# Practical Applications of Boolean Satisfiability

Joao Marques-Silva

***Abstract*— Boolean Satisfiability (SAT) solvers have been the subject of remarkable improvements since the mid 90s. One of the main reasons for these improvements has been the wide range of practical applications of SAT. Indeed, examples of modern applications of SAT range from termination analysis in term-rewrite systems to circuit-level prediction of crosstalk noise. The success of SAT solvers motivated many practical applications, but many practical applications have also provided the examples and the challenges that allowed the development of more efficient SAT solvers. This paper provides an overview of some of the most well-known applications of SAT and outlines several other successful applications of SAT. Moreover, the improvements in SAT solvers motivated the development of new algorithms for strategic extensions of SAT. As a result, the paper also provides a brief survey of recent work on extensions of SAT, including pseudo-Boolean constraints, maximum satisfiability, model counting and quantified Boolean formulas.**

## I. Introduction

Boolean Satisfiability (SAT) is a well-known decision problem, that consists in deciding whether a propositional logic formula can be satisfied given suitable value assignments to the variables of the formula. SAT is a widely used modeling framework for solving combinatorial problems. SAT is also a well-known NP-complete decision problem [15]. As a result, unless $P = NP$, all SAT algorithms require worst-case exponential time. However, modern SAT algorithms are extremely effective at coping with large search spaces, by exploiting the problem's structure when it exists [38], [44], [18] (also, see [12]). The performance improvements made to SAT solvers since the mid 90s motivated their application to a wide range of practical applications, from crosstalk noise prediction in integrated circuits [10] to termination analysis in term-rewrite systems [21]. In some applications, the use of SAT provides remarkable performance improvements. Examples include model-checking of finite-state systems [8], [50], [43], design debugging [52], AI planning [49], [46], and haplotype inference in bioinformatics [34]. Additional successful examples of practical applications of SAT include knowledge-compilation [16], software model checking [25], [14], software testing [26], package management in software distributions [55], checking of pedigree consistency [35], test-pattern generation in digital systems [30], design debugging and diagnosis [52], identification of functional dependencies in Boolean functions [31], technology-mapping in logic synthesis [47], circuit delay computation [41], as well as the ones mentioned above [21], [10]. However, this list is incomplete as the number of applications of SAT has been on the rise in recent years (e.g. [55], [35], [31]).

This work has been partially supported by EU projects IST/033709 and ICT/217069 and by EPSRC grant EP/E012973/1.

J. Marques-Silva is with the School of Electronics and Computer Science, University of Southampton, SO17 1BJ, United Kingdom. Email: `jpms@ecs.soton.ac.uk`

Besides practical applications, SAT has also influenced a number of related decision and optimization problems, which will be referred to as extensions of SAT. Most extensions of SAT either use the same algorithmic techniques as used in SAT, or use SAT as a core engine. One of the most promising extensions of SAT is Satisfiability Modulo Theories (SMT) [4], [22] (also, see [11]). Other applications of SAT include pseudo-Boolean (PB) constraints [36], [19], maximum satisfiability (MaxSAT) [33], [24], model counting (#SAT) [48], and Quantified-Boolean Formulas (QBF) [32].

This paper provides an overview of some of the most successful practical applications of SAT, and summarizes some other well-known applications. The paper also briefly surveys the use of SAT in some of its best known extensions. The paper is organized as follows. Section II introduces the notation used in the remainder of the paper. Afterwards, Section III illustrates practical applications of SAT, by focusing on a number of concrete case studies. Section IV outlines research work in representative extensions of SAT. Finally, the paper concludes in Section V.

## II. Definitions

### A. Propositional Formulas and Satisfiability

Propositional formulas[1] are defined over a finite set of Boolean variables $X$. Individual variables can be represented by letters $x$, $y$, $z$, $w$ and $o$, and subscripts may be used (e.g. $x_1$). The propositional connectives considered will be $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. Parenthesis will be used to enforce precedence. Most SAT algorithms require propositional formulas to be represented in Conjunctive Normal Form (CNF). A CNF formula $\varphi$ consists of a conjunction of clauses $\omega$, each of which consists of a disjunction of literals. A literal is either a variable $x_i$ or its complement $\neg x_i$. A CNF formula can also be viewed as a set of clauses, and each clause can be viewed as a set of literals. Throughout this paper the representation used will be clear from the context. The conversion from arbitrary propositional formulas to CNF formulas is addressed in the next section.

In the context of search algorithms for SAT, variables can be *assigned* a logic value, either 0 or 1. Alternatively, variables may also be *unassigned*. Assignments to the problem variables can be defined as a function $\nu : X \rightarrow \{0, u, 1\}$,

[1]A more comprehensive description of Boolean Satisfiability and basic algorithms is provided elsewhere [27].

where $u$ denotes an *undefined* value used when a variable has not been assigned a value in $\{0, 1\}$. Given an assignment $\nu$, if all variables are assigned a value in $\{0, 1\}$, then $\nu$ is referred to as a *complete assignment*. Otherwise it is a *partial assignment*.

Assignments serve for computing the values of literals, clauses and the complete CNF formula, respectively, $l^\nu$, $\omega^\nu$ and $\varphi^\nu$. A total order is defined on the possible value assignment, $0 < u < 1$. Moreover, $1 - u = u$. As a result, the following definitions apply:

$$l^\nu = \begin{cases} \nu(x_i) & \text{if } l = x_i \\ 1 - \nu(x_i) & \text{if } l = \neg x_i \end{cases} \quad (1)$$

$$\omega^\nu = \max \{l^\nu \mid l \in \omega\} \quad (2)$$

$$\varphi^\nu = \min \{\omega^\nu \mid \omega \in \varphi\} \quad (3)$$

The assignment function $\nu$ will also be viewed as a set of tuples $(x_i, v_i)$, with $v_i \in \{0, 1\}$. Adding a tuple $(x_i, v_i)$ to $\nu$ corresponds to assigning $v_i$ to $x_i$, such that $\nu(x_i) = v_i$. Removing a tuple $(x_i, v_i)$ from $\nu$ corresponds to assigning $u$ to $x_i$.

