0 \\ x_{1,1}{=}\neg x_{2,1} \end{array}\right\} \\ \hline \texttt{I}_1 = [\texttt{x}_{1,1}, 0, \ldots, 0], \\ \texttt{I}_2 = [\neg\texttt{x}_{1,1}, 0, \ldots, 0], \\ \texttt{I}_3 = [1, 1, \texttt{x}_{3,3}, \texttt{x}_{3,4}], \\ \texttt{I}_4 = [1, 1, \texttt{x}_{4,3}, \texttt{x}_{4,4}], \\ \texttt{I}_5 = [1, 1, \texttt{x}_{5,3}, \texttt{x}_{5,4}] \\ \hline \end{array}$$

□

To facilitate the implementation of ad-hoc equi-propagation of all-different constraints, BEE adopts a dual representation for integer variables occurring in these constraints. When declaring an integer variable I, the bit-blast in the order encoding applies the corresponding unification $\texttt{I} = [\texttt{x}_1, \ldots, \texttt{x}_\texttt{n}]$. When encountering I in an allDiff constraint, an additional bit-blast introduces $\texttt{I}' = [\texttt{d}_0, \ldots, \texttt{d}_\texttt{n}]$ in the, so-called, direct encoding, and a channeling formula $\texttt{channel}(\texttt{I}, \texttt{I}')$ is introduced.

The direct encoding is a unary representation $\texttt{I}' = [\texttt{d}_0, \ldots, \texttt{d}_\texttt{n}]$ where each bit d_i is true if and only if $\texttt{I}' = \texttt{i}$. So, exactly one of the bits takes the value true. For example, the value 3 in the interval $[0, 5]$ is represented in 6 bits as $[0, 0, 0, 1, 0, 0]$. In the dual representation the following channeling formula captures the relation between the two representations of an integer variable $\texttt{I} = [\texttt{x}_1, \ldots, \texttt{x}_\texttt{n}]$ and $\texttt{I}' = [\texttt{d}_0, \ldots, \texttt{d}_\texttt{n}]$.

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

During constraint simplification, the $\texttt{allDiff}([\texttt{I}_1, \ldots, \texttt{I}_\texttt{n}])$ constraint is viewed through its direct encoding as a bit matrix where each row consists of the bits $[\texttt{d}_{\texttt{i}0}, \ldots, \texttt{d}_{\texttt{i}\texttt{m}}]$ 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. This representation has one main advantage: in the direct encoding we can decompose $\texttt{allDiff}([\texttt{I}_1, \ldots, \texttt{I}_\texttt{n}])$, to conjunction of n $\left\{ \texttt{bool\_array\_sum\_leq}() \,\middle|\, 1 \leq j \leq n \right\}$ constraints, which is arc-consistent. As soon as $d_{i,j} = 1$ ($I_j = i$) we have $d_{i,j'} = 0$ ($I_{j'} \neq i$) for all $j' \neq j$. In contrast in the order encoding the decomposition to $O(n^2)$ constraints $\left\{ \texttt{int\_neq}(\texttt{I}_\texttt{i}, \texttt{I}_\texttt{j}) \,\middle|\, i < j \right\}$ is not arc-consistent. We illustrate the advantage of the dual encoding for the allDiff constraint in Section 8.1.

7 Using BEE

BEE is implemented in (SWI) Prolog and can be applied in conjunction with the CryptoMiniSAT solver [56] through a Prolog interface [13]. BEE can be downloaded from [42] where one can find also the examples from this thesis and others.

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

Fig. 7. A generic application of BEE.

A typical BEE application has the form depicted as Figure 7 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 `bCompile/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 `bCompile/2` and `sat/1` provide the interface to BEE and the underlying SAT solver.

7.1 Example BEE Application: magic graph labeling

We illustrate the application of BEE using Prolog as a modeling language to solve a graph labeling problem. 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 [28] 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. In the context of Figure 7, 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 8 illustrates an example problem instance together with its solution.

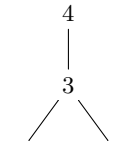
An Instance	The Graph	A Solution
$\texttt{Instance} = \texttt{vmtl}(\texttt{G},\texttt{K}),$ $\texttt{G} = (\texttt{V},\texttt{E}),$ $\texttt{V} = [1,2,3,4],$ $\texttt{E} = [(1,2),(1,3),$ $\quad (2,3),(3,4)],$ $\texttt{K} = 14$		$\begin{bmatrix} \texttt{V}_1 = 4, & \texttt{E}_{(1,2)} = 7, \\ \texttt{V}_2 = 5, & \texttt{E}_{(1,3)} = 3, \\ \texttt{V}_3 = 1, & \texttt{E}_{(2,3)} = 2, \\ \texttt{V}_4 = 6, & \texttt{E}_{(3,4)} = 8 \end{bmatrix}$

Fig. 8. A VMTL instance with a solution.

Figure 9 illustrates a Prolog program that implements the `encode/3` predicate for the VMTL problem. The call to predicate `declareInts/4` introduces the constraints which declare the integer variables for each vertex and edge in the graph, and generates the map. The call to predicate `sumToK/5` introduces the constraints that require the sum of the labels for each vertex with its incident edges to equals K. The auxilary predicate `getVars/3` receives a list of identifiers (vertices and edges) and extracts the corresponding list of integer variables from the map.

Given the VMTL instance from Figure 8, the call to predicate `encode/3` from Figure 9 generates the map and the constraints detailed in Figure 10.

Solving the constraints from Figure 10 binds the Map as follows, indicating a solution (in unary order encoding):

$$\texttt{M} = \begin{bmatrix} (1,\ [1,1,1,1,0,0,0,0]),\ ((1,2),\ [1,1,1,1,1,1,1,0]), \\ (2,\ [1,1,1,1,1,0,0,0]),\ ((1,3),\ [1,1,1,0,0,0,0,0]), \\ (3,\ [1,0,0,0,0,0,0,0]),\ ((2,3),\ [1,1,0,0,0,0,0,0]), \\ (4,\ [1,1,1,1,1,1,0,0]),\ ((3,4),\ [1,1,1,1,1,1,1,1]) \end{bmatrix}$$

Using BEE to compile the constraints from Figure 10 generates a CNF which contains 301 clauses and 48 Boolean variables. Encoding the same set of constraints without applying simplification rules generates a larger CNF which contains 642 clauses and 97 Boolean variables.

In Section 8.3 we report that using BEE enables us to solve interesting instances of the VMTL problem not previously solvable by other techniques.

7.2 BumbleBEE

The BEE distribution includes also a command line solver, which we call BumbleBEE. BumbleBEE enables one to specify a BEE model in an input file where each line contains a single constraint from the model and the last line specifies the type of goal. BumbleBEE reads the input file, compiles the constraint model to CNF, solves the CNF using the embedded CryptoMiniSAT solver [56] and outputs a set of bindings to the declared variables in the model (or a message

```
encode(vmtl((Vs,Es),K),Map,Constraints):-
    append(Vs,Es,VEs), length(VEs,N),
    declareInts(VEs,N,Map,Constraints-Cs2),
    sumToK(Vs,Es,Map,K,Cs2-Cs3),
    getVars(VEs,Map,Vars),
    Cs3=[int_array_allDiff(Vars)].

declareInts([],_,_,Cs-Cs).
declareInts([ID|IDs],N,[(ID,X)|Map],[new_int(X,1,N)|CsH]-CsT):-
    declareInts(IDs,N,Map,CsH-CsT).

sumToK([],_,_,_,Cs-Cs).
sumToK([VID|Vs],Es,Map,K,[int_array_plus(Vars,K)|CsH]-CsT):-
    findall((X,Y),(member((X,Y),Es),(X=VID ; Y=VID)),EsIDs),
    getVars([VID|EsIDs],Map,Vars),
    sumToK(Vs,Es,Map,K,CsH-CsT).

getVars([],_,[]).
getVars([ID|IDs],Map,[Var|Vars]):-
    member((ID,Var),Map),
    getVars(IDs,Map,Vars).
```

Fig. 9. encode/3 predicate for the VMTL application of BEE

The Map	The Constraints
$((1,2),E_1),(1,V_1),$ $((1,3),E_2),(2,V_2),$ $((2,3),E_3),(3,V_3),$ $((3,4),E_4),(4,V_4)$	$\texttt{new\_int}(V_1,1,8),\texttt{new\_int}(E_1,1,8),\texttt{int\_array\_plus}([V_1,E_1,E_2],K),$ $\texttt{new\_int}(V_2,1,8),\texttt{new\_int}(E_2,1,8),\texttt{int\_array\_plus}([V_2,E_1,E_3],K),$ $\texttt{new\_int}(V_3,1,8),\texttt{new\_int}(E_3,1,8),\texttt{int\_array\_plus}([V_3,E_2,E_3,E_4],K),$ $\texttt{new\_int}(V_4,1,8),\texttt{new\_int}(E_4,1,8),\texttt{int\_array\_plus}([V_4,E_4],K),$ $\texttt{new\_int}(K,14,14),\texttt{allDiff}([V_1,V_2,V_3,V_4,E_1,E_2,E_3,E_4])$

Fig. 10. A VMTL instance with the constraints and map generated by `encode/3`.

indicating that the constraints are not satisfiable). Figure 11 contains on the left the BumbleBEE input file for the VMTL instance from Figure 8 and on the right the BumbleBEE output, which is a solution for the constraint model. Further details and more examples can be found in the BEE distribution [45].

8 Experiments

We report on our experience in applying BEE. To appreciate the ease in its use the reader is encouraged to view the example encodings available with the tool [45]. 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.

Content of BumbleBEE input file	BumbleBEE output
new_int(V1, 1, 8)	
new_int(V2, 1, 8)	
new_int(V3, 1, 8)	V1 = 4
new_int(V4, 1, 8)	V2 = 5
new_int(E1, 1, 8)	V3 = 1
new_int(E2, 1, 8)	V4 = 6
new_int(E3, 1, 8)	E1 = 7
new_int(E4, 1, 8)	E2 = 3
int_array_plus([V1, E1, E2], 14)	E3 = 2
int_array_plus([V2, E1, E3], 14)	E4 = 8
int_array_plus([V3, E2, E3, E4], 14)	----------
int_array_plus([V4, E4], 14)	==========
int_array_allDiff([V1, V2, V3, V4, E1, E2, E3, E4])	
solve satisfy	

Fig. 11. Solving VMTL instance using BumbleBEE.

8.1 Quasigroup Completion Problems

A Quasigroup Completion Problem (proposed by Gomes *et al.* in [32] as a constraint satisfaction benchmark) (QCP) is given as an $n \times n$ board of integer variables (in the range $[1, n]$) in which some are assigned integer values. The task is to assign values to all variables, so that no column or row contains the same value twice. The constraint model is a conjunction of `allDiff` constraints. In [1], the authors argue the advantage of the direct encoding for QCP.

We consider 15 instances from the 2008 CSP competition[3]. Table 3 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 than Sugar but BEE still generates CNF 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.

8.2 Word Design for DNA

This is Problem 033 of CSPLib which 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)**

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

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 3. QCP results for 25×25 instances with 264 holes

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 [40], Mancini *et al.* 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 [61] model with lexicographic order to break symmetry. In [27], 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. The authors report a solution composed from two pairs of (t-part and m-part) sets[4] $\langle T_1, M_1\rangle$ 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.[5]

Building on the approach described in [27], 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 $\texttt{dna}(\texttt{n}_1, \texttt{n}_2)$ 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 $S = T \times M$.

[4] Their notions of t-part and m-part are slightly different than ours.

[5] See `http://www.cs.st-andrews.ac.uk/~ianm/CSPLib/`.

Using BEE, we find, in a fraction of a second, sets of t-parts of size 14 and m-parts 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.[6] Using BEE, we also prove that there does not exist a set of 15 t-parts (0.15 seconds), nor a set of 9 m-parts (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.

8.3 Vertex Magic Total Labeling

In [39] 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 8), 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 4 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).

