Notes on Satisfiability-Based Problem Solving
Application Examples

David Mitchell
mitchell@cs.sfu.ca
February 8, 2020

This is a preliminary draft. Please do not distribute. Corrections and suggestions welcome.

In this section, we describe (very basic versions of) methods of solving some application problems by reduction to SAT.

1 Applications to Digital Circuits

A k-ary Boolean function is a function $f : \{0,1\}^k \rightarrow \{0,1\}$. A combinational digital circuit is a device for computing Boolean functions, constructed by composing, or wiring together, components called "gates" that implement simple Boolean functions such as AND, OR and NOT. A circuit computing a k-ary function is said to have k inputs (or input wires) and one output. For example, the circuit of Figure 1 has three inputs, one AND gate, one OR gate, and one output. We can also construct circuits with multiple output wires: a circuit C with k input wires and l output wires computes a function $f_C : \{0,1\}^k \rightarrow \{0,1\}^l$.

Here we illustrate simple versions of two problems involving combinational circuits. For any circuit C with k inputs and one output, we can write a formula of propositional logic that computes the same function as C, using k atoms $X_1, \ldots, X_k$ corresponding to the k input wires $x_1, \ldots, x_k$. For each input vector $\bar{a} \in \{0,1\}^k$, we consider $\bar{a}$ to also be a truth assignment for the atoms with $\bar{a}(X_1) = \mathit{true}$ iff $\bar{a}(x_1) = 1$. The formula must be satisfied by exactly the truth assignments for which the circuit outputs 1. For the circuit of Figure 1, we obtain the formula $((X1 \wedge X2) \vee X3)$.

However, in our applications, we want to model the entire computation performed by a circuit with k inputs and l outputs, sometimes with reference to the internal wires. (Also note that, if we don't have variables corresponding to internal wires, the formula might have to be much larger than the circuit.) To do this, we construct a formula with one propositional atom for each input, each output, and each internal wire that connects an output of one of the gates to the inputs of one or more other gates. We construct the

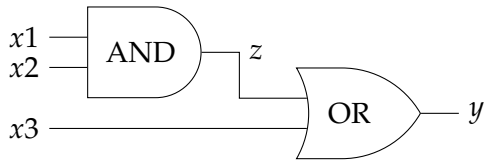

Figure 1: A simple circuit.

formula from sub-formulas modelling the computation of each gate within the circuit. For example, let C be the circuit of Figure 1, with input wires x_1, x_2, x_3, output wire y, and internal wire z. The gates have semantics similar to connectives in propositional logic, with wires taking values in $\{0,1\}$. For example, wire z, the output wire of the AND gate, will have the value 1 if the value applied to both of the gates input wires, $x1$ and $x2$, is 1, and otherwise will have the value 0. The circuit computes a Boolean function $f_C : \{0,1\}^3 \to \{0,1\}$. When values are applied to the input wires, the output wire will have the value $y = f_C(x1, x2, x3)$.

Our formula is $\phi_C = (Z \leftrightarrow (X1 \wedge X2)) \wedge (Y \leftrightarrow (Z \vee X3))$. The satisfying assignments for ϕ_C correspond exactly to the computations of the circuit. In particular, let α be truth assignment for the atoms of ϕ_C. As before, we consider α also to be an assignment of values to the corresponding wires of the circuit, with $\alpha(x1) = 1$ iff $\alpha(X1) = \mathit{true}$, etc. Then the truth assignments that satisfy ϕ_C are exactly those for which $f_C(\alpha) = 1$ iff $\alpha(Y) = \mathit{true}$. That is, a truth assignment α satisfies ϕ_C iff the value $\alpha(Y)$ assigned to the "output atom" Y corresponds to the value $f_C(\alpha(x1), \alpha(x2), \alpha(x3))$ computed by the circuit.

1.1 Circuit Equivalence

Two circuits with the same number of inputs and outputs are equivalent if they compute the same function. Testing circuit equivalence arises often in the following form. We have a given circuit C, which computes a desired function f_C, but we hope to obtain a better circuit that computes this function, for example one using fewer gates or less energy. If we have a candidate circuit D, we would like to check whether D is equivalent to C or not. That is, we want to determine if, for every tuple $\bar{a} \in \{0,1\}^k$, $f_D(\bar{a}) = f_C(\bar{a})$. We can do this by writing a formula $\phi_{C \neq D}$ that, intuitively, says there is an input on which the outputs of C and D are different.