Given an assignment, clauses and CNF formulas can be characterized as *unsatisfied*, *satisfied*, or *unresolved*. A clause is unsatisfied if all its literals are assigned value 0. A clause is satisfied if at least one of its literals is assigned value 1. A clause is unresolved if it is neither unsatisfied nor satisfied. A CNF formula $\varphi$ is satisfied iff *all* of its clauses are satisfied, and it is unsatisfied iff at least one of its clauses is unsatisfied. Otherwise it is unresolved. The SAT problem for a CNF formula $\varphi$ consists in deciding whether there exists an assignment to the problem variables, such that $\varphi$ is satisfied, or proving that no such assignment exists. As mentioned earlier, the satisfiability problem for general propositional formulas is NP-complete [15] and so is the satisfiability problem for CNF formulas.

## B. Boolean Circuits

Many practical applications are often represented in some intermediate representation, from which a CNF formula is then generated. One of the most often used intermediate representations are combinational Boolean circuits [49], [8], [25], [55]. Combinational Boolean circuits are composed of gates and connections between gates. In this paper, only simple gates are considered and restricted to basic operations: NOT (negation), AND (conjunction), OR (disjunction), XOR (negated equivalence), or alternatively $^-$, $\cdot$, $+$, $\oplus$. Observe that $\text{XOR}(x, y) = \text{OR}(\text{AND}(x, \text{NOT}(y)), \text{AND}(\text{NOT}(x), y))$, or alternatively $x \oplus y = x \cdot \bar{y} + \bar{x} \cdot y$. Moreover, for simplicity, two-input single-output gates are assumed. The notation $y = \text{OP}(x_1, x_2)$ denotes a gate with output $y$ and inputs $x_1$ and $x_2$, and OP is one of the basic operations.

Converting Boolean circuits to CNF is straightforward, and follows the procedure first outlined by G. Tseitin [54]. Consider a gate $y = \text{OP}(x_1, x_2)$. The CNF representation captures the valid assignments between the gate inputs and outputs, Hence, $\varphi(y, x_1, x_2) = 1$ iff the predicate $y = \text{OP}(x_1, x_2)$ holds true. The CNF representations for simple gates is shown in Table I (observe that XOR gates can be replaced by NOT, AND and OR as described above). For generality, the number of inputs considered for AND and OR gates is unrestricted. Even though Tseitin's transformation is arguably the most often used, there are a number of effective alternatives including Plaisted and Greenbaum's [45].

TABLE I
CNF REPRESENTATION OF SIMPLE GATES

| Gate | CNF Representation |
|---|---|
| $y = \text{NOT}(x_1)$ | $(\neg y \vee \neg x_1) \wedge (y \vee x_1)$ |
| $y = \text{AND}(x_1, \ldots, x_k)$ | $(y \vee \neg x_1 \vee \ldots \vee \neg x_k) \wedge \bigwedge_{i=1}^k (x_i \vee \neg y)$ |
| $y = \text{OR}(x_1, \ldots, x_k)$ | $(\neg y \vee x_1 \vee \ldots \vee x_k) \wedge \bigwedge_{i=1}^k (\neg x_i \vee y)$ |

Another often used technique is to exploit the sharing of common structure in Boolean circuits. Examples of representations that exploit structural sharing are *Reduced Boolean Circuits* (RBC) [1] and *Boolean Expression Diagrams* (BED) [3].

Observe that is is straightforward to represent arbitrary propositional formulas as Boolean circuits. First, note that $\neg$, $\wedge$ and $\vee$ represents a sufficient set of connectives. Second, associate a new Boolean variable with each level of parenthesis in the propositional formula. As a result, it is straightforward to represent arbitrary propositional formulas in CNF.

## C. Linear Inequalities

Linear inequalities over Boolean variables are a widely used modeling technique. For example, with the objective of modeling an integer variable $r$ that can take one out of $k$ values, i.e. $1 \le r \le k$, one often used approach is to create $k$ Boolean variables $x_1, \ldots, x_k$, such that $x_i = 1$, $1 \le i \le k$, iff $r = i$. In addition, since $r$ must take one of its possible values, then one of the $x_i$ variables must be assigned value 1. Hence,

$$\sum_{i=1}^k x_i = 1 \quad (4)$$

which can be represented as:

$$(\sum_{i=1}^k x_i \le 1) \wedge (\sum_{i=1}^k x_i \ge 1) \quad (5)$$

The previous example illustrates special cases of linear inequalities, referred to as *cardinality constraints*, the general form being of the form $\sum x_i \le k$. More general constraints are often necessary, and so it is necessary to develop solutions for encoding linear inequalities of the form:

$$\sum_{i=1}^k a_i \, x_i \le b \quad (6)$$

The encoding proposed by J. Warners [57] ensures that linear inequalities can be encoded into CNF in linear time and space, and uses adders as the basic operator. Despite being optimal in terms of space required, Warners' encoding does not guarantee *arc-consistency*, i.e. the ability of implying all necessary assignments given a partial assignment. Other encodings exist [19], [5], the most effective of which being based on Binary Decision Diagrams (BDDs) and sorting networks. For arbitrary linear inequalities, BDDs guarantee arc-consistency but can require exponential space in the worst-case. Sorting networks require polynomial space but do not guarantee arc-consistency.