K_8	k	BEE	Sugar
	143	1.26	2.87
	142	10.14	1.62
	141	7.64	2.94
	140	14.68	6.46
	139	25.60	6.67
	138	12.99	2.80
	137	22.91	298.58
	136	14.46	251.82
	135	298.54	182.90
	134	331.80	∞
Average			
clauses $\times$**1000**		248	402
vars		5688	9370

K_{10}	k	BEE	Sugar
	277	5.31	9.25
	276	7.11	9.91
	275	13.57	19.63
	274	4.93	9.24
	273	45.94	9.03
	272	22.74	86.45
	271	7.35	9.49
	270	6.03	55.94
	269	5.20	11.05
	268	94.44	424.89
clauses $\times$**1000**			
vars			

k	BEE	Sugar
267	88.51	175.70
266	229.80	247.56
265	1335.31	259.45
264	486.09	513.61
263	236.68	648.43
262	1843.70	6429.25
261	2771.60	7872.76
260	4873.99	∞
259	∞	∞
258	∞	∞
Average		
1229		1966
15529		25688

Table 4. VMTL results for K_8 and K_{10} (times are in seconds)

[6] See `http://www.hakank.org/comet/word_design_dna1.co`.

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 just over 1 million clauses and 15 thousand 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 [36] cannot handle these.[7] The comparison with Sugar indicates the impact of the compiler.

8.4 Balanced Incomplete Block Designs

This is Problem 028 of CSPlib (BIBD) where an instance is defined by a 5-tuple of positive integers $\langle v, b, r, k, \lambda \rangle$ and requires to partition v distinct objects into b blocks such that each block contains k different objects, exactly r objects occur in each block, and every two distinct objects occur in exactly λ blocks.

The naive model for a BIBD instance $\langle v, b, r, k, \lambda \rangle$ introduces the following constraints on a v by b Boolean incidence matrix: (1) exactly r ones in each row, (2) exactly k ones in each column, and (3) exactly λ ones in each scalar product of two (different) rows.

This model does not contain a sufficient degree of information to start the equi-propagation process. In order to take advantage of the BEE simplifications we added symmetry breaking as described in [26]: Each row is viewed as sequance of four parts $A \ldots D$ in sizes λ, $(r - \lambda)$, $(r - \lambda)$, and $(b - 2r + \lambda)$.

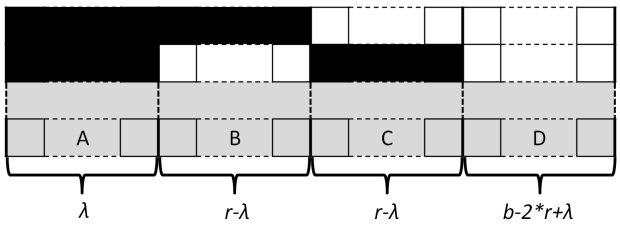


Fig. 12. BIBD symmetry breaking.

[7] Personal communication (Gerold Jäger), March 2012.

The first row is fixed by assign parts A and B with ones (mark in black) and parts C and D with zeros (mark in white).

The second row is fixed by assign parts A and C with ones (mark in black) and parts B and D with zeros (mark in white).

For the third and all subsequent rows the sum constraints are decomposed into summing each part ($A \dots D$) and sum the results as followed: $A + B = \lambda$, $A + C = \lambda$, $C + D = r - \lambda$, and $B + D = r - \lambda$. This ensure that the row contains exactly r ones and the scalar product with the first (and second) row is λ.

We denote this constraint model `SymB` (for symmetry breaking).

instance	BEE (SymB)			Sugar (SymB)			SatELite (SymB)		
$\langle v, b, r, k, \lambda \rangle$	comp (sec.)	cnf size (clauses)	SAT (sec.)	comp (sec.)	cnf size (sec.)	SAT (sec.)	prepro (sec.)	cnf size (clauses)	SAT (sec.)
$\langle 7, 420, 180, 3, 60 \rangle$	1.65	698579	1.73	12.01	2488136	13.24	1.67	802576	2.18
$\langle 7, 560, 240, 3, 80 \rangle$	3.73	1211941	13.60	11.74	2753113	36.43	2.73	1397188	5.18
$\langle 12, 132, 33, 3, 6 \rangle$	0.95	180238	0.73	83.37	1332241	7.09	1.18	184764	0.57
$\langle 15, 45, 24, 8, 12 \rangle$	0.51	116016	8.46	4.24	466086	∞	0.64	134146	∞
$\langle 15, 70, 14, 3, 2 \rangle$	0.56	81563	0.39	23.58	540089	1.87	1.02	79542	0.20
$\langle 16, 80, 15, 3, 2 \rangle$	0.81	109442	0.56	64.81	623773	2.26	1.14	105242	0.35
$\langle 19, 19, 9, 9, 4 \rangle$	0.23	39931	0.09	2.27	125976	0.49	0.4	44714	0.09
$\langle 19, 57, 9, 3, 1 \rangle$	0.34	113053	0.17	∞	-	-	10.45	111869	0.14
$\langle 21, 21, 5, 5, 1 \rangle$	0.02	0	0.00	31.91	3716	0.01	0.01	0	0.00
$\langle 25, 25, 9, 9, 3 \rangle$	0.64	92059	1.33	42.65	569007	8.52	1.01	97623	8.93
$\langle 25, 30, 6, 5, 1 \rangle$	0.10	24594	0.06	16.02	93388	0.42	1.2	23828	0.05
Total	36.66			> 722.93			> 219.14		

Table 5. BIBD results (180 sec. timeout)

Table 5 shows results comparing BEE (compilation time, cnf size, and sat solving time) with Sugar using `SymB` model. We also comparing BEE with SatELite [20], a CNF minimizer, where SatELite inputs were the CNF encoding for `SymB` model generated by BEE without applying any simplifications.

This experiment indicates that BEE generates a significantly smaller CNF than Sugar which affects the SAT solving time. Moreover, Sugar compilation time is extremely long. When comparing BEE with SatELite we can see that both output CNF which is similar in size but as SatELite applied on the entire CNF, for some instances the compilation time is significantly longer than the solving time.

Table 6 shows results comparing BEE using `SymB` model with the Minion constraint solver [31]. We consider three different models for Minion: `[M'06]` indicates results using the BIBD model described in [31], `SymB` uses the same model we use for the SAT approach, `SymB`$^+$, is an enhanced symmetry breaking model with all of the tricks applied also in the `[M'06]` model. For the columns with no timeouts we show total times (for the compiler this includes compile time and sat solving). Note that by using a clever modeling of the problem we have improved also the previous runtimes for Minion.

This experiment indicates that BEE is significantly faster than Minion on its BIBD models (`[M'06]`). Only when tailoring our `SymB` model, Minion becomes competitive with ours.

instance	BEE (SymB)		Minion		
$\langle v, b, r, k, \lambda \rangle$	comp (sec.)	SAT (sec.)	[M'06] (sec.)	SymB (sec.)	SymB$^+$ (sec.)
$\langle 7, 420, 180, 3, 60 \rangle$	1.65	1.73	0.54	1.36	0.42
$\langle 7, 560, 240, 3, 80 \rangle$	3.73	13.60	0.66	1.77	0.52
$\langle 12, 132, 33, 3, 6 \rangle$	0.95	0.73	5.51	∞	1.76
$\langle 15, 45, 24, 8, 12 \rangle$	0.51	8.46	∞	∞	75.87
$\langle 15, 70, 14, 3, 2 \rangle$	0.56	0.39	12.22	1.42	0.31
$\langle 16, 80, 15, 3, 2 \rangle$	0.81	0.56	107.43	13.40	0.35
$\langle 19, 19, 9, 9, 4 \rangle$	0.23	0.09	53.23	38.30	0.31
$\langle 19, 57, 9, 3, 1 \rangle$	0.34	0.17	∞	1.71	0.35
$\langle 21, 21, 5, 5, 1 \rangle$	0.02	0.00	1.26	0.67	0.15
$\langle 25, 25, 9, 9, 3 \rangle$	0.64	1.33	∞	∞	0.92
$\langle 25, 30, 6, 5, 1 \rangle$	0.10	0.06	∞	1.37	0.31
Total	36.66				81.24

Table 6. BIBD results (180 sec. timeout)

9 Conclusion

There is a considerable body of work on CNF simplification techniques with a clear trade-off between amount of reduction achieved and invested time. Most of these approaches determine binary clauses implied by the CNF, which is certainly enough to determine Boolean equalities. The problem is that determining all binary clauses implied by the CNF is prohibitive when the SAT model may involve many (hundreds of) thousands of variables. Typically only some of the implied binary clauses are determined, such as those visible by unit propagation. The trade-off is regulated by the choice of the techniques applied to infer binary clauses, considering the power and cost. See for example [20] and the references therein. There are also approaches [38] that detect and use Boolean equalities during runtime, which are complementary to our approach.

In our approach, the beast is tamed by introducing a notion of locality. We do not consider the full CNF. Instead, by maintaining the original representation, a conjunction of constraints, each viewed as a Boolean formula, we can apply powerful reasoning techniques to separate parts of the model and maintain efficient preprocessing.

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. Applying optimizations based on ad-hoc equipropagation and partial evaluation rules on a high level view of the problem allows us to simplify the problem more aggressively than is possible with a CNF representation. The resulting CNF models can be significantly smaller than those resulting from straight translation.

Although 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.

BEE uses ad-hoc equi-propagation and partial evaluation rules which keeps compilation times typically small (measured in seconds) even for instances which result in several millions of CNF clauses. And the reduction in SAT solving time can be larger in orders of magnitude. Hence we believe that Boolean equi-propagation, combined with CNF simplification tools, such as SatELite [20] or Coprocessor [41], makes an important contribution to the encoding of CSPs to SAT.

Part II

Model-Based Diagnosis

1 Introduction

Automated diagnosis is concerned with reasoning about the health of systems, including the identification of abnormal behavior, isolation of faulty components and prediction of system behavior under normal and abnormal conditions. As systems become large-scale and more complex, their automated diagnosis becomes more challenging. Model Based Diagnosis (MBD) is an artificial intelligence based approach that aims to cope with the diagnosis problem [50, 19]. In MBD, a model of the system is first built. A diagnoser then observes the system to predict its behavior by the model. Discrepancies between the observation and the prediction are used as the input for a diagnosis algorithm which produces a set of possible faults that can explain the observation.

MBD is known to be a hard problem where algorithms have exponential runtime (in the size of the system). Moreover, the number of potential diagnoses that may explain an observation can be huge. Therefore, MBD algorithms typically focus on minimal diagnoses – that do not contain other diagnoses, and on minimal cardinality diagnoses – that are smallest in size. Computing the first minimal diagnosis is in P, but computing the next one is NP-hard [10]. Computing the minimal cardinality is NP-hard, even for the first diagnosis [51].

In case of large-scale systems, solving an MBD problem is often impractical, especially for high-cardinality faults. For instance, in a system of 1000 components, to find a minimal cardinality diagnosis of size 5, a diagnosis engine must verify the absence of a diagnosis consisting of 4 components (there are more than 10^{11} such combinations). To overcome this problem we consider a novel encoding to SAT.

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.

SAT-based solutions for MBD have already been proposed. Smith *et al.* [54] encode a circuit, representing each component through its clauses and add constraints for cardinality. This is the basis for all the other SAT-based encodings, including the one we contribute in this thesis. Bauer [7] introduces a tailored SAT solver specifically designed to return many diagnoses. Stein *et. al.* [57] address diagnosis of qualitative models of physical systems with multiple fault modes. More recently, Feldman *et al.* [23] propose an encoding to MAX-SAT and demonstrate that off-the-shelf solvers require more calls to a SAT solver than the stochastic diagnosis algorithm SAFARI [24].

These previous applications of SAT for MBD appear to indicate that SAT and MAX-SAT solvers are doomed to perform poorly on the standard benchmarks [23]. This thesis proves the contrary. In our approach, sophisticated MBD

preprocessing techniques are first applied to facilitate the construction of a carefully designed constraint model that captures more precisely the MBD problem in terms of its substructures. This constraint model is then compiled to a corresponding CNF in a process which first simplifies constraints propagating information which then enables to perform an encoding to CNF which significantly improves the subsequent runtime of the underlying SAT solver.