Given two circuits C and D, each with n inputs and m outputs, we can check if they compute the same function from $\{0,1\}^n$ to $\{0,1\}^m$ as follows. Think of a circuit that combines C and D as in Figure 2, producing a new circuit that computes a function from

$\{0,1\}^n$ to $\{0,1\}^{2m}$. If C and D are not equivalent, there will be an input for this combined circuit for which one of the outputs v_i of D has a different value than the corresponding output y_i of C. We write our formula $\phi_{C \neq D}$ based on this circuit.

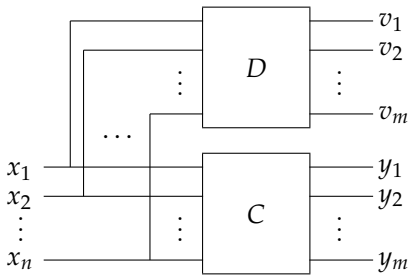

Figure 2: Circuit for testing equivalence of C_A and C_B.

Suppose we have two formulas, ϕ_C and ϕ_D, which model C and D (respectively), as described earlier in this section. Suppose that ϕ_C has atoms $X_1, \ldots, X_n$, for the inputs to C, atoms $Y_1, \ldots, Y_m$, for the outputs of C, and atoms $Z_1, \ldots Z_k$ for the internal wires of C. Further assume (or, by renaming atoms, arrange that) formula ϕ_D has the same atoms $X_1, \ldots, X_n$ corresponding to inputs $x_1, \ldots, x_n$, atoms $V_1, \ldots V_k$ corresponding to outputs $v_1, \ldots v_k$ (with, of course, each v_i corresponding to y_i in the output of C), and atoms $W_1, \ldots, W_j$ corresponding to the internal wires of D.

Now, a formula that models the computation of this circuit is just $(\phi_C \wedge \phi_D)$. To obtain a formula $\phi_{C \neq D}$ that says there is an input to the combined circuit for which the outputs of C and D are different, we simply conjoin to this the statement that the outputs are different, obtaining as $\phi_{C \neq D}$ the formula

$$\phi_C \wedge \phi_D \wedge \neg((Y1 \leftrightarrow V1) \wedge (Y2 \leftrightarrow V2) \wedge \ldots \wedge (Ym \leftrightarrow Vm)).$$

1.2 Automated Test Pattern Generation (ATPG)

In this application, we suppose we are manufacturing chips containing a digital circuit. The manufacturing process is imperfect, and some chips will be flawed. We want to construct tests that detect the most likely flaws in the chips. A common kind of flaw is known as a "single stuck at fault", in which the effect of a chip flaw is that a particular wire in the circuit has a fixed value. To illustrate, suppose that in chips implementing the circuit of Figure 1, sometimes there is a flaw which has the wire z stuck at 0. The flawed version of the circuit is as shown in Figure 3.

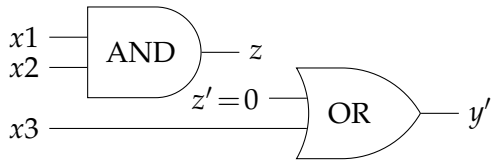

Figure 3: A circuit with a flawed wire stuck at 0.

We want to generate an input for which the correct and flawed circuits will produce different outputs. We can use this to test the chips for the flaw, because on this input the correct chips will give a different output than the flawed chips.

We proceed much as in the equivalence case, but here the circuits are partly the same, and we need only model the way in which they are different. To do this, we make a formula ϕ_{flaw} which models just that part of the circuit affected by the flaw. The only wires that may have values different from the corresponding wires in the correct circuit are z' and y'. So, ϕ_{flaw} is $(\neg Z' \wedge (Y' \leftrightarrow (Z' \vee X_3)))$.

Now, we obtain the following formula, ϕ_{test}, that says there is an input on which the correct and flawed circuits have different output:

$$\phi_C \wedge \phi_{flaw} \wedge \neg(Y \leftrightarrow Y').$$

A satisfying assignment for ϕ_{test} gives an input (the values given to x_1,x_2 and x_3) which constitutes a test pattern for the flaw: a flawed chip will give a different output value than a chip with no flaws.

In this section, we describe a method of solving planning problems by reduction to SAT.