For cardinality constraints, a number of polynomial encodings ensure arc-consistency, including BDDs, sorting networks [19], and sequential counters [51]. Given its widespread use, the encoding for $\sum x_i \leq 1$ using sequential counters is given below:

$$\begin{array}{l}(\neg x_1 \vee s_1) \wedge (\neg x_k \vee \neg s_{k-1}) \wedge \\ \bigwedge_{1<i<k} \left((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1})\right)\end{array} \tag{7}$$

where $s_i$ are additional auxiliary Boolean variables. Inspection of the formula allows concluding that at most one $x_i$ can be assigned value 1, for which $s_{i-1}$, with $i > 1$, is assigned value 0 and $s_i$ is assigned value 1. For all $x_i$, with $i > 1$, for which $s_{i-1} = s_i$, then $x_i$ must be assigned value 0. Moreover, observe that encoding $\sum x_i \geq 1$ is immediate with a single clause and, given (7), so is the encoding of $\sum x_i = 1$.

Finally, more general constraints can be encoded into CNF (e.g. [56]), albeit this is seldom used in practical settings.

## III. SAT Applications

This section overviews the application of SAT in a number of areas, namely combinational equivalence checking [9], automatic test-pattern generation [30], model checking [8], planning [49] and haplotype inference [34]. The applications are organized by increasing complexity of the associated SAT representation.

### A. Combinational Equivalence Checking

An essential circuit design task is to check the functional equivalence of two circuits. The simplest form of equivalence checking addresses combinational circuits. Let $C_A$ and $C_B$ denote two combinational circuits, both with inputs $x_1, \ldots, x_n$ and both with $m$ outputs, $C_A$ with outputs $y_1, \ldots, y_m$ and $C_B$ with outputs $w_1, \ldots, w_m$. The function implemented by each of the two circuits is defined as follows: $\mathbf{f_A} : \{0,1\}^n \rightarrow \{0,1\}^m$, and $\mathbf{f_B} : \{0,1\}^n \rightarrow \{0,1\}^m$. Let $\mathbf{x} \in \{0,1\}^n$ and define $\mathbf{f_A}(\mathbf{x}) = (f_{A,1}(\mathbf{x}), \ldots, f_{A,m}(\mathbf{x}))$ and $\mathbf{f_B}(\mathbf{x}) = (f_{B,1}(\mathbf{x}), \ldots, f_{B,m}(\mathbf{x}))$. The two circuits are *not* equivalent if the following condition holds:

$$\exists_{\mathbf{x} \in \{0,1\}^n} \exists_{1 \leq i \leq m} f_{A,i}(\mathbf{x}) \neq f_{B,i}(\mathbf{x}) \tag{8}$$

which can be represented as the following satisfiability problem:

$$\bigvee_{i=1}^{n} (f_{A,i}(\mathbf{x}) \oplus f_{B,i}(\mathbf{x})) = 1 \tag{9}$$

The resulting satisfiability problem is illustrated in Figure 1, and is referred to as a *miter* [9]. From the results of the previous section it is straightforward to encode the combinational equivalence checking problem in CNF.

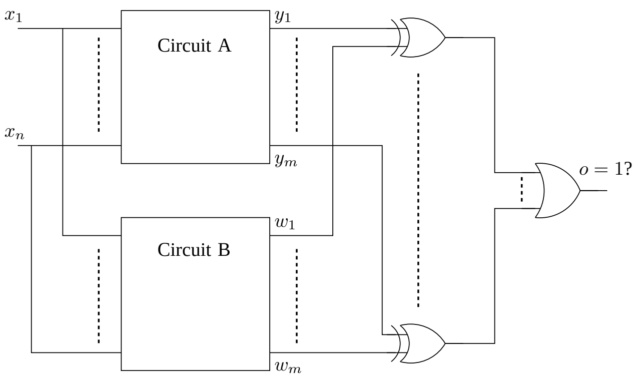

Fig. 1. Equivalence Checking Miter

Somewhat surprisingly, combinational equivalence checking can be challenging for SAT solvers. Hence, a number of techniques, including miter preprocessing and solving intermediate equivalence checking problems, are often used [37], [28].

### B. Automatic Test-Pattern Generation

Fabricated integrated circuits may be subject to defects, which may cause circuit failure. The most widely used approach for identifying fabrication defects is automatic test-pattern generation (ATPG) [2]. Moreover, the most often used model for representing fabrication defects is the single stuck-at fault model (SSF) [2], where a single connection in the circuit is assumed to be stuck at a given logic value, either 0 or 1, denoted respectively by stuck-at 0 (or sa-0) and stuck-at 1 (or sa-1). ATPG consists in computing input assignments that allow demonstrating the existence or absence of each target fault, or proving that no assignment exists (hence it is essentially a modified satisfiability problem). When such an assignment exists, it is said that it *detects* the target fault. In what follows combinational circuits are assumed, but the same ideas can be extended to sequential circuits [2].