We focus on minimal cardinality diagnosis evaluating our approach using two standard benchmarks: ISCAS-85 [9] and 74XXX [33]. We run three known sets of observations with minimal cardinalities between 1–30, and for the first time succeed to compute a minimal cardinality diagnosis for all observations in the benchmark. We compare our approach to a wide collection of the best state-of-the-art algorithms for MBD, including: HA* [25], CDA* [63], SAFARI [24], HDIAG [53] and DCAS [52]. Results are unequivocal. Our approach outperforms the others, often by orders of magnitude, in terms of runtime. This result is even more significant, as SAFARI is a stochastic algorithm, known as fast, which does not even aim to guarantee minimal cardinality. Our approach, on the other hand, guarantees a minimal cardinality diagnosis and runs faster than SAFARI.

This part of the thesis is an extended [47]. In the next section we discuss additional related work, then in Section 3 we formalize the MBD problem and in Section 4 we present the standard approach to model MBD with SAT. Section 5 is the main part in which we describe the building blocks of our tool to find minimal cardinality diagnosis. Beyond the presentation in [47] here we formalize each one of the components of our approach, present algorithms, prove correctness and demonstrate through a running example. Section 6 encapsulates the different components into one process. Section 7 shows an efficient method to extend the search of a single minimal cardinality diagnosis to all minimal cardinality diagnoses. Evaluation of our approach is given in Section 8. And in Section 9 conclusions.

2 Related work

Many of the existing diagnosis techniques propose to apply a combination of deterministic reasoning and search algorithms. One classic approach involves a two stage process. First, it identifies conflict sets, each of which includes at least one fault. Then, it applies a hitting set algorithm to compute sets of multiple faults that explain the observation [19, 63]. These methods guarantee sound diagnoses, and some of them are even complete. However, they tend to fail for large systems due to infeasible runtime or space requirements.

Another approach proposes to consider the diagnosis problem in terms of inductive learning. Here, one tries to learn relations between the symptoms and the faults [48]. One disadvantage of most works in this approach is that they learn only a single fault rather than multiple faults [4]. In addition, inductive learning methods do not guarantee sound diagnoses nor completeness. We, on the other hand, propose a method which addresses multiple faults and guarantee sound and completed minimal cardinality diagnoses.

Feldman et al. [24] propose a stochastic diagnosis algorithm, called SAFARI. Although this method is not guaranteed to return diagnoses of minimal cardinality, it presents solutions which are close to minimal cardinality in very low runtime. In Section 8, we demonstrate that our approach outperforms SAFARI in terms of runtime, and also guarantees that minimal cardinality diagnoses are returned.

Compilation-based methods have also been proposed in the MBD context. Torasso and Torta apply BDDs to compile the model [60]. Darwiche [17] compiles a system description into Decomposable Negation Normal Form (DNNF) where a minimal cardinality diagnosis can be found in time that is polynomial in the size of the DNNF. However, the size of the DNNF may grow exponentially and is shown to become a bottleneck [53].

Siddiqi [53] suggests to optimize MBD by identifying components that *dominate* others. We adopt this idea and apply it in our SAT-based approach. Another compilation-based diagnosis algorithm is the HA* algorithm [25]. HA* is designed to exploit a given hierarchy of the diagnosed system. This is done by converting a given system hierarchy to a DNF hierarchy. Each element in this DNF hierarchy is then solved by a simple best-first search using as a heuristic function given a prior probability on the health of the system components. In Section 8, we demonstrate that our approach substantially outperforms HA*.

3 Model-Based Diagnosis

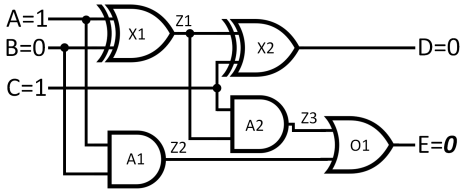
Model Based Diagnosis problems arise when the normal behavior of a system is violated due to faulty components as indicated by certain observations. We focus on *weak fault models* (WFM), which ignore the mode of abnormal behavior of components. An MBD problem is specified as a triplet $\langle SD, COMPS, OBS\rangle$ where: SD is a system description, $COMPS$ is a set of components, and OBS is an observation. The system description takes into account that some components might be abnormal (faulty). This is specified by an unary predicate $h(\cdot)$ on components such that $h(c)$ is true when component c is healthy, while $\neg h(c)$ is true when c is faulty. Denoting the correct behavior of c as a propositional formula, φ_c, SD is given formally as

$$SD = \bigwedge_{c \in COMPS} h(c) \Rightarrow \varphi_c$$

Namely, each component which is not faulty follows its correct behavior. A diagnosis problem (DP) arises when, under the assumption that none of the components are faulty, there is an inconsistency between the system description and the observations [19, 50].

Definition 1. [Diagnosis Problem]. *Given an MBD problem, $\langle SD, COMPS, OBS\rangle$, a diagnosis problem arises when*

$$SD \wedge \bigwedge_{c \in COMPS} h(c) \quad \wedge OBS \vdash \bot$$



$\mathit{COMPS} = \{X_1, X_2, A_1, A_2, O_1\}$
$\mathit{OBS} = \{A, \neg B, C, \neg D, \neg E\}$

Fig. 1. MBD: A full adder.

For example, a diagnosis problem arises for the MBD of Figure 1 as normal behavior would give output $E = 1$. Once there is an inconsistency, a diagnosis algorithm tries to find a subset $\Delta \subseteq \mathit{COMPS}$ which, if assumed faulty, explains the observation.

Definition 2. [Diagnosis] *Given an MBD problem,* $\langle \mathit{SD}, \mathit{COMPS}, \mathit{OBS} \rangle$*, the set of components* $\Delta \subseteq \mathit{COMPS}$ *is a* diagnosis *if*

$$SD \wedge \bigwedge_{c \in \Delta} \neg h(c) \quad \wedge \bigwedge_{c \notin \Delta} h(c) \ \wedge \ OBS \quad \nvdash \bot$$

We say that Δ *is a* minimal diagnosis *if no proper subset* $\Delta' \subset \Delta$ *is a diagnosis, and that* Δ *is a* minimal cardinality diagnosis *if no other diagnosis* $\Delta' \subseteq \mathit{COMPS}$ *exists such that* $|\Delta'| < |\Delta|$.

For the MBD of Figure 1, Δ_1=$\{X_1, X_2\}$, Δ_2=$\{O_1\}$, Δ_3=$\{A_2\}$ are minimal diagnoses, and Δ_2, Δ_3 are minimal cardinality diagnoses, as there is no smaller diagnosis.

4 The Standard Approach to SAT-Based MBD

The standard encoding of an MBD problem $\langle \mathit{SD}, \mathit{COMPS}, \mathit{OBS} \rangle$ to Boolean Satisfiability (as introduced in [54]) associates each component $c \in \mathit{COMPS}$ with a propositional formula, φ_c, denoting its correct behavior, and with a Boolean variable, H_c, signifying if c is "healthy". Viewing the observation as a propositional statement, an encoding is obtained by specifying

$$\varphi = \mathit{OBS} \ \wedge \bigwedge_{c \in \mathit{COMPS}} H_c \Rightarrow \varphi_c \tag{1}$$

In a satisfying assignment for φ, the health variables assigned the value *false* determine a (not necessarily minimal) diagnosis Δ.

For example, consider the MBD problem of Figure 1, and let $\texttt{comp}(\texttt{A}, \texttt{B}, \texttt{C})$ with $\texttt{comp} \in \{\texttt{and}, \texttt{or}, \texttt{xor}\}$ denote the propositional formula describing the behaviour of the component which is an "`and`", "`or`" or "`xor`" gate with inputs $\texttt{A}, \texttt{B}$ and output $\texttt{C}$. So, Equation (1) takes the form:

$$\varphi = \begin{pmatrix} \mathtt{A} \wedge \neg \mathtt{B} \wedge \mathtt{C} \wedge \neg \mathtt{D} \wedge \neg \mathtt{E} \wedge \mathtt{H}_{\mathtt{X}_1} \Rightarrow \mathtt{xor}(\mathtt{A}, \mathtt{B}, \mathtt{Z}_1) \wedge \\ \mathtt{H}_{\mathtt{A}_1} \Rightarrow \mathtt{and}(\mathtt{A}, \mathtt{B}, \mathtt{Z}_2) \quad \wedge \mathtt{H}_{\mathtt{X}_2} \Rightarrow \mathtt{xor}(\mathtt{Z}_1, \mathtt{C}, \mathtt{D}) \wedge \\ \mathtt{H}_{\mathtt{A}_2} \Rightarrow \mathtt{and}(\mathtt{Z}_1, \mathtt{C}, \mathtt{Z}_3) \quad \wedge \mathtt{H}_{\mathtt{O}_1} \Rightarrow \mathtt{or}(\mathtt{Z}_2, \mathtt{Z}_3, \mathtt{E}) \end{pmatrix} \tag{2}$$

This formula is satisfied by the assignment of variables $\{\mathtt{A}, \mathtt{C}, \mathtt{H}_{\mathtt{A}_1}, \mathtt{H}_{\mathtt{A}_2}, \mathtt{H}_{\mathtt{O}_1}\}$ to *true* and of variables $\{\mathtt{B}, \mathtt{D}, \mathtt{E}, \mathtt{Z}_1, \mathtt{Z}_2, \mathtt{Z}_3, \mathtt{H}_{\mathtt{X}_1}, \mathtt{H}_{\mathtt{X}_2}\}$ to *false*. This assignment indicates that $\Delta = \{\mathtt{X}_1, \mathtt{X}_2\}$ is a diagnosis.

To obtain a minimal cardinality diagnosis we seek a satisfying assignment with a minimal number of health variables taking value *false*. For example the assignment of variables $\{\mathtt{A}, \mathtt{C}, \mathtt{Z}_1, \mathtt{Z}_3, \mathtt{H}_{\mathtt{X}_1}, \mathtt{H}_{\mathtt{X}_2}, \mathtt{H}_{\mathtt{A}_1}, \mathtt{H}_{\mathtt{A}_2}\}$ to *true* and of variables $\{\mathtt{B}, \mathtt{D}, \mathtt{E}, \mathtt{Z}_2, \mathtt{H}_{\mathtt{O}_1}\}$ to *false* which also satisfies Equation (2) and indicates only one faulty component. This can be achieved using a MAX-SAT solver, as in [23], or as done in this thesis, by introducing a cardinality constraint which constrains the sum of the negated health variables (viewing *true* as 1 and *false* as 0).

Cardinality constraints are encoded to CNF using standard techniques, as described for example in [22, 2, 14]. In this setting, for a constant k, The formula

$$\varphi_k = \varphi \wedge (\sum_{c \in COMPS} \neg H_c \leq k) \tag{3}$$

is satisfied only if at most k health variables take the value *false*. More specifically, we seek a minimal value of k such that (the CNF corresponding to) φ_k is satisfiable. This involves iterating over calls to the SAT solver with formulae φ_k for decreasing values of k until φ_k is satisfiable but φ_{k-1} is not.

5 Our Approach to SAT-Based MBD

Our approach to encoding an MBD problem $\langle SD, COMPS, OBS \rangle$ to SAT proceeds as follows: First, we adopt a finite domain constraint based representation to express the basic model. Second, we analyze the structure and substructures of the SD to introduce additional (redundant) constraints that will later boost the search for a minimal cardinality analysis. Third, we introduce constraints to model the given observation OBS with an additional constraint that imposes a bound on the cardinality of the diagnosis (the number of unhealthy gates). This additional constraint reduces the subsequent number of iterations in search of the minimal cardinality diagnosis. Each such iteration involves a call to the underlying SAT solver, so reducing this number is important. Given all of these constraints, we apply BEE, a finite domain constraint compiler [45] to simplify and encode them to a corresponding CNF. Finally we apply a SAT solver to seek a suitable satisfying assignment and solve the problem. In the rest of this section we describe these phases in more detail.