2 Propositional STRIPS Planning

A propositional STRIPS planning instance Π is a tuple $\Pi = \langle F, I, A, G \rangle$ consisting of:

- **Set F of "Facts":** Possible states of the world are described in terms of a set F of "state variables", or "facts", each of which may be true or false. We will take F to be a set of propositional atoms, so each truth assignment to F is a possible state of the world. We denote by $lits(F)$ be the set of literals over F, that is $\{f, \neg f \mid f \in F\}$. The maximal satisfiable subsets of $lits(F)$ are one-to-one with the truth assignments for F, and thus with states of the world.

- **Initial state I:** The initial state is given by a truth assignment for F.
- **Set A of "Actions":** Each action $a \in A$ is defined by two satisfiable sets of literals from $lits(F)$:
  - The set $\text{pre}(a)$ of preconditions of a;
  - The set $\text{eff}(a)$ of effects of a.
- **Goal G:** A set of states, specified by a satisfiable set of literals from $lits(F)$. Any state which satisfies G is a goal state.

An action a is executable in a state S if each literal in $\text{pre}(a)$ is true in S, that is, if $S \models \text{pre}(a)$. The result of executing a in S is the state S' defined by:

$$S'(f) = \begin{cases} f & \text{if } f \in \text{eff}(a) \\ \neg f & \text{if } \neg f \in \text{eff}(a) \\ S(f) & \text{otherwise.} \end{cases}$$

In other words, S' is the same as S, except in-so-far as necessary so that $S' \models \text{eff}(a)$.

A plan P for Π of length L is a sequence of actions $P = \langle a_1, a_2, \ldots, a_L \rangle$ such that there is a sequence of states $S_P = \langle s_0, s_1, \ldots, s_L \rangle$ satisfying

1. $s_0 = I$;
2. For each $1 \leq i \leq L$, action a_i is executable in s_{i-1};
3. For each $1 \leq i \leq L$, state s_i is the result of executing action a_i in state s_{i-1};
4. s_L is a goal state, that is, $s_L \models G$.

2.1 Representing Planning in CNF

Fact 1. *Given as input a propositional STRIPS instance Π, deciding if Π has a plan is PSPACE-complete.*

Intuitively, the reason is that the shortest plan may be of length exponential in the size of the planning instance. As a consequence, representing the set of plans in propositional logic requires formulas which are of size exponential in the size of the instance, which seems undesirable. Instead of doing this, we consider a more convenient task: representing plans of a given length.

Fact 2. *Given as input a propositional STRIPS planning instance Π and a natural number L, deciding existence of a plan of length at most L is NP-complete.*

Actually, we will not model plans of bounded length, but rather plans which are fairly naturally modelled with a bounded number of time-steps. In particular, we will devise a family of formulas Φ_T^Π, parameterized by planning instance Π and positive integer T, with the property that

> Φ_T^Π is satisfiable iff there is a plan for Π involving at most T time steps.

Often, we leave the planning instance Π implicit, writing simply Φ_T.

Notice the shift in perspective here: we defined plans as a sequence of actions, but the description of the formula refers to time steps rather than actions. In fact, the formula we will use:

1. allows time steps in which no action is performed;
2. allows multiple actions to be performed at one time step (with some constraints);
3. bounds the number of time steps, but not (at least directly) the number of actions.

These properties seem to make solving easier. The first allows finding plans without an exactly specified number of steps. Any "no-op" steps can be trivially eliminated in post-processing. The second property allows us to make the formula smaller. The size of the formula needed to represent the plans grows linearly with the number of steps involved, so allowing multiple actions per step reduces the size of formula needed to find plans for a given instance. We call these "parallel plans", but they do not model concurrent actions in any serious way. We require that they be serializable, and therefore we need to add clauses to ensure this.

To write the formula, we will use two sets of atoms:

- **State Atoms:** For each fact $f \in F$ and each time $t \in \{0, \ldots T\}$, we have atom f_t. The intuitive meaning of f_t is that f is true at time t.
- **Action Atoms:** For each action $a \in A$ and each time $t \in \{1, \ldots T\}$, we have atom a_t. The intuitive meaning of a_t is that action a is executed in the t^{th} time step, which is the transition from the state at time $t-1$ to the state at time t.

The formula will be the union of the following sets of clauses, each enforcing a particular constraint on plans. (In some cases, for improved readability, we do not write in clause form, but the translation to clauses is easy.)

1. **Initial State:** The state at time 0 corresponds to the initial state.

   For each fact $f \in F$, include unit clause (f_0) if f is in I, and $(\neg f_0)$ otherwise.