In order to compute an input assignment to detect a given target fault $x$ sa-$v$, two copies of the circuit are considered. The first copy represents the circuit without the fault, as is referred to as the *good* circuit. The second copy represents the circuit with the fault, and is referred to as the *faulty* circuit. Using the notation of the previous section, a Boolean function is associated with each copy of the circuit: the good circuit is described by $\mathbf{f_G} : \{0,1\}^n \rightarrow \{0,1\}^m$, and the faulty circuit is described by $\mathbf{f_F} : \{0,1\}^n \rightarrow \{0,1\}^m$. As a result, the fault will be detected iff for some input assignment, the outputs of the two circuits differ:

$$\exists_{\mathbf{x} \in \{0,1\}^n} \exists_{1 \leq i \leq m} f_{G,i}(\mathbf{x}) \neq f_{F,i}(\mathbf{x}) \tag{10}$$

As before, this condition can be represented as the following satisfiability problem:

$$\bigvee_{i=1}^{n} (f_{G,i}(\mathbf{x}) \oplus f_{F,i}(\mathbf{x})) = 1 \tag{11}$$

Observe that the miter can also be used for representing the problem of ATPG, where $A$ represents the good circuit and

$B$ represents the faulty circuit. Even though equation (11) can be encoded directly into CNF and solved with a SAT solver, this is in general not effective. As a result, the model is modified to provide additional structural information [30], [53], [39]. The faulty circuit is only partially represented, involving only the nodes whose value can differ from the good circuit. For each such node $x$, an additional variable $x_S$ is used to denote whether the values in the two circuits differ. $x_S$ is referred to as the sensitization variable of node $x$ and takes value 1 if the value of $x$ in the two circuits differs. If $x_G$ is the value in the good circuit and $x_F$ is the value in the faulty circuit, then $x_S$ is defined as follows:

$$x_S \leftrightarrow (x_G \oplus x_F) \tag{12}$$

The use of SAT in ATPG was first proposed by T. Larrabee [30]. Improvements based on preprocessing were described in [53]. Additional improvements were further proposed in [39], including the reuse of learnt clauses in between target faults and the encoding of conditions for unique sensitization points [2].

## C. Model Checking

Given a set of propositional symbols $\Sigma$, a Kripke structure is defined as a 4-tuple $\mathcal{M} = (S, I, T, L)$, where $S$ is a finite set of states, $I \subseteq S$ is a set of initial states, $T \subseteq S \times S$ is a transition relation, and $L : S \rightarrow \mathcal{P}(\Sigma)$ is a labelling function, where $\mathcal{P}(\Sigma)$ denotes the powerset over the set of propositional symbols. Temporal logics allow describing properties of systems. Two propositional temporal logics are widely used: Linear-Time Logic (LTL) and Computation-Tree Logic (CTL) [13]. In this paper temporal properties are described in LTL, but CTL could also have been considered. Model checking algorithms can be characterized as explicit-state or implicit-state (or symbolic) [13]. Explicit state model checking algorithms represent explicitly the states of the transition relation, whereas symbolic model checking algorithms do not. Initial symbolic model checking algorithms were based on Binary Decision Diagrams (BDDs) [42]. Over the last decade, a number of alternatives based on Boolean Satisfiability (SAT) have been proposed [8], [50], [43].

Most work on SAT-based model checking assumes *safety* properties $\mathsf{G}\,\psi_S$ [2], where $\psi_S$ is a purely propositional formula. The interpretation is that $\psi_S$ must hold on *all* reachable states of $\mathcal{M}$. For simplicity, the Kripke structure $\mathcal{M} = (S, I, T, L)$ will be represented by the 3-tuple $M = (I, T, F)$, where $I$ is a predicate representing the initial states, $T$ is a predicate representing the transition relation, and $F$ is a predicate representing the failing property (i.e. $F = \neg\psi_S$), defined on state variables (denoted as set $Y$). Moreover, the predicates $I$, $T$ or $F$ assume the underlying Kripke structure $\mathcal{M} = (S, I, T, L)$ and associated target formula $\psi_S$. Observe that the states are not explicitly represented. A set of variables $Y$ encodes the possible states, and predicate $T$ encodes whether the system can go from state (represented with variables) $Y_i$ to state $Y_{i+1}$.

[2]A detailed account of LTL temporal operators is given in [13].

**Algorithm 1** Organization of BMC

```
BMC($M = (I, T, F)$, $\mu$)
1  $k \leftarrow 0$
2  while $k \leq \mu$
3      do $\varphi \leftarrow$ CNF(BMC($M, k$), $W$)
4         if SAT($\varphi$)
5            then return false ▷ Found counterexample
6         $k \leftarrow k + 1$
7  return true
```

The use of SAT for model checking purposes entails iteratively unfolding the transition relation, and is referred to as bounded model checking (BMC) (where bounded indicates that a fixed unfolding is considered). Given a safety property $\mathsf{G}\,\psi_S$, the solution to address this problem with SAT is to consider the complement $\mathsf{F}\,\neg\psi_S$ of the safety property, representing the condition that $\psi_S$ will not hold in some reachable state. The condition $\neg\psi_S$ will be referred to as the failing property, and represented with a predicate $F$. Bounded model checking consists of iteratively unfolding the transition relation, while checking whether the failing property holds. The generic Boolean formula associated with SAT-based BMC is the following [8]:

$$I(Y_0) \wedge \bigwedge_{0 \leq i < k} T(Y_i, Y_{i+1}) \wedge \left( \bigvee_{0 \leq i \leq k} F(Y_i) \right) \tag{13}$$

Equation (13) is referred to as $\textsc{Bmc}(M, k)$, and represents the unfolding of the transition relation for $k$ time steps, where $I(Y_0)$ represents the initial state (at time step 0), $T(Y_i, Y_{i+1})$ represents the transition relation between states at time steps $i$ and $i+1$, respectively $Y_i$ and $Y_{i+1}$, and $F(Y_i)$ represents the failing property at time step $i$. Given the proposition formula $\textsc{Bmc}(M, k)$, it is straightforward to generate a CNF formula $\varphi$ as described earlier in the paper. The resulting CNF formula can then be evaluated by a SAT solver.