5.1 The basic model for MBD

We build on the standard approach, as in Equation (1). However, we observe that for model based diagnosis in the weak fault model with a single observation

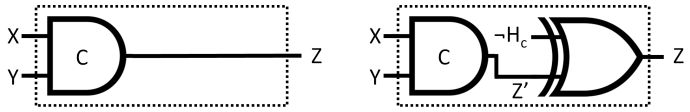


Fig. 2. Modeling component c by composition with `xor`

and when searching for a minimal diagnosis, the behaviour of a faulty component can be assumed to produce an output opposite to that of its normal behavior. This is because any diagnosis that assumes that a component c is faulty but still produces its normal output can be replaced by a smaller diagnosis that does not contain c. Thus, if Δ is a minimal diagnosis (i.e, no subset of Δ is a diagnosis), this means that all the components in Δ are assumed to produce the opposite of their normal output. In this thesis we focus on minimal cardinality diagnoses, which are in particular also minimal subset, so we modify Equation (1) as follows.

$$\varphi = \mathit{OBS} \ \wedge \bigwedge_{c \in \mathit{COMPS}} H_c \Leftrightarrow \varphi_c \tag{4}$$

We model the behaviour ($H_c \Leftrightarrow \varphi_c$) of a possibly faulty component c as if encapsulated together with a `xor` gate as illustrated in Figure 2. Here, the output of the encapsulated component is the `xor` of the usual output of c and its negated health variable $\neg H_c$. One can observe that if H_c is true then this composition is equivalent to the normal behavior of c, otherwise it is equivalent to component c with a negated output.

Our decision to model the relation between a component c and its health variable H_c by introducing an additional `xor` gate (instead of just introducing CNF clauses to directly encode $H_c \Leftrightarrow \varphi_c$) has two motivations: (1) to improve CNF encodings we provide tools to reason about, and simplify circuit components — so there is an advantage to a uniform representation where all of the logic is expressed in the circuit itself; and (2) the underlying SAT solver that we apply, CryptoMiniSat [56], offers direct support for `xor` clauses. Because of (1) our MBD problem is more amenable to simplification, and because of (2) the underlying SAT solver can optimize the search for a satisfying assignment.

As in Equation (2), we write $\mathtt{comp}(\mathtt{A},\mathtt{B},\mathtt{C})$ with $\mathtt{comp} \in \{\mathtt{and},\mathtt{or},\mathtt{xor}\}$ to represent a component which is an "`and`", "`or`" or "`xor`" gate with inputs $\mathtt{A},\mathtt{B}$ and output $\mathtt{C}$. We also write $\mathtt{comp}_\mathtt{H}(\mathtt{A},\mathtt{B},\mathtt{C})$ to represent the corresponding encapsulated component with a health variable $\mathtt{H}$. So,

$$\mathtt{comp}_\mathtt{H}(\mathtt{A},\mathtt{B},\mathtt{C}) = \mathtt{comp}(\mathtt{A},\mathtt{B},\mathtt{C}') \wedge \mathtt{xor}(\neg\mathtt{H},\mathtt{C}',\mathtt{C})$$

and we view

$$\boxed{\mathtt{comp}_\mathtt{H}(\mathtt{A},\mathtt{B},\mathtt{C})} \qquad \text{(Constraints 1)}$$

as a constraint on the Boolean variables $\mathtt{A},\mathtt{B},\mathtt{C}$ and $\mathtt{H}$. Given this notation, the system depicted as Figure 1 is modeled by the following constraints:

$$\mathtt{xor}_{\mathtt{H}_{\mathtt{X}_1}}(\mathtt{A},\mathtt{B},\mathtt{Z}_1) \ \wedge \mathtt{and}_{\mathtt{H}_{\mathtt{A}_1}}(\mathtt{A},\mathtt{B},\mathtt{Z}_2) \wedge \mathtt{xor}_{\mathtt{H}_{\mathtt{X}_2}}(\mathtt{Z}_1,\mathtt{C},\mathtt{D}) \wedge$$
$$\mathtt{and}_{\mathtt{H}_{\mathtt{A}_2}}(\mathtt{Z}_1,\mathtt{C},\mathtt{Z}_3) \wedge \mathtt{or}_{\mathtt{H}_{\mathtt{O}_1}}(\mathtt{Z}_2,\mathtt{Z}_3,\mathtt{E})$$

Finally, we add to the constraints representing the circuit components an additional cardinality constraint:

$$\boxed{\texttt{sum\_leq}(\{\ \neg \mathtt{H_c} \mid c \in \mathit{COMPS}\ \}, \mathtt{k})} \qquad \text{(Constraint 2)}$$

to specify for an integer constant k that the number of faulty components must be at most k. For example, for the system depicted as Figure 1 and a constant k, we introduce the constraint $\texttt{sum\_leq}(\{\neg \mathtt{H_{X_1}}, \neg \mathtt{H_{X_2}}, \neg \mathtt{H_{A_1}}, \neg \mathtt{H_{A_2}}, \neg \mathtt{H_{O_1}}\}, \mathtt{k})$. Later we will require to satisfy the constraints of the model and also to minimize the value of k.

To summarize this presentation of the basic model, we show the complete constraint model for the minimal cardinality diagnosis of the MBD problem of Figure 1. For an integer value k a solution of these constraints is a diagnosis of cardinality $\leq k$:

$$\begin{aligned}
&\mathtt{xor}_{\mathtt{H_{X_1}}}(\mathtt{A}, \mathtt{B}, \mathtt{Z_1}) \ \wedge \mathtt{and}_{\mathtt{H_{A_1}}}(\mathtt{A}, \mathtt{B}, \mathtt{Z_2}) \wedge \mathtt{xor}_{\mathtt{H_{X_2}}}(\mathtt{Z_1}, \mathtt{C}, \mathtt{D}) \wedge \\
&\mathtt{and}_{\mathtt{H_{A_2}}}(\mathtt{Z_1}, \mathtt{C}, \mathtt{Z_3}) \wedge \mathtt{or}_{\mathtt{H_{O_1}}}(\mathtt{Z_2}, \mathtt{Z_3}, \mathtt{E}) \ \wedge \\
&\mathtt{sum_leq}(\{\neg \mathtt{H_{X_1}}, \neg \mathtt{H_{X_2}}, \neg \mathtt{H_{A_1}}, \neg \mathtt{H_{A_2}}, \neg \mathtt{H_{O_1}}\}, \mathtt{k}) \ \wedge \\
&\mathtt{A} = 1 \ \wedge \ \mathtt{B} = 0 \ \wedge \ \mathtt{C} = 1 \ \wedge \ \mathtt{D} = 0 \ \wedge \ \mathtt{E} = 0
\end{aligned}$$

This type of constraint model can be solved by encoding it to a CNF formula and then applying a SAT solver. By repeatedly seeking a solution for descreasing values of k we can find a minimal cardinality diagnosis. However, we do not apply this basic modeling. Instead we further refine it as described in the rest of this section.

5.2 Introducing (redundant) Cardinality Constraints

Reasoning about relations between the components in a system description *SD* enables to infer (cardinality) constraints about the number of unhealthy components in certain subsystems of *SD*. These constraints when compiled into the CNF, help boost the search, by the SAT solver, for a minimal cardinality diagnosis. To this end, we adopt the concept of "gate domination" introduced in [37] and applied also by Siddiqi who further introduces the notion of a "cone". The following wording is taken from [53] in a setting where the system is a logical circuit and the components are its gates.

Definition 3 (Dominator and Cone). *A gate X in the fan-in region of gate G is dominated by G, and conversely G is a dominator of X, if any path from X to an output of the circuit contains G. The cone corresponding to a gate G is the set of gates dominated by G. A maximal cone is one that is either contained in no other cone or contained in exactly one other cone which is the entire circuit.*

For example, in the circuit depicted as Figure 1, the components $\{\mathtt{A_1}, \mathtt{A_2}, \mathtt{O_1}\}$ form a cone, since any path from $\mathtt{A_1}$ or from $\mathtt{A_2}$ to a system output contains $\mathtt{O_1}$. Here $\mathtt{O_1}$ is the dominator and $\mathtt{A_1}$ and $\mathtt{A_2}$ are the dominated gates.

Cones are single-output sub-circuits and as such, a minimal cardinality diagnosis will always, independent of the observation, indicate at most one unhealthy component per cone. Hence, given a partition to (maximal) cones, we can introduce cardinality constraints to state that each cone contains at most one unhealthy gate. These constraints, though redundant, improve considerably the search for minimal cardinality diagnosis. This because search can backtrack as soon as it indicates two unhealthy components in a cone. For the cone $\{\mathtt{A_1}, \mathtt{A_2}, \mathtt{O_1}\}$ in the circuit from Figure 1 we add the redundant constraint $\mathtt{sum\_leq}(\{\neg \mathtt{H_{A_1}}, \neg \mathtt{H_{A_2}}, \neg \mathtt{H_{O_1}}\}, \mathtt{1})$ which specifies that at most one of the corresponding components is not healthy.

In Section 8 we show that reasoning about cones to introduce this kind of redundant cardinality constraints indeed improves considerably the search for minimal cardinality diagnosis.

Motivated by the utility of partitioning a system to cones, we seek a more general partitioning, which enables to apply similar cardinality constraints to larger subsystems of components. To this end we introduce the notion of a "section". We denote by $sysout(c)$ the set of system outputs which occur at the end of a path from a component c. As an example, in the circuit depicted in Figure 3 $sysout(C_1) = \{O_2, O_3\}$ and $sysout(C_5) = \{O_1, O_2\}$.

Definition 4 (Section). *Given a system description* SD *with components* COMPS *we define a disjoint partitioning* $\mathit{COMPS} = S_1 \cup S_2 \cup \cdots \cup S_n$ *such that for every* $c_1, c_2 \in \mathit{COMPS}$, c_1 *and* c_2 *are in the same section* S_i *if and only if* $sysout(c_1) = sysout(c_2)$.

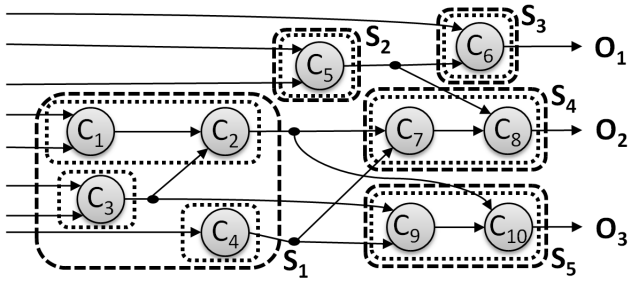


Fig. 3. Partitioning a system to cones and sections.

Figure 3 shows a partitioning of a circuit to maximal cones and sections. The cones are depicted with dotted lines, and the sections with dashed. We observe that partitioning a circuit to sections can be done in polynomial time as demonstrated by Algorithm 1 presented below. Given a partitioning $\{S_1, \ldots, S_n\}$ to sections, we introduce to the constraint model the following constraints which further improve the encoding and hence the subsequent search for minimal cardinality diagnosis. For each section S_i, the constraint

$$\boxed{\mathtt{sum\_leq}(\{\ \neg \mathtt{H_c} \mid c \in S_i\ \}, \mathtt{b_i})} \qquad \text{(Constraints 3)}$$

expresses that the sum of the negated health variables in S_i is bound from above by a constant b_i that is the smaller of the following two bounds on the number of unhealthy components in section S_i: (a) viewing S_i as a system, following the assertion of de Kleer, the number of outputs from S_i is a bound, and (b) taking some component $c \in S_i$, $|sysout(c)|$ is a bound. Note that by Definition 4, this value is equivalent for any $c \in S_i$. We state that this is correct below in Proposition 1.

Reasoning about constraints on the number of faulty components per section contributes to the MBD encoding in another way. Consider again Constraint 2 which talks about the total number of faulty gates in the system. When encoding this total sum (to constrain that it must take a value less equal to k), we take advantage of the partial sums on the numbers of faulty gates per section already encoded for Constraints 3.

To illustrate the utility of sections, consider the section labeled S_1 in the system given as Figure 3. where sections are depicted in dashed lines and cones in doted lines. It has 3 outputs, but each component $c \in S_1$ has only 2 corresponding system outputs ($|sysout(c)| = 2$). Hence, 2 is an upper bound on the number of unhealthy gates in S_1. This is also an improvement over the reasoning with cones where the bound on the number of unhealthy components in S_1 is 3 (since there are three cones).

The following justifies Constraints 3.