2. **Goal States:** The state at time T satisfies the goal conditions.

For each fact $f \in F$, include unit clause (f_T) if f is in G, and the unit clause $(\neg f_T)$ if $\neg f$ is in G.

3. **Action Preconditions:** If action a is executed in time step t, then the preconditions of a hold a time $t-1$.

   For each action a, and each time $t \in \{1, \ldots, T\}$, include clauses equivalent to

$$a_t \rightarrow \bigwedge_{l \in \text{pre}(a)} l_{t-1}.$$

4. **Action Effects:** If action a is executed in time step t, then its effects hold at time t.

   For each action a, and each time $t \in \{1, \ldots, T\}$, include clauses equivalent to

$$a_t \rightarrow \bigwedge_{l \in \text{eff}(a)} l_t.$$

5. **Explanatory Frame Axioms:** These are to ensure that the state only changes as a result of actions being executed. In particular, if a "fact" changes truth value during some time step, then it must be the effect of an action executed during that step.

   For each fact $f \in F$, and each time $t \in \{1, \ldots, T\}$ include clauses equivalent to:

$$(f_{t-1} \wedge \neg f_t) \rightarrow \bigvee_{\{a | \neg f \in \text{eff}(a)\}} a_t,$$

   and

$$(\neg f_{t-1} \wedge f_t) \rightarrow \bigvee_{\{a |\ f \in \text{eff}(a)\}} a_t.$$

6. **Serializability of Actions:** If multiple actions $\{a_1, a_2, \ldots a_k\}$ are executed during time step t, then we require there to be an ordering of the actions which constitutes a (sequential) plan (because we are using "parallel" plans as a convenience, not to model truly concurrent actions). We may enforce this simply by requiring the actions $a_1, \ldots a_k$ to be pairwise non-conflicting, in the sense that the execution of one does not preclude the other being executed in the resulting state.

   For each pair a, b of distinct actions, if $\text{pre}(a) \cup \text{eff}(b)$ is unsatisfiable, then for each time $t \in \{1, \ldots T\}$, include the clause

$$(\neg a_t \vee \neg b_t).$$

2.2 "Optimal" Planning via Satisfiability

We now have a family of formulas which allow us to use a SAT solver to find plans bounded by some number of time steps. Our goal is to find the shortest plans possible. While finding optimum-length plans would be ideal, we will be satisfied with finding "parallel" plans using a minimum number of time steps. Let T^* denote the minimum number of time steps for which a plan exists. To establish that we have an optimum plan, we will need (at least) two calls to the SAT solver: one to show that Φ_{T^*} is satisfiable, and one to show that Φ_{T^*-1} is unsatisfiable.

Unless we are extremely lucky and guess T^*, we will need to call the solver with a sequence of formulas generated using a sequence of time bounds $\sigma = \langle T1, \ldots Tk \rangle$. We would like to choose this sequence to minimize the total time required to find an optimum plan (or perhaps the best plan we can find within some allotted amount of time). The easiest scheme is to use $\sigma = \langle 1, 2, 3, \ldots, T^*-1, T^* \rangle$. While this seems wasteful, since in most cases T^* is not very close to 1, generating and testing the formulas when T is very small tends to be very fast. This method has the obvious advantage that no guessing is required, and in fact it works quite well in practice, provided T^* is not too large.

Another natural idea is to use binary search. If we know an upper bound T_{UB} on T^*, then we perform a binary search in the interval $[0, T_{UB}]$, which will find T^* in about $\log T_{UB}$ calls to the solver. If we don't know such a bound, we can make a sequence of calls with time bounds $\langle 1, 2, 4, \ldots, 2^i \rangle$, where 2^i is the smallest power of 2 with $2^i \geq T^*$. We know when we reach i because it is the first call to the solver that returns a plan. The last call in this sequence gives us $T_{UB} = 2^i$, after which we perform binary search in the interval $(2^{i-1}, 2^i)$ for T^*, so we find T^* in time $O(\log T^*)$. Unfortunately, minimizing the number of solver calls may not (and typically will not) minimize time, because the running time for the call varies dramatically with T. The typical pattern is, roughly, that time to solve Φ_1 is trivial; times increase dramatically as $T^* - 1$ is approached; solving time drops moderately just above $T^* - 1$, and then increases further (due primarily to the large size of the formula). Thus, the time to establish the optimum value tends to dominate, except in the case that poor guesses well beyond the optimum are not made.