The typical organization of BMC for safety properties is illustrated in Algorithm 1. The details regarding the sets of variables associated with each propositional formula are omitted, but are clear from the context. Moreover, the encoding of the BMC formula to CNF is shown as function $\text{CNF}()$, and uses a set of auxiliary Boolean variables $W$. Finally, $\mu$ represents an upper bound on the unfolding of the transition relation. Experimental evidence has confirmed SAT-based BMC to be an extremely competitive technique, that has been used in industrial settings [7].

A key difficulty with BMC is its inability for proving that there is no counterexample for a given safety property $\mathsf{G}\ \psi_S$. Unless the recurrence (or the reachability) diameter [7] of an automaton is known, it is not possible to pre-compute the value of the upper bound ($\mu$) used in Algorithm 1. In general the recurrence diameter of an automaton is not known, and so BMC is incomplete. Hence, if the BMC algorithm returns true it does not imply that a counterexample cannot be identified. In recent years different approaches have been proposed for ensuring the completeness of SAT-based model checking. Well-known examples include the use of induction [50] and interpolants [43].

## D. Planning in Artificial Intelligence

AI planning was one of the first successful practical applications of SAT [49]. The SAT formulation of planning actually motivated the work on bounded model checking, described in the previous section. As a result, the description of planning as satisfiability is formulated similarly to bounded model checking [46].

A deterministic transition system is a 4-tuple $(S, I, T, G)$, where $S$ is a set of states, $I \in S$ is an initial state, $T \subseteq S \times S$ is a set of operators (describing changes of states), and $G \subseteq S$ is a set of goal states. As before the states are encoded with a set of states variables $Y$.

Given a number of states $k$, denoting the length of the plan, SAT-based planning consists in deciding whether one of the goal states can be reached in $k$ time steps. This condition can be represented as follows:

$$\psi_k = I(Y_0) \wedge \bigwedge_{0 \leq i < k} T(Y_i, T_{i+1}) \wedge G(Y_k) \tag{14}$$

SAT-based planning considers possible plans of increasing length, starting from 0. The length is increased while the resulting instance of SAT is unsatisfiable. Observe that, as for most of the other applications, SAT algorithms must be able to prove unsatisfiability.

## E. Haplotyping in Bioinformatics

Haplotypes encode the genetic constitution of an individual chromosome. The genetic constitution of a chromosome is described by a DNA sequence. A DNA sequence is specified by four nucleotides, which can be distinguished by the bases they contain: A (adenine), C (cytosine), T (thymine), and G (guanine). In practice, haplotypes encode Single Nucleotide Polymorphism (SNPs) in DNA, where SNPs denote genetic mutations. An SNP variation occurs when a single nucleotide replaces one of the other three nucleotides.

The identification of haplotypes may be useful for identifying specific diseases, as well as to predict patients response to drugs. However, existing technology is unable to reveal the actual haplotypes, providing instead genotypes. The problem of haplotype inference consists in deriving haplotype data from genotype data. More concretely, given a set $\mathcal{G}$ of $n$ genotypes, each of length $m$, haplotype inference consists in finding a set $\mathcal{H}$ of $2 \cdot n$ haplotypes, not necessarily different, such that for each genotype $g_i \in G$ there is at least one pair of haplotypes $(h_k, h_l)$, with $h_k, h_l \in \mathcal{H}$, that *explains* $g_i$. The variable $n$ denotes the number of individuals in the sample, $m$ denotes the number of SNP sites, and $g_i$ denotes a specific genotype where $1 \leq i \leq n$. (Furthermore $g_{ij}$ denotes a specific site $j$ in genotype $g_i$ where $1 \leq j \leq m$.)

Without loss of generality, the two possible values of each SNP are assumed to be either 0 or 1. Value 0 represents the wild type and value 1 represents the mutant type. A haplotype is then a string over the alphabet {0,1}. Moreover, genotypes may be represented by extending the alphabet used to represent haplotypes to {0,1,2}. Homozygous sites are represented by values 0 or 1 depending, respectively, on whether both haplotypes have value 0 or 1 at that site. Heterozygous sites are represented by value 2. Two haplotypes $(h_k, h_l)$ explain a given genotype $g_i$ iff for each site $j$, if site $j$ is homozygous, then $h_{kj} = h_{lj} = g_{ij}$, and if site $j$ is heterozygous, then $h_{kj} \neq h_{lj}$.

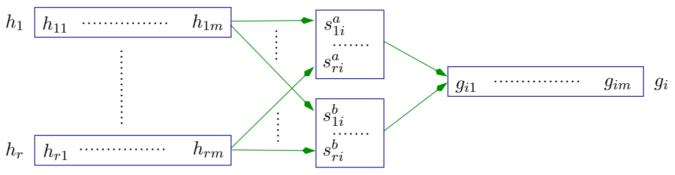

Fig. 2. Boolean variables used in SHIPs

One approach for haplotype inference is based on pure parsimony [23]. Given a set of genotypes, the haplotype inference by pure parsimony (HIPP) finds a solution to the haplotype inference problem such that the total number of distinct haplotypes used is minimum. The HIPP problem is NP-hard (e.g. see [29]).