Proposition 1. *Let $\langle SD, COMPS, OBS\rangle$ be an MBD problem, $S \subseteq SD$ be a section, $c \in S$ be a component and Δ be a minimal cardinality diagnosis. Then, the number of unhealthy gates (from Δ) in S is at most $|sysout(c)|$.*

Proof: Assume the premise of the proposition, denote $|\Delta| = k$ and $|S \cap \Delta| = t$ (so $t \leq k$). Assume for contradiction that $t > |sysout(c)|$. We construct a diagnostic Δ' with less than k unhealthy gates. First note the obvious: that given Δ we can propagate the observed system inputs to the system outputs where in each step we choose a component with known inputs and produce its normal output if the component is healthy, or its opposite to normal output otherwise. Because Δ is a diagnosis this process will result in no contradictions between propagated outputs and the observed outputs.

Now, take $\Delta' = \Delta \setminus S$. This is not (yet) a diagnosis. With Δ' propagate the observed system inputs in the same way as before with Δ. Now because Δ' is not a diagnosis there will be so "flipped" system outputs (those which contradict the observed outputs). Each such "flipped" output o must be due to one of the unhealthy gates in S that was marked healthy in Δ' and so we have $o \in sysout(c)$. Now consider the gate g which outputs o. If $g \in \Delta'$, then remove it; and if $g \notin \Delta'$, then add it. So, now Δ' is a diagnosis and $k' = |\Delta'| \leq k - t + |sysout(c)| < k$. □

Algorithm 1 describes how to partition a circuit to sections as defined in Definition 4. Denoting the components and outputs of the circuit as $COMPS = \{c_1, \ldots, c_n\}$ and $OUTS = \{o_1, \ldots, o_m\}$, an $n \times m$ Boolean matrix $\mathbf{b}$ is computed so that $b_{ij} = true$ if $o_j \in sysout(c_i)$ and *false* otherwise. Figure 4 shows an example of this matrix $\mathbf{b}$ for the circuit in Figure 3. So, by Definition 4, a pair of

components c_i, c_j are in the same section if and only if row i and row j in matrix $\mathbf{b}$ are identical. The computational complexity of this partitioning process is the complexity of running a graph search algorithm for every system output and is in the worst case $O(n^2 \cdot m)$. Pseudo-code for this partitioning is given in Algorithm 1. The algorithm returns a mapping from components to bit vectors which can be seen as section identifiers. So, the algorithm returns a mapping of components to sections.

The algorithm returns a mapping from components to bit vectors which can be seen as section identifiers. So, the algorithm returns a mapping of components to sections.

Algorithm 1: partitioning a circuit to sections.

Input: A circuit (view it as a graph)
Output: A partitioning of the circuit to sections
Denote: $\mathit{COMPS} = \{c_1, \ldots, c_n\}$ the circuit components
$\mathit{OUTS} = \{o_1, \ldots, o_m\}$ the circuit outputs
$\mathbf{b} = (b_{ij})$ an $n \times m$ Boolean matrix

1 **foreach** $(o_j \in \mathit{OUTS})$ **do**
2 | apply DFS on the reverse edges of the circuit, with $\mathit{source} = o_j$
3 | **foreach** $(c_i \in \mathit{COMPS})$ **do** $b_{ij} = (c_i$ is reachable from $o_j)$
4 **end**
5 **return** $\{\ c_i \mapsto \langle b_{i1}, \ldots, b_{im} \rangle \mid 1 \le i \le n\ \}$

Example 13 (partition to sections). Consider the (abstract) circuit depicted as Figure 3 where $\mathit{COMPS} = \{c_1, \ldots, c_7\}$ and $\mathit{OUTS} = \{o_1, \ldots, o_3\}$. The Boolean matrix evaluated by application of Algorithm 1 is illustrated in Figure 4.

There is another benefit of partitioning to sections: the identification of cones may be performed "per section" which is more efficient. This works because, if gate X is dominated by gate G, then $\mathit{sysout}(X) = \mathit{sysout}(G)$ implying that the gates of a cone are always in the same section. For example, in Figure 3 component $\mathtt{C}_1$ is dominated by $\mathtt{C}_2$ and $\mathit{sysout}(\mathtt{C}_1) = \mathit{sysout}(\mathtt{C}_2) = \{\mathtt{O}_1, \mathtt{O}_2\}$.

$$\mathbf{b} = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{matrix} \} \mathtt{S}_1 \\ \\ \\ \\ \} \mathtt{S}_2 \\ \} \mathtt{S}_3 \\ \} \mathtt{S}_4 \\ \\ \} \mathtt{S}_5 \end{matrix}$$

Fig. 4. Partitioning the system from Figure 3 to sections

The recursively defined Algorithm 2 shows how to compute cones given a partition to sections. It computes the set of dominators for a component c in a section S of the circuit. We denote by $\mathit{succ}(c)$ the set of components that c feeds into directly. If $c \in S$ feeds into a component that is not in S then it is only dominated by itself. Otherwise, c is dominated by $c' \in S$ only if c' is dominated by all elements of $\mathit{succ}(c)$. It is straightforward to implement

Algorithm 2 efficiently using a memoing table to avoid recomputing dominators for components already encountered. Since a circuit is a directed acyclic graph, the recursion in Algorithm 2 will halt when a "leaf" node is reached. Thus the complexity of calculating the dominators of every component c in a section S is $O(|S|^2)$. Given the sets of dominators per component, it is straightforward to specify the set of maximal cones. A component c is a dominator of a maximal cone, if it is only dominated by itself, and the maximal cone corresponding to such a c is the set of components which have c as dominator.

To find all cones in a system, Algorithm 2 is applied once per component per section, and the cost depends on the size of the largest section. In contrast, without the partition to sections, the same algorithm is applied, but considering all of the components in the system instead of all the components in a section. Practice shows that the partition to sections benefits the computation of cones.

Algorithm 2: dominators (component c, section S, circuit C)

Data: component c in section S of circuit C
Output: The set of dominators of c
Denote: $succ(c) = \{\ c' \in C \mid \text{the output of } c \text{ is an input to } c'\ \}$
if $(succ(c) \subseteq S)$ **then**
  return $\{c\} \cup \bigcap_{c' \in succ(c)} \text{dominators}(c', S, C)$
else return $\{c\}$

5.3 Introducing Constraints to Break Symmetry

Symmetry breaking (e.g. [16]) is about adding constraints to select a particular solution in case it is just as good as some other set of solutions. Consider a cone C in SD. Any minimal cardinality diagnosis of SD will indicate at most one unhealthy component in C. Without loss of generality, we may assume that all dominated components in C are healthy. This is correct because if X is unhealthy in some minimal cardinality diagnosis and dominated by G, then G must be healthy. So, there exists another minimal cardinality diagnosis where X is healthy and G is not. For example, in the circuit depicted in Figure 3, if there exists a minimal diagnosis that assumes C_7 is unhealthy, then there exists a minimal diagnosis in which C_7 is healthy but C_8, which dominates C_7, is unhealthy.

Based on this observation we can restrict the search for so-called "top-level" minimal cardinality diagnoses. The following is equivalent to the corresponding definition from [53].

Definition 5 (top-level diagnosis (TLD)). *We say that a minimal cardinality diagnosis is top-level if it does not contain any dominated gates.*

The following justifies the focus on top-level diagnoses:

Proposition 2. *Let Δ' be a minimal cardinality diagnosis for a given MBD problem. Then there is a top-level diagnosis Δ', of the same cardinality.*

Proof: Straightforward: to obtain Δ, replace each dominated component from Δ' by its corresponding dominator. □

To restrict the search to top-level minimal cardinality diagnoses we add the following constraints where D denotes the set of dominated gates.

$$\boxed{\bigwedge_{c \in D} H_c} \qquad \text{(Constraint 4)}$$

Introducing constraints to indicate healthy components reduces the number of (unassigned) health variables and hence boosts the search for minimal cardinality diagnosis.

5.4 Modeling the Observation and Further Boosting the Search

Let OBS^+ and OBS^- denote the sets of variables assigned true and false in OBS, respectively. Then, to model the observation we add the obvious constraints.

$$\boxed{\bigwedge_{x \in OBS^+} x \quad \wedge \quad \bigwedge_{x \in OBS^-} \neg x} \qquad \text{(Constraint 5)}$$

To improve the search for a minimal cardinality diagnosis one can introduce an upper bound on the minimal cardinality: applying the assertion from [18], the number of outputs in a system is an upper bound on the minimal cardinality. Siddiqi proposes [52] to improve this bound for a given observation by propagating the input values through the system, and taking as an upper bound the number of contradictions between the observed and the propagated outputs. For example, considering the MBD problem from Figure 1, $k = 2$ is an upper bound on the size of a minimal cardinality diagnosis because the system has 2 outputs. Siddiqi's proposal states that also 1 is an upper bound because when propagating the inputs through the system there is only one contradiction to the observed outputs.

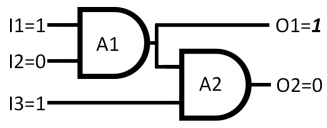


Fig. 5. Minimal cardinality diagnosis is of size 2, but propagating observed inputs leads to 1 contradiction to the observed outputs.

While Siddiqi's proposal is intuitively appealing, it is correct only in case that no observed output is also input to another circuit, and in fact the results in [52] are restricted to systems where this is the case. This does not work for the example in Figure 5. Propagating the observed inputs through the circuit assigns 0 to both outputs indicating a single contradiction with the observation (on O_1). However, the smallest diagnosis for this example has cardinality 2. This example is not contrived: 83 of the 350 observations for system 74181 of the 74XXX benchmark, exhibit a minimal cardinality diagnosis larger than the bound obtained when counting conflicts between propagated and observed outputs.

Algorithm 3 is about computing an upper bound on the number of faulty components by propagating observed inputs and counting conflicts. The algorithm computes a diagnosis, Δ and $|\Delta|$ is thus an upper bound on the minimal cardinality of a diagnosis. The basic idea is to propagate inputs as long as they do not contradict observed, or other already computed, outputs. The components in the system are processed one at a time. At line 3 we select some component c whose inputs are already determined (initially only the system inputs are determined). For this c we consider its 'already determined' output, o_{obs}, and denote $o_{obs} = \perp$ if its output is not yet determined. We also consider its propagated output o_{prop} which is obtained by propagating the inputs through c assuming that c is healthy. Now there are three cases (lines 6–10): if c has no already determined output then we fix its output to o_{prop} and mark c as healthy. If c has an already determined output and it is consistent with o_{prop} then we also mark c as healthy. Otherwise we mark c as not healthy, but do not propagate its output (which is already determined).

Algorithm 3: Find a diagnosis Δ (and upper bound $|\Delta|$ on min. card.)

Input: A circuit with components, *COMPS*, and observation, *OBS*
Output: A diagnosis Δ

```
1  S ← COMPS,   Δ = ∅
2  while (S ≠ ∅) do
3  |  select c ∈ S such that the inputs of c are determined
4  |  o_obs ← the value on the output of c (⊥ if it is undefined)
5  |  o_prop ← the value if propagating the inputs of c (assume c is healthy)
6  |  if (o_obs = ⊥) then
7  |  |  set output of c to o_prop and mark c as healthy
8  |  else
9  |  |  if (o_obs = o_prop) then mark c as healthy
10 |  |  else mark c as faulty and Δ = Δ ∪ {c}
11 |  end
12 |  S ← S \ {c}
13 end
14 return Δ
```

When Algorithm 3 terminates we have marked all components as healthy or faulty and have in fact determined a correct diagnosis. As such the number of components marked as faulty is an upper bound on a minimal cardinality diagnosis. Note that this procedure is correct also when given probes (observed values on the outputs from internal gates). Assuming that the components are maintained in a data-structure where components are sorted (topologic) according to their depth, Algorithm 3 is performed as a single linear traversal of this data-structure with complexity $O(|COMPS|)$.

As an example of Algorithm 3, consider the circuit in Figure 5. Propagating the inputs of gate A_1 gives the output 0 in contradiction to the observation on

O_1. Hence, we mark A_1 as unhealthy and propagate the observation $O_1 = 1$ as an input to A_2 together with $I_3 = 1$. This results in an additional contradiction to the observation $O_2 = 0$ and so we mark A_2 as unhealthy too, and report $\Delta = \{A_1, A_2\}$ hence the value 2 as an upper bound for the minimal cardinality.