One possible approach for solving the HIPP problem is to use SAT, one concrete example being SHIPs [34]. The SAT-based HIPP solution algorithm starts from a lower bound $lb$ on the number of haplotypes necessary to explain the set of genotypes; a trivial value for $lb$ is 1. The algorithm searches for the smallest least value $r$ such that there exists a set $\mathcal{H}$ of haplotypes with $r = |\mathcal{H}|$, which explain all genotypes in $\mathcal{G}$. Observe that the value of $r$ is guaranteed to satisfy $lb \leq r \leq 2\,n$, since a solution with $2\,n$ haplotypes is guaranteed to exist. For each value of $r$ considered, a CNF formula $\varphi^r$ is created, and a SAT solver is invoked.

In what follows we assume $n$ genotypes each with $m$ sites. The same indexes will be used throughout: $i$ ranges over the genotypes and $j$ over the sites, with $1 \leq i \leq n$ and $1 \leq j \leq m$. In addition, $r$ candidate haplotypes are considered, each with $m$ sites, and with $1 \leq r \leq 2\,n$. An additional index $k$ is associated with haplotypes, such that $1 \leq k \leq r$. As a result, $h_{kj} \in \{0, 1\}$ denotes the $j^{th}$ site of haplotype $k$.

For a given value of $r$, the SHIPs model considers $r$ haplotypes and seeks to associate two haplotypes (possibly corresponding to the same haplotype) with each genotype $g_i$, where $1 \leq i \leq n$. The Boolean variables used by SHIPs are depicted in Figure 2. For each genotype $g_i$ the model uses *selector* variables for selecting which haplotypes are used for explaining $g_i$. Since the genotype is to be explained by *two* haplotypes, the model uses two sets, $a$ and $b$, of $r$ selector variables, respectively $s^a_{ki}$ and $s^b_{ki}$ with $k = 1, \ldots, r$. Hence, genotype $g_i$ is explained by haplotypes $h_{k_1}$ and $h_{k_2}$ if $s^a_{k_1 i} = 1$ and $s^b_{k_2 i} = 1$. Clearly, $g_i$ is also explained by the same haplotypes if $s^a_{k_2 i} = 1$ and $s^b_{k_1 i} = 1$.

We can now derive the conditions for the SHIPs model:

- If a site $g_{ij}$ is 0 (resp. 1), and if haplotype $k$ is selected for explaining genotype $i$, either by the $a$ or the $b$

representative, then the value of haplotype $k$ at site $j$ *must* be 0 (resp. 1). In CNF, if site $g_{ij}$ is 0, the model includes $(\neg s^a_{ki} \vee \neg h_{kj}) \wedge (\neg s^b_{ki} \vee \neg h_{kj})$, and if site $g_{ij}$ is 1, then the model includes $(\neg s^a_{ki} \vee h_{kj}) \wedge (\neg s^b_{ki} \vee h_{kj})$, in both cases for $k = 1, \ldots, r$.

- Otherwise, one requires that the haplotypes explaining the genotype $g_i$ have opposing values at site $i$. This is done by creating a variable $t_{ij} \in \{0, 1\}$ such that site $j$ of the haplotype selected by the $a$ representative selector assumes the same value as $t_{ij}$, and site $j$ of the haplotype selected by the $b$ representative selector assumes the complementary value of $t_{ij}$. As a result the model requires $(h_{kj} \vee \neg t_{ij} \vee \neg s^a_{ki}) \wedge (\neg h_{kj} \vee t_{ij} \vee \neg s^a_{ki}) \wedge (h_{kj} \vee t_{ij} \vee \neg s^b_{ki}) \wedge (\neg h_{kj} \vee \neg t_{ij} \vee \neg s^b_{ki})$ for $k = 1, \ldots, r$. Observe that $h_{kj}$ equals $t_{ij}$ if $s^a_{ki} = 1$, and $h_{kj}$ equals $\neg t_{ij}$ if $s^b_{ki} = 1$.

Clearly, for each genotype $g_i$, and for $a$ or $b$, it is necessary that exactly one haplotype is used, and so exactly one selector variable can be assigned value 1. This can be captured with the following cardinality constraints:

$$\left( \sum_{k=1}^{r} s^a_{ki} = 1 \right) \wedge \left( \sum_{k=1}^{r} s^b_{ki} = 1 \right) \tag{15}$$

As shown in Section II, these cardinality constraints can be encoded in CNF in linear space, by introducing additional auxiliary variables. Besides the basic model outlined above, SAT-based haplotyping requires the inclusion of a number of effective techniques, including lower bounds and identification of symmetries [34].

## IV. Extensions of SAT

A number of extensions of SAT allow greater modeling flexibility than plain SAT. Purely Boolean examples include Quantified Boolean Formulas (QBF), Pseudo-Boolean (PB) solving and optimization, and Maximum Satisfiability (MaxSAT) and variants. The most effective algorithmic techniques used in SAT have also been applied in most extensions of SAT, thus enabling significant practical applications. This section briefly surveys these extensions of SAT.

Pseudo-Boolean (PB) constraints generalize SAT by considering linear inequalities over Boolean variables instead of clauses. Moreover, a linear cost function can be considered [6]. The problem of optimizing PB-constraints is NP-hard. Moreover, as in the case of SAT, a number of effective algorithms have been proposed [36], [19] that integrate and extend the most effective SAT techniques.

The maximum satisfiability (MaxSAT) problem can be stated as follows. Given an instance of SAT represented in Conjunctive Normal Form (CNF), compute an assignment to the variables that maximizes the number of satisfied clauses. Variations of the MaxSAT problem include the partial MaxSAT, the weighted MaxSAT problem and the weighted partial MaxSAT problem. In the partial MaxSAT problem some clauses (i.e. the *hard* clauses) must be satisfied, whereas others (i.e. the *soft* clauses) may not be satisfied. In the weighted MaxSAT problem, each clause has a given weight, and the objective is to maximize the sum of the weights of satisfied clauses. Finally, in the weighted partial MaxSAT, the hard clauses must be satisfied, a weight is associated with each soft clause, and the objective is to maximize the sum of the weights of satisfied clauses. MaxSAT and variants provide a versatile modeling solution and a growing number of practical applications [24], [33], including the ability to solve PB optimization problems. Despite the potential applications, the most effective SAT techniques cannot be applied directly in algorithms for MaxSAT. As a result, the best performing algorithms use branch and bound search with sophisticated bounding [24], [33]. Recent work has shown how to iteratively use SAT for solving MaxSAT [20], [40].

One SAT-related decision problem is Quantified Boolean Formulas (QBF), a well-known example of PSPACE-complete decision problems. QBF finds a large number of potential practical applications, including model checking [17]. A QBF formula is a CNF formula where the Boolean variables are quantified, and is of the form:

$$Q_1\, x_1\; Q_2\, x_2 \ldots\; Q_n\, x_n\; \varphi \tag{16}$$

where $Q_i \in \{\exists, \forall\}$ and $\varphi$ is a CNF formula. Recent algorithms for QBF have integrated and extended the most effective SAT techniques [32]. Nevertheless, the performance improvements in QBF solvers has not been as significant as in SAT solvers.

Besides extensions of SAT based on Boolean domains, a number of extensions exist, including Satisfiability Modulo Theories [4], [22] (also, see [11]).

## V. Conclusions

SAT is an NP-complete decision problem, and all existing algorithms require worst-case exponential time in the size of the problem representation. Nevertheless, modern SAT algorithms are remarkably efficient, capable of solving large complex examples from real applications. The efficiency of SAT algorithms has motivated their use in an ever increasing number of practical applications, ranging from crosstalk noise prediction in integrated circuits to termination analysis of term-rewrite systems, and including model checking of hardware and software systems. This paper provides an overview of some of the most successful applications of SAT, and highlights other representative applications. In addition, examples of well-known extensions of SAT are summarized.

## References

[1] P. A. Abdulla, P. Bjesse, and N. Een, "Symbolic reachability analysis based on SAT solvers," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2000.

[2] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[3] H. R. Andersen and H. Hulgaard, "Boolean expression diagrams (extended abstract)," in *Logic in Computer Science*, 1997, pp. 88–98.

[4] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani, "A SAT based approach for solving formulas over Boolean and linear mathematical propositions," in *International Conference on Automated Deduction*, 2002, pp. 195–210.

[5] O. Bailleux, Y. Boufkhad, and O. Roussel, "A translation of pseudo Boolean constraints to SAT," *Journal on Satisfiability, Boolean Modeling and Computation,* vol. 2, March 2006.

[6] P. Barth, “A Davis-Putnam Enumeration Algorithm for Linear Pseudo-Boolean Optimization,” Max Plank Institute for Computer Science, Tech. Rep. MPI-I-95-2-003, 1995.

[7] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, *Advances in Computers*. Academic Press, 2003, ch. Bounded Model Checking.

[8] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without BDDs,” in *Tools and Algorithms for the Construction and Analysis of Systems*, March 1999, pp. 193–207.

[9] D. Brand, “Verification of large synthesized designs,” in *International Conference on Computer-Aided Design*, 1993, pp. 534–537.

[10] P. Chen and K. Keutzer, “Towards true crosstalk noise analysis,” in *International Conference on Computer-Aided Design*, November 1999, pp. 132–138.

[11] A. Cimatti, “Satisfiability modulo theories,” in *Special Session on SAT at WODES08*, 2008.

[12] K. Claessen, N. Een, M. Sheeran, and N. Sörensson, “SAT-solving in practice,” in *Special Session on SAT at WODES08*, 2008.

[13] E. M. Clarke, O. Grumberg, and A. Peled, *Model Checking*. MIT Press, 1999.

[14] E. M. Clarke, D. Kroening, and F. Lerda, “A tool for checking ANSI-C programs,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2004, pp. 168–176.

[15] S. Cook, “The complexity of theorem proving procedures,” in *Proceedings of the Third Annual Symposium on Theory of Computing*, 1971, pp. 151–158.

[16] A. Darwiche, “New advances in compiling CNF into decomposable negation normal form,” in *European Conference on Artificial Intelligence*, 2004, pp. 328–332.

[17] N. Dershowitz, Z. Hanna, and J. Katz, “Bounded model checking with QBF,” in *International Conference on Theory and Applications of Satisfiability Testing*, 2005, pp. 408–414.

[18] N. Een and N. Sörensson, “An extensible SAT solver,” in *International Conference on Theory and Applications of Satisfiability Testing*, May 2003, pp. 502–518.

[19] ——, “Translating pseudo-Boolean constraints into SAT,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, March 2006.

[20] Z. Fu and S. Malik, “On solving the partial MAX-SAT problem,” in *International Conference on Theory and Applications of Satisfiability Testing*, August 2006, pp. 252–265.

[21] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl, “SAT solving for termination analysis with polynomial interpretations,” in *International Conference on Theory and Applications of Satisfiability Testing*, 2007, pp. 340–354.

[22] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, “DPLL(T): Fast decision procedures,” in *Computer-Aided Verification*, 2004, pp. 175–188.

[23] D. Gusfield and S. Orzach, *Handbook on Computational Molecular Biology*. CRC Press, December 2005, vol. 9, ch. Haplotype Inference.

[24] F. Heras, J. Larrosa, and A. Oliveras, “MiniMaxSat: a new weighted Max-SAT solver,” in *International Conference on Theory and Applications of Satisfiability Testing*, May 2007, pp. 41–55.