Proposition 3. *Application of Algorithm 3 indicates an upper-bound on the minimal cardinality diagnosis.*

For the proof we note that Definition 2 works also in case that *OBS* does not assign values to all of the outputs of a system.

Proof Consider a run of the algorithm, and let $c_1, c_2, \ldots, c_n$ be the ordered sequence of components as selected at line 3. We focus on sub-circuits with the sets of components selected up till stage i: $\mathit{COMPS}_i = \{c_1, \ldots, c_i\}$. Let Δ_i be the corresponding values of Δ at each step i. Observe that upon completion of the algorithm, $\mathit{COMPS} = \mathit{COMPS}_n$ and $\Delta = \Delta_n$. The proof is by induction on the steps of the algorithm. In other words on the number of components already selected at line 3. We show that for each i, Δ_i is a correct diagnosis for the sub-circuit with components COMPS_i and the observation *OBS*.

base case: Consider the system with $\mathit{COMPS}_1 = \{c_1\}$ and its propagated output o_{prop}. If the output of c_1 is an observed value o_{obs} and $o_{prop} \neq o_{obs}$ then the algorithm sets $\Delta_1 = \{c_1\}$ and this is a correct diagnosis; otherwise its sets $\Delta_1 = \emptyset$ which is a correct diagnosis.

step: Assume for $1 < k < n$ that Δ_k is a correct diagnosis for COMPS_k with *OBS*. We show that Δ_{k+1} is a correct diagnosis for COMPS_{k+1}.

Consider component c_{k+1} with its propagated output o_{prop} and let o_{obs} be defined as in the algorithm. The algorithm distinguishes three cases:

1. if o_{obs} is not an observed value then $o_{obs} = \perp$ (because the fan-in for any circuit is 1). In this case $\Delta_{k+1} = \Delta_k$ is a correct diagnosis for COMPS_{k+1}.
2. if o_{obs} is an observed value and $o_{prop} = o_{obs}$. Also in this case $\Delta_{k+1} = \Delta_k$ is a correct diagnosis for COMPS_{k+1}.
3. if o_{obs} is an observed value and $o_{prop} \neq o_{obs}$. Then $\Delta_{k+1} = \Delta_k \cup \{c_{k+1}\}$. Because Δ_k is a correct observation for COMPS_k and marking c_{k+1} as faulty settles the discrepancy between o_{prop} and o_{obs}, hence, Δ_{k+1} is a correct diagnosis for COMPS_{k+1}.

□

Let k_{UB} be the bound found by application of Algorithm 3. We refine Constraint 2 and introduce instead:

$$\boxed{\texttt{sum\_leq}(\{\ \neg \mathtt{H_c} \mid c \in \mathit{COMPS}\ \}, \mathtt{k_{UB}})} \qquad \text{(Constraint 2')}$$

To appreciate the impact of Algorithm 3 we note that, for the ISCAS-85 and 74XXX benchmarks, there are many instances for which Algorithm 3 provides a bound identical to the actual minimal cardinality. For such observations, our SAT-based approach still needs to validate that there is no diagnosis of smaller size.

5.5 Compiling Constraints to CNF

In Part 1 of this thesis we introduce BEE, a compiler that encodes finite domain constraints to CNF. Besides facilitating the encoding process, this compiler also applies partial evaluation and other optimizations to simplify the constraints before encoding them to CNF. In particular, it applies "equi-propagation" [46] which is the process of identifying equalities between literals (and constants) implied by other such equations and a given constraint. If $X{=}L$ is implied by a constraint (where X is a variable and L is a literal or a Boolean constant), then all occurrences of X can be replaced by L, reducing the number of variables in the subsequent CNF encoding. We illustrate constraint simplification for the diagnosis of the circuit in Figure 1.

$$
\begin{array}{ll}
(\mathbf{1}) & \mathtt{xor}_{\mathtt{H}_{\mathtt{X}_1}}(\mathtt{A},\mathtt{B},\mathtt{Z}_1) \ \wedge \mathtt{and}_{\mathtt{H}_{\mathtt{A}_1}}(\mathtt{A},\mathtt{B},\mathtt{Z}_2) \wedge \mathtt{xor}_{\mathtt{H}_{\mathtt{X}_2}}(\mathtt{Z}_1,\mathtt{C},\mathtt{D}) \wedge \\
(\mathbf{2}) & \mathtt{and}_{\mathtt{H}_{\mathtt{A}_2}}(\mathtt{Z}_1,\mathtt{C},\mathtt{Z}_3) \wedge \mathtt{or}_{\mathtt{H}_{\mathtt{O}_1}}(\mathtt{Z}_2,\mathtt{Z}_3,\mathtt{E}) \ \wedge \\
(\mathbf{3}) & \mathtt{sum\_leq}(\{\neg\mathtt{H}_{\mathtt{X}_1},\neg\mathtt{H}_{\mathtt{X}_2},\neg\mathtt{H}_{\mathtt{A}_1},\neg\mathtt{H}_{\mathtt{A}_2},\neg\mathtt{H}_{\mathtt{O}_1}\},\mathtt{k}) \wedge \\
(\mathbf{4}) & \mathtt{A}=1 \ \wedge \ \mathtt{B}=0 \ \wedge \ \mathtt{C}=1 \ \wedge \ \mathtt{D}=0 \ \wedge \ \mathtt{E}=0 \wedge \\
(\mathbf{5}) & \mathtt{sum\_leq}(\{\neg\mathtt{H}_{\mathtt{A}_1},\neg\mathtt{H}_{\mathtt{A}_2},\neg\mathtt{H}_{\mathtt{O}_1}\},1) \wedge \\
(\mathbf{6}) & \mathtt{H}_{\mathtt{A}_1}=1 \ \wedge \ \mathtt{H}_{\mathtt{A}_2}=1
\end{array}
$$

The constraints on lines (**1**) − (**3**) comprise the basic constraint model described in Section 5.1; the constraints on line (**4**) model the observation; the constraint in line (**5**) expresses that at most one component from the cone $\{\mathtt{A}_1,\mathtt{A}_2,\mathtt{O}_1\}$ is faulty; and the constraints in line (**6**) express that without loss of generality the dominated gates $\{\mathtt{A}_1,\mathtt{A}_2\}$ are healthy. We observe the following equi-propagation steps:

1. $(\mathtt{A}=1) \ \wedge \ (\mathtt{B}=0) \ \wedge \ \mathtt{xor}_{\mathtt{H}_{\mathtt{X}_1}}(\mathtt{A},\mathtt{B},\mathtt{Z}_1) \ \models \ (\mathtt{Z}_1=\mathtt{H}_{\mathtt{X}_1})$
2. $(\mathtt{A}=1) \ \wedge \ (\mathtt{B}=0) \ \wedge \ (\mathtt{H}_{\mathtt{A}_1}=1) \ \wedge \ \mathtt{and}_{\mathtt{H}_{\mathtt{A}_1}}(\mathtt{A},\mathtt{B},\mathtt{Z}_2) \ \models \ (\mathtt{Z}_2=0)$
3. $(\mathtt{C}=1) \ \wedge \ (\mathtt{H}_{\mathtt{A}_2}=1) \ \wedge \ \mathtt{and}_{\mathtt{H}_{\mathtt{A}_2}}(\mathtt{Z}_1,\mathtt{C},\mathtt{Z}_3) \ \models \ (\mathtt{Z}_1=\mathtt{Z}_3)$
4. $(\mathtt{C}=1) \ \wedge \ (\mathtt{D}=0) \ \wedge \ \mathtt{xor}_{\mathtt{H}_{\mathtt{X}_2}}(\mathtt{Z}_1,\mathtt{C},\mathtt{D}) \ \models \ (\mathtt{Z}_1=\mathtt{H}_{\mathtt{X}_2})$
5. $(\mathtt{E}=1) \ \wedge \ (\mathtt{Z}_2=0) \ \wedge \ \mathtt{or}_{\mathtt{H}_{\mathtt{O}_1}}(\mathtt{Z}_2,\mathtt{Z}_3,\mathtt{E}) \ \models \ (\mathtt{Z}_3=-\mathtt{H}_{\mathtt{O}_1})$

From these (and the other given) equalities between literals we obtain a substitution:

$$
\theta = \left\{ \begin{array}{l} \mathtt{A}\mapsto 1,\mathtt{B}\mapsto 0,\mathtt{C}\mapsto 1,\mathtt{D}\mapsto 0,\mathtt{E}\mapsto 0,\mathtt{Z}_1\mapsto \mathtt{H}_{\mathtt{X}_1},\mathtt{Z}_2\mapsto 0, \\ \mathtt{Z}_3\mapsto \mathtt{H}_{\mathtt{X}_1},\mathtt{H}_{\mathtt{X}_2}\mapsto \mathtt{H}_{\mathtt{X}_1},\mathtt{H}_{\mathtt{A}_1}\mapsto 1,\mathtt{H}_{\mathtt{A}_2}\mapsto 1,\mathtt{H}_{\mathtt{O}_1}\mapsto -\mathtt{H}_{\mathtt{X}_1} \end{array} \right\}
$$

Applying θ to specialize the constraint system we get:

$$
\begin{array}{ll}
(\mathbf{1}) & \mathtt{xor}_{\mathtt{H}_{\mathtt{X}_1}}(1,0,\mathtt{H}_{\mathtt{X}_1}) \wedge \mathtt{and}_1(1,0,0) \quad \wedge \mathtt{xor}_{\mathtt{H}_{\mathtt{X}_1}}(\mathtt{H}_{\mathtt{X}_1},1,0) \wedge \\
(\mathbf{2}) & \mathtt{and}_1(\mathtt{H}_{\mathtt{X}_1},1,\mathtt{H}_{\mathtt{X}_1}) \wedge \mathtt{or}_{-\mathtt{H}_{\mathtt{X}_1}}(0,\mathtt{H}_{\mathtt{X}_1},0) \wedge \\
(\mathbf{3}) & \mathtt{sum\_leq}(\{\neg\mathtt{H}_{\mathtt{X}_1},\neg\mathtt{H}_{\mathtt{X}_1},0,0,\mathtt{H}_{\mathtt{X}_1}\},\mathtt{k}) \wedge \\
(\mathbf{4}) & 1=1 \ \wedge \ 0=0 \ \wedge \ 1=1 \ \wedge \ 0=0 \ \wedge \ 0=0 \wedge \\
(\mathbf{5}) & \mathtt{sum\_leq}(\{0,0,\mathtt{H}_{\mathtt{X}_1}\},1) \wedge \\
(\mathbf{6}) & 1=1 \ \wedge \ 1=1
\end{array}
$$

Now, most of the constraints are tautologies and we remove them. All that remains is:

$$(\mathbf{3}) \ \mathtt{sum\_leq}(\{\neg\mathtt{H}_{\mathtt{X}_1},\neg\mathtt{H}_{\mathtt{X}_1},\mathtt{H}_{\mathtt{X}_1}\},\mathtt{k})$$

which is satisfied for $k = 1$ when $\mathtt{H}_{\mathtt{X}_1} = \mathtt{1}$, and as implied from θ then we have $\mathtt{H}_{\mathtt{A}_1} = \mathtt{1}, \mathtt{H}_{\mathtt{A}_2} = \mathtt{1}, \mathtt{H}_{\mathtt{X}_2} = \mathtt{1}, \mathtt{H}_{\mathtt{O}_1} = \mathtt{0}$. This example illustrates how equi-propagation and partial evaluation are applied to simplify constraints prior to their encoding to CNF. We note that, as argued in [46], constraint simplification in BEE is polynomial in the size of the constraint model.

6 Single Minimal Cardinality Diagnosis: Process and Implementation

In this section we focus on the derivation of a single minimal cardinality diagnosis. We summarize the four phases of the diagnosis process in our approach. Let $\mu = \langle SD, COMPS, OBS \rangle$ be an MBD problem. In the first two phases we construct a constraint model. First, focusing on SD, to introduce constraints which are independent of the observation, and then "per observation" to introduce further constraints. In the third phase we encode the constraint model to a CNF, φ_μ, and finally in the fourth, solve φ_μ using a SAT solver to compute a minimal cardinality diagnoses.