[25] D. Jackson, I. Schechter, and I. Shlyakhter, “Alcoa: the Alloy constraint analyzer,” in *International Conference on Software Engineering*, 2000, pp. 730–733.

[26] S. Khurshid and D. Marinov, “TestEra: Specification-based testing of java programs using SAT,” *Autom. Softw. Eng.*, vol. 11, no. 4, pp. 403–434, 2004.

[27] H. Kleine-Büning and T. Lettman, *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, 1999.

[28] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust Boolean reasoning for equivalence checking and functional property verification,” *IEEE Transactions on Computer-Aided Design*, vol. 21, no. 12, 2002.

[29] G. Lancia, C. M. Pinotti, and R. Rizzi, “Haplotyping populations by pure parsimony: complexity of exact and approximation algorithms,” *INFORMS Journal on Computing*, vol. 16, no. 4, pp. 348–359, 2004.

[30] T. Larrabee, “Test pattern generation using Boolean satisfiability,” *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 1, pp. 4–15, January 1992.

[31] C.-C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko, “Scalable exploration of functional dependency by interpolation and incremental SAT solving,” in *International Conference on Computer-Aided Design*, 2007, pp. 227–233.

[32] R. Letz, “Lemma and model caching in decision procedures for quantified Boolean formulas,” in *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, July 2002, pp. 160–175.

[33] C. M. Li, F. Manyà, and J. Planes, “New inference rules for Max-SAT,” *Journal of Artificial Intelligence Research*, vol. 30, pp. 321–359, 2007.

[34] I. Lynce and J. Marques-Silva, “Efficient haplotype inference with Boolean satisfiability,” in *National Conference on Artificial Intelligence*, July 2006.

[35] P. Manolios, M. G. Oms, and S. O. Valls, “Checking pedigree consistency with PCS,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2007, pp. 339–342.

[36] V. Manquinho and J. Marques-Silva, “Search pruning conditions for Boolean optimization,” in *European Conference on Artificial Intelligence*, August 2000, pp. 130–107.

[37] J. Marques-Silva and T. Glass, “Combinational equivalence checking using satisfiability and recursive learning,” in *Design, Automation and Test in Europe Conference*, March 1999, pp. 145–149.

[38] J. Marques-Silva and K. Sakallah, “GRASP: A new search algorithm for satisfiability,” in *International Conference on Computer-Aided Design*, November 1996, pp. 220–227.

[39] ——, “Robust search algorithms for test pattern generation,” in *IEEE Fault-Tolerant Computing Symposium*, June 1997, pp. 152–161.

[40] J. Marques-Silva and J. Planes, “Algorithms for maximum satisfiability using unsatisfiable cores,” in *Design, Automation and Testing in Europe Conference*, March 2008.

[41] P. C. McGeer, A. Saldanha, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “Timing analysis and delay-fault test generation using path-recursive functions,” in *International Conference on Computer-Aided Design*, 1991, pp. 180–183.

[42] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[43] ——, “Interpolation and SAT-based model checking,” in *Computer-Aided Verification*, 2003, pp. 1–13.

[44] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Engineering an efficient SAT solver,” in *Design Automation Conference*, June 2001, pp. 530–535.

[45] D. A. Plaisted and S. Greenbaum, “A structure-preserving clause form translation,” *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293–304, September 1986.

[46] J. Rintanen, K. Heljanko, and I. Niemelä, “Planning as satisfiability: parallel plans and algorithms for plan search,” *Artificial Intelligence*, vol. 170, no. 12-13, pp. 1031–1080, 2006.

[47] S. Safarpour, A. G. Veneris, G. Baeckler, and R. Yuan, “Efficient SAT-based Boolean matching for FPGA technology mapping,” in *Design Automation Conference*, 2006, pp. 466–471.

[48] T. Sang, P. Beame, and H. A. Kautz, “Heuristics for fast exact model counting,” in *International Conference on Theory and Applications of Satisfiability Testing*, 2005, pp. 226–240.

[49] B. Selman and H. Kautz, “Planning as satisfiability,” in *European Conference on Artificial Intelligence*, 1992, pp. 359–363.

[50] M. Sheeran, S. Singh, and G. Stalmarck, “Checking safety properties using induction and a SAT solver,” in *Formal Methods in Computer-Aided Design*, 2000, pp. 108–125.

[51] C. Sinz, “Towards an optimal CNF encoding of Boolean cardinality constraints,” in *International Conference on Principles and Practice of Constraint Programming*, October 2005, pp. 827–831.

[52] A. Smith, A. G. Veneris, M. F. Ali, and A. Viglas, “Fault diagnosis and logic debugging using Boolean satisfiability,” *IEEE Transactions on Computer-Aided Design*, vol. 24, no. 10, pp. 1606–1621, 2005.

[53] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, “Combinational test generation using satisfiability,” *IEEE Transactions on Computer-Aided Design*, vol. 15, no. 9, pp. 1167–1176, September 1996.

[54] G. S. Tseitin, “On the complexity of derivation in propositional calculus,” *Studies in Constructive Mathematics and Mathematical Logic, Part II*, pp. 115–125, 1968.

[55] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner, “OPIUM: Optimal package install/uninstall manager,” in *International Conference on Software Engineering*, 2007, pp. 178–188.

[56] T. Walsh, “SAT $v$ CSP,” in *International Conference on Principles and Practice of Constraint Programming*, September 2000, pp. 441–456.

[57] J. P. Warners, “A linear-time transformation of linear inequalities into conjunctive normal form,” *Information Processing Letters*, vol. 68, no. 2, pp. 63–69, 1998.