Modeling the system ("offline"): The system SD is first preprocessed to partition it to sections and cones. Then, we model it in terms of its component Constraints 1 and introduce cardinality Constraints 3 to bound the number of unhealthy components per section. Finally, using information about cones, we add symmetry breaking Constraint 4 which assert that, without loss of generality, all dominated components are healthy. All of the system preprocessing is performed "offline", once per system.

Modeling the observation ("online"): The observation OBS is modeled by introducing Constraint 5 and an additional cardinality Constraint $2'$ to bound the total number of unhealthy components (by k_{UB}) based on Algorithm 3.

Encoding: The constraint system is then simplified "online", for each observation and encoded to a CNF φ_μ^k, applying the optimizing CNF compiler [46]. The parameter k reflects the value in the cardinality Constraint Constraint $2'$ which bounds the number of inhealthy components in a diagnosis. Initially k takes the value k_{UB}.

Solving: To compute a diagnosis, Δ, we seek a satisfying assignment for the encoding, φ_μ^k, by applying the CryptoMiniSat solver [56]. Δ is then the set of health variables assigned *false* by this assignment. Denoting $|\Delta| = k'$, we again seek a satisfying assignment, but this time for the formula $\varphi_\mu^{k'-1}$. If a satisfying assignment is found, it indicates a smaller diagnosis, Δ'. Otherwise, Δ is of minimal cardinality. This process is invoked repeatedly, each time finding a smaller diagnosis, until for some k' the formula $\varphi_\mu^{k'-1}$ is not satisfiable. Then, the diagnosis found in the previous iteration is of minimal cardinality.

To facilitate the search for a minimal cardinality diagnosis, we apply the SAT solver wrapper, SCryptoMiniSat [44]. SCryptoMiniSat takes as input a CNF formula (φ_μ^k) and the Boolean variables representing the number k. It provides a satisfying assignment which minimizes k.

7 All Minimal Cardinality Diagnosis: Process and Implementation

To find all minimal cardinality diagnoses we first observe that the set of all minimal cardinality diagnosess can be obtained by expanding the set of top-level minimal cardinality diagnoses. Given a minimal cardinality top-level diagnosis, $\Delta = \{c_1, \ldots, c_\ell\}$ consisting of ℓ dominators from corresponding cones $\{C_1 \ldots, C_\ell\}$, denote $\chi_i = \left\{ \; c_i' \in C_i \,\middle|\, \Delta \setminus \{c_i\} \cup \{c_i'\} \text{ is a diagnosis } \right\}$. We say that Δ expands to the set of minimal cardinality diagnoses defined by: $\gamma(\Delta) = \chi_1 \times \cdots \times \chi_\ell$.

Proposition 4. *Assume the context of a given MBD problem. Then Δ' is a minimal cardinality diagnosis if and only if there is a top-level minimal cardinality diagnosis Δ that expands to include Δ'.*

Proof: The proof is straightforward from the construction. However it is important to note that the sets χ_i are easy to compute given Δ: for each component $c_i' \in C_i$ checking if $\Delta \setminus \{c_i\} \cup \{c_i'\}$ is a diagnosis means propagating the observed inputs through the system, flipping the outputs when propagating through a component in $\Delta \setminus \{c_i\} \cup \{c_i'\}$ and checking if there is no conflict to the observed outputs. □

For example, consider the system from Figure 1 with the observation $\mathit{OBS} = \{A, B, C, \neg D, \neg E\}$. The cones in the system are $C_1 = \{\mathtt{X_1}\}$, $C_2 = \{\mathtt{X_2}\}$, and $C_3 = \{\mathtt{A_1}, \mathtt{A_2}, \mathtt{O_1}\}$. The corresponding MBD problem has a two top-level minimal cardinality diagnoses, $\Delta_1 = \{\mathtt{X_1}, \mathtt{O_1}\}$ and $\Delta_2 = \{\mathtt{X_2}, \mathtt{O_1}\}$. For Δ_1 we have $\gamma(\Delta_1) = \{X_1\} \times \{O_1\}$. For Δ_2 we have $\gamma(\Delta_2) = \{X_2\} \times \{A_1, O_1\}$.

The implementation applies a similar strategy as described in Section 6 where an MBD problem is encoded to a CNF formula φ_μ^k and assuming we already have a single top-level minimal cardinality diagnosis with k' faulty components.

To enumerate the set of all top-level minimal cardinality diagnoses we apply an additional functionality of SCryptoMiniSat which allows to enumerate (possibly with a specified time-out) all, or a specified number of, satisfying assignments for a given CNF. We apply this option to enumerate all satisfying assignments for $\varphi_\mu^{k'}$.

Note that the observation that a TLD can be expanded easily to all minimal cardinality diagnoses applies for any diagnosis algorithm. As such, diagnosis algorithms in general can focus, and be compared on finding TLDs, instead of finding all minimal cardinality diagnoses.

8 Experimental Results

This section presents an experimental evaluation of our proposed SAT-based encoding for MBD. We compare the performance of our approach to other MBD algorithms in two categories: (1) find a single minimal cardinality diagnosis, and (2) find all minimal cardinality diagnoses.

Table 1 provides basic details about our benchmark suite: ISCAS-85 [9], and 74XXX [33]. The columns indicate the system names and the number of components, inputs and outputs in each of the systems. The rightmost column indicates the preprocessing time per system (for the SAT-based approach). This includes all actions performed "once per system": decomposing the system to sections and cones, computing the bounds per section, etc.

Name	|COMP|	in	out	offline secs.
74181	65	14	8	0.02
74182	19	9	5	0.01
74283	36	9	5	0.01
c432	160	36	7	0.03
c499	202	41	32	0.08
c880	383	60	26	0.06
c1355	546	41	32	0.24
c1908	880	33	25	0.37
c2670	1193	233	140	0.29
c3540	1669	50	22	0.71
c5315	2307	178	123	1.50
c6288	2416	32	32	1.48
c7552	3512	207	108	1.73

Table 1. The Benchmark suite

8.1 A Single Minimal Cardinality Diagnosis

In the first experiment we evaluate our algorithm in finding **a single** minimal cardinality diagnoses. Table 2 presents the results for scenarios (observations) generated by Feldman *et al.* [24], and compare our SAT based diagnosis algorithm (denoted as SATbD) to the algorithms: HA* [25], CDA* [63] and SAFARI [24].

The second column indicates for each system the number of observations we experimented. The next three columns show the distribution of the obserations according to their minimal cardinality. Note that this benchmark set of scenarios is larger than the set of scenarios used in the diagnosis competition (DXC) of 2009. In addition, it includes hard scenarios with high minimal cardinality. For each algorithm, Table 2 indicates the percentage of observations solved within the prescribed timeout (succ. rate %) and the average search time (time sec.) where the average is computed over the set of observations excluding timeouts. All experiments are run on the same machine.[8]

We note that SAFARI applies a stochastic approach which does not guarantee minimal cardinality. Feldman *et al.* report [24] that even for single and double fault cardinalities, SAFARI does not always find the minimal cardinality. We run SAFARI in a configuration which instead guarantees a minimal subset diagnosis. So, SAFARI is often faster, comparing to HA* and CDA*, at the expense of minimality. For SAFARI, the tables indicates in brackets (minimal%) also the percentage of observations, excluding timeouts, where a minimal cardinality diagnosis was found. Our algorithm is presented in the last two columns (SATbD).

The results in Table 2 clearly show that SATbD clearly outperforms all the other evaluated algorithms, both in terms of success rate and in terms of average runtime. Also, we succeed to compute and verify minimal cardinality diagnosis even for scenarios with a minimal cardinality of 30. To the best of our knowledge, no algorithm before succeeded to compute minimal cardinality diagnosis for such hard scenarios.

[8] Intel Core 2 Duo (E8400 3.00GHz CPU, 4GB memory) under Linux (Ubuntu lucid, kernel 2.6.32-24-generic).

		obs. with min card. of:			HA*		CDA*		SAFARI		SATbD	
Name	#obs.	1–10	11-20	21–30	succ. rate%	time sec.	succ rate%	time sec.	succ. rate% (minimal%)	time sec.	succ. rate%	time sec.
74181	350	100%	0%	0%	68.3	3.15	46.3	4.51	100.0 (44)	0.00	100.0	0.02
74182	250	100%	0%	0%	100.0	0.00	100.0	0.01	100.0 (91)	0.00	100.0	0.01
74283	202	100%	0%	0%	100.0	0.04	100.0	1.45	100.0 (57)	0.00	100.0	0.02
c432	301	100%	0%	0%	78.1	3.63	38.2	5.15	100.0 (28)	0.03	100.0	0.03
c499	835	93%	6%	0%	24.1	5.45	10.1	1.22	100.0 (7)	0.05	100.0	0.04
c880	1182	44%	43%	12%	11.9	3.76	6.3	6.66	100.0 (48)	0.18	100.0	0.05
c1355	836	87%	13%	0%	11.4	3.90	0.0	-	100.0 (5)	0.37	100.0	0.07
c1908	846	93%	7%	0%	6.4	1.75	0.0	-	100.0 (17)	1.08	100.0	0.14
c2670	1162	66%	34%	0%	12.3	4.83	0.0	-	100.0 (14)	2.71	100.0	0.15
c3540	756	98%	1%	0%	3.7	4.30	0.0	-	100.0 (9)	5.25	100.0	0.27
c5315	2038	47%	45%	8%	2.7	11.94	0.0	-	100.0 (9)	13.34	100.0	0.42
c6288	404	100%	0%	0%	13.6	7.87	0.0	-	53.5 (25)	16.18	100.0	0.56
c7552	1557	48%	45%	5%	4.2	1.06	0.0	-	0.0	-	99.3	1.07
80 sec timeout												
c7552					7.3	20.77	0.0	0.0	99.5(13)	43.50	100.0	1.49

Table 2. Search for a Single Minimal Cardinality Diagnosis (30 sec. timeout)

Note that in the results shown in Table 2 there are 11 scenarios from the total 1,557 scenarios for system `c7552` that were not solved by the proposed algorithm within the 30 second timeout. These scenarios are, however, solved within 80 seconds each, as indicated by the last row in the table. One may observe that while our SAT based approach solves all observations given 80 seconds, the other algorithms still are not able to solve these observations, with the exception of SAFARI. Given the extended 80 seconds timeout, SAFARI is also able to solve almost all of the instances for this system, but returns minimal cardinality diagnoses in only 13% of the cases.

8.2 All Minimal Cardinality Diagnoses:

In the second experiment we evaluate our algorithm in finding **all** minimal cardinality diagnoses.

Table 3 presents the results for scenarios generated by Siddiqi [52] where minimal cardinality is bounded by 8. We compare our algorithm with HDIAG [53] and DCAS [52] for which we present results from [52] where experiments are reported for a 2.4GHz Intel Xeon X3220 with 2Gb RAM (we reported the results of our algorithm only for those systems reported by Siddiqi). Both algorithms verify minimal cardinality diagnosis. The first and third colums report the percentage of scenarios for which HDIAG and DCAS find **all** minimal cardinality diagnoses within 1800 seconds. The second and fourth columns report the average runtimes for the same sets of scenarios.

	HDAIG		DCAS		SATbD		
Name	Succ. rate%	Time Secs.	Succ rate%	Time Secs.	Succ. rate%	TLD Secs.	ALL Secs.
c432	100.0	0.21	100.0	0.31	100.0	0.07	0.09
c499	100.0	0.12	100.0	0.20	100.0	0.08	0.10
c880	99.0	0.07	99.0	0.12	100.0	0.08	0.11
c1355	99.5	0.16	99.5	0.15	100.0	0.13	0.16
c1908	90.5	368.13	76.5	82.25	100.0	0.25	0.30
c2670	90.0	176.17	100.0	3.15	100.0	0.23	0.29
c5315	0.0	-	97.5	52.34	100.0	0.58	0.67
c6288	0.0	-	27.5	305.10	50.0	104.58	105.14
c7552	0.0	-	87.5	260.93	100.0	1.01	1.12

Table 3. Find All Minimal Cardinality Diagnoses (1800 sec. timeout)

In column 5 we present the success rate (same success criteria) of SATbD running on our machine[1], in column six we report the average runtime to compute

the top level diagnoses, and in column seven, the average runtime to compute all minimal cardinality diagnoses. We observe that the difference between finding top level diagnoses and all minimal cardinality diagnoses is small (supporting the fact that it is performed in polynomial time per top-level diagnosis as described above).

SATbD clearly outperforms HDIAG and DCAS. It succeeds to compute all minimal cardinality diagnoses for all observations for all the systems except c6288 where it succeeds on 50% of the 40 observations compared to 26.5% for DCAS. Note that because of the higher success rate, the average runtimes of our SAT algorithm involve harder instances not solved by DCAS.

9 Conclusion

This part of the thesis presents a novel encoding of MBD to SAT which enables to determine, for the first time, minimal cardinality diagnoses for the entire standard ISCAS-85 benchmark. The power behind our approach comes from a combination of sophisticated system preprocessing, improved modeling, and the application of SAT compilation techniques. Experimental evaluation considers the ISCAS-85 and 74XXX benchmarks with large sets of scenarios involving observations with minimal cardinalities of up to 30. We compared our SAT-based algorithm to HA*, CDA*, SAFARI, HDIAG and DCAS. Results are unequivocal. Our algorithm outperforms the others, in finding a single minimal cardinality diagnosis as well as all minimal caridnality diagnoses often by orders of magnitude, in terms of runtime. We succeed to find and verify a minimal cardinality diagnosis for all but 11 of the scenarios in under 30 seconds per scenario, and for the remaining 11 in under 80 seconds. Further details regarding the experimental evaluation as well as a prototype implementation of our SAT-based MBD tool can be found at [43].

Summary

In this thesis we present a new approach to compile Constraint Satisfaction Problem (CSP) to Boolean satisfaction problem (SAT) and introduce a tool, BEE applying that approach.

We first introduce an approach to propagation based SAT encoding, Boolean equi-propagation, which detect Boolean equilities. But instead of consider the full CNF, our approach consider small fragments of the CNF, apply powerful reasoning technique on each fragment and propagate the learned equalities to the other fragments.

Since the original representation of the CNF is a constraint model, we maintaining the original representation and partitioning the constraint model to small fragments, usually containing single constraints, each viewed as Boolean formula.

The second important component in our approach to compile constraint models to CNF is partial evaluation. By applying alternating step of equi-propagation and partial evaluation the constraint model is simplify before encoding it to CNF.

Although we using standard encoding for constraints, choosing the encoding after applying equi-propagation and partial evaluation gives the opportunity to choose a better encoding for a constraint.

To demonstrate our approach, we introduce BEE, a tool which encode finite domain constraint models to CNF. BEE apply equi-propagation and partial evaluation to improve the generated CNF. BEE has two main advantages. On the one hand, it facilitates the process of modeling. On the other hand, given a model, it simplifies the corresponding CNF which, in many cases, is faster to solve than with other approaches.

We report on our experience in applying BEE for several cases and report that using BEE enables us to solve intresting instances not previously solved by other techniques.

In addition to the experiments reported in Part 1, in Part 2 we present a novel encoding to Model Base Diagnosis to SAT. By using sophisticated system preprocessing, improved modeling, and the application of BEE our approach is superior to all published algorithms for minimal cardinality MBD.

References

1. C. Ansótegui, A. del Val, I. Dotú, C. Fernández, and F. Manyà. Modeling choices in quasigroup completion: SAT vs. CSP. In D. L. McGuinness and G. Ferguson, editors, *AAAI*, pages 137–142, San Jose, California, USA, 2004. AAAI Press / The MIT Press.
2. R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.
3. O. Bailleux and Y. Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In F. Rossi, editor, *CP*, volume 2833 of *LNCS*, pages 108–122, Kinsale, Ireland, 2003. Springer.
4. K. Balakrishnan and V. Honavar. Intelligent diagnosis systems. *Journal of Intelligent Systems*, 8(3/4):239–290, 1998.
5. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
6. K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314, Atlantic City, NJ, USA, 1968. Thomson Book Company, Washington D.C.
7. A. Bauer. Simplifying diagnosis using LSAT: a propositional approach to reasoning from first principles. In R. Barták and M. Milano, editors, *Proceedings of the 2005 International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR)*, volume 3524 of *Lecture Notes in Computer Science*, pages 49–63, Berlin, Heidelberg, June 2005. Springer-Verlag.
8. C. Bessiere, G. Katsirelos, N. Narodytska, and T. Walsh. Circuit complexity and decompositions of global constraints. In *Proceedings of IJCAI 2009*, pages 412–418, 2009.
9. F. Brglez, D. Bryan, K. Kozminski, and M. C. of North Carolina. *Combinatorial profiles of sequential benchmark circuits*. MCNC technical report. 1989.
10. T. Bylander, D. Allemang, M. C. Tanner, and J. R. Josephson. The computational complexity of abduction. *Artif. Intell.*, 49(1-3):25–60, 1991.
11. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. *Artificial Intelligence*, 162(1-2):89–120, 2005.
12. L. Chu-Min. Equivalent literal propagation in the DLL procedure. *Discrete Applied Mathematics*, 130(2):251–276, 2003.
13. M. Codish, V. Lagoon, and P. J. Stuckey. Logic programming with satisfiability. *TPLP*, 8(1):121–128, 2008.
14. M. Codish and M. Zazon-Ivry. Pairwise cardinality networks. In *LPAR (Dakar)*, pages 154–172, 2010.
15. J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In B. Hayes-Roth and R. E. Korf, editors, *AAAI*, volume 2, pages 1092–1097, Seattle, WA, USA, 1994. AAAI Press / The MIT Press.
16. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR*, pages 148–159, 1996.
17. A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.
18. J. de Kleer. An improved approach for generating max-fault min-cardinality diagnoses. In *19th International Workshop on Principles of Diagnosis (DX-08)*, Australia, 2008.

19. J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
20. N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.
21. N. Eén and N. Sörensson. An extensible sat-solver. In *SAT2003*, pages 502–518, 2003.
22. N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2(1-4):1–26, 2006.
23. A. Feldman, G. Provan, J. de Kleer, S. Robert, and A. van Gemund. Solving model-based diagnosis problems with max-sat solvers and vice versa. `https://www.phmsociety.org/events/workshop/dx/10/proceedings`, October 2010.
24. A. Feldman, G. Provan, and A. van Gemund. Approximate model-based diagnosis using greedy stochastic search. *J. Artif. Int. Res.*, 38:371–413, May 2010.
25. A. Feldman and A. J. C. van Gemund. A two-step hierarchical algorithm for model-based diagnosis. In *AAAI*, 2006.
26. A. M. Frisch, C. Jefferson, and I. Miguel. Symmetry breaking as a prelude to implied constraints: A constraint modelling pattern. In *Proc. 16th Euro. Conf. on AI, 171175*, pages 171–175. Press, 2004.
27. A. G. Frutos, Q. Liu, A. J. Thiel, A. M. W. Sanner, A. E. Condon, L. M. Smith, and R. M. Corn. Demonstration of a word design strategy for DNA computing on surfaces. *Journal of Nucleic Acids Research*, 25(23):4748–4757, 1997.
28. J. A. Gallian. A dynamic survey of graph labeling. *The Electronic Journal of Combinatorics*, 18, 2011.
29. M. Gavanelli. The log-support encoding of csp into sat. In C. Bessiere, editor, *CP*, volume 4741 of *LNCS*, pages 815–822, Providence, RI, USA, 2007. Springer.
30. A. V. Gelder. Toward leaner binary-clause reasoning in a satisfiability solver. *Ann. Math. Artif. Intell.*, 43(1):239–253, 2005.
31. I. P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *ECAI*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 98–102. IOS Press, 2006.
32. C. P. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In G. Smolka, editor, *CP*, volume 1330 of *LNCS*, pages 121–135. Springer, 1997.
33. M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the iscas-85 benchmarks: A case study in reverse engineering. *IEEE Des. Test*, 16:72–80, July 1999.
34. M. Heule, M. Järvisalo, and A. Biere. Efficient cnf simplification based on binary implication graphs. In K. A. Sakallah and L. Simon, editors, *SAT*, volume 6695 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2011.
35. J. Huang. Universal booleanization of constraint models. In *CP2008*, pages 144–158, 2008.
36. G. Jäger. An effective sat encoding for magic labeling. In U. Faigle, R. Schrader, and D. Herrmann, editors, *CTW*, pages 97–100, 2010.
37. T. Kirkland and M. R. Mercer. A topological search algorithm for atpg. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, DAC '87, pages 502–508, New York, NY, USA, 1987. ACM.
38. C.-M. Li. Equivalent literal propagation in the dll procedure. *Discrete Appl. Math.*, 130(2):251–276, Aug. 2003.

39. J. MacDougall, M. Miller, M. Slamin, and W. Wallis. Vertex-magic total labelings of graphs. *Utilitas Mathematica*, 61:3–21, 2002.
40. T. Mancini, D. Micaletto, F. Patrizi, and M. Cadoli. Evaluating ASP and commercial solvers on the CSPLib. *Constraints*, 13(4):407–436, 2008.
41. N. Manthey. Coprocessor 2.0 - a flexible cnf simplifier - (tool presentation). In A. Cimatti and R. Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 436–441. Springer, 2012.
42. A. Metodi. BEE. `http://amit.metodi.me/research/bee/`, 2012.
43. A. Metodi. Sat-based mbd tool. `http://amit.metodi.me/research/mbdsolver/`, 2012.
44. A. Metodi. SCryptominisat. `http://amit.metodi.me/research/scrypto/`, 2012.
45. A. Metodi and M. Codish. Compiling finite domain constraints to sat with `bee`. *Theory and Practice of Logic Programming*, 2012.
46. A. Metodi, M. Codish, V. Lagoon, and P. J. Stuckey. Boolean equi-propagation for optimized SAT encoding. In J. H.-M. Lee, editor, *CP*, volume 6876 of *LNCS*, pages 621–636. Springer, 2011.
47. A. Metodi, R. Stern, M. Kalech, and M. Codish. Compiling model-based diagnosis to boolean satisfaction.
48. J. Murray, G. Hughes, and K. Kreutz-Delgado. Machine learning methods for predicting failures in hard drives: A multiple-instance application. *Journal of Machine Learning Research*, 6(1):783, 2006.
49. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *CP2007*, pages 529–543, 2007.
50. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–96, 1987.
51. B. Selman and H. J. Levesque. Abductive and default reasoning: A computational core. In *AAAI*, pages 343–348, 1990.
52. S. A. Siddiqi. Computing minimum-cardinality diagnoses by model relaxation. In *IJCAI*, pages 1087–1092, 2011.
53. S. A. Siddiqi and J. Huang. Hierarchical diagnosis of multiple faults. In *IJCAI*, pages 581–586, 2007.
54. A. Smith, A. Veneris, M. F. Ali, and A. Viglas. Fault diagnosis and logic debugging using Boolean satisfiability. *IEEE TRANS. ON CAD*, 24:1606–1621, 2005.
55. F. Somenzi. CUDD: Colorado University Decision Diagram package. (Online, accessed 13 April 2011), Feb. 2009. `http://vlsi.colorado.edu/~fabio/CUDD/`.
56. M. Soos. CryptoMiniSAT, v2.5.1. `http://www.msoos.org/cryptominisat2`, 2010.
57. B. Stein, O. Niggemann, and T. Lettmann. Speeding up model-based diagnosis by a heuristic approach to solving sat. In *Proceedings of the 24th IASTED international conference on Artificial intelligence and applications*, pages 273–278, Anaheim, CA, USA, 2006. ACTA Press.
58. N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
59. R. Tarjan. Efficiency of a good but not linear set union algorithm. *JACM*, 22(2):215–225, 1975.
60. P. Torasso and G. Torta. Model-based diagnosis through obdd compilation: A complexity analysis. In *Reasoning, Action and Interaction in AI Theories and Systems*, pages 287–305, 2006.
61. P. van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999.

62. W. J. van Hoeve. The alldifferent constraint: A survey, 2001.
63. B. C. Williams and R. J. Ragno. Conflict-directed A* and its role in model-based embedded systems. *Discrete Appl. Math.*, 155(12):1562–1595, 2007.