Bounded Model Checking: A brief view

Department of Computer Science & Engineering
Konkuk University
Kunha Kim

Bounded Model Checking

- We construct a Boolean formula that is satisfiable if and only if the underlying state transition system can realize a **finite sequence of state transitions** that reaches certain states of interest.

- If such a path segment cannot be found at a given length, k, the search is continued for large k.

- BMC uses SAT solver, such as PROVER, SATO, GRASP.

Advantages of BMC

- Firstly, SAT Solvers like PROVEE do not require exponential space and large designs can be verified very fast, since the state space is searched in an arbitrary order.

- Also, the procedure is able to find paths of minimal length, which helps the user understand the example that are generated.

- Lastly, the SAT tools generally need far less by hand manipulation than BDDs. Usually the default case splitting heuristics are sufficient.

Disadvantages of BMC

- While the method may be extendable, it has thus far only been used for specifications where fixpoint operations are easy to avoid.
- In addition, the method as applied is generally not complete, meaning one cannot be guaranteed a true or false determination for every specification.
- This is because the propositional formula subject to satisfiability solving grows with each time step, and this greatly inhibits the ability to find long witnesses for counter-example.
- By the way, SAT is used in various filed (formal verification, AI planning, ..)

Definition

- The propositional formula created by a bounded model checker is formed as follows:
  - A transition system, M
  - A temporal logic formula, f
  - A user - supplied time bound k
- We construct a formula $[M,f]_k$ which will be satisfiable iff the formula f is valid along some computation path of M.
- $[M]_k$ is denoted by $I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k)$
- $[M,f]_k$ is denoted by
  $I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge (p(s_0) \vee p(s_1) \vee \cdots \vee p(s_k))$
  when f is EFp

Conversion to CNF

- A formula f in CNF is represented as a set of clause. For example, $((a \vee \neg b \vee c) \wedge (d \vee \neg e)$ is represented as $\{\{a, \neg b, c\}, \{d, \neg e\}\}$
- Given a Boolean formula f, we can convert it to CNF. But if f is in disjunctive normal form, it requires $O(2^n)$
- To avoid the exponential explosion, we use a structure preserving clause form transformation.

Safety property example

$$(a' \leftrightarrow \neg a) \wedge \left(b' \leftrightarrow a \oplus b\right)$$

$$\begin{aligned}
I(s_0):&\quad (\neg a_0 \wedge \neg b_0) \wedge \\
T(s_0, s_1):&\quad ((a_1 \leftrightarrow \neg a_0) \wedge (b_1 \leftrightarrow (a_0 \oplus b_0))) \wedge \\
T(s_1, s_2):&\quad ((a_2 \leftrightarrow \neg a_1) \wedge (b_2 \leftrightarrow (a_1 \oplus b_1))) \wedge \\
p(s_0):&\quad (a_0 \wedge b_0 \vee \\
p(s_1):&\quad a_1 \wedge b_1 \vee \\
p(s_2):&\quad a_2 \wedge b_2)
\end{aligned}$$

Liveness property example

- Let function $inc(s, s')$ as $(a' \leftrightarrow \neg a) \wedge (b' \leftrightarrow (a \oplus b))$.
- On this example, $T(s, s') = inc(s, s') \wedge (b \wedge \neg a \wedge b' \wedge \neg a')$
- We want to know new counter must eventually reach state (1,1). It means $AF(b \wedge a)$, and this can be expressed as EGp, $p = \neg b \vee \neg a$.
- We assume the time bound $k = 2$

Liveness property example

$$T(s_2, s_3) \wedge (s_3 = s_0 \ \vee \ s_3 = s_1 \ \vee \ s_3 = s_2)$$

$$\begin{array}{rllr}
I(s_0): & (& \neg a_0 \wedge \neg b_0 &) \wedge \\
T(s_0, s_1): & & ((a_1 \leftrightarrow \neg a_0) \wedge (b_1 \leftrightarrow (a_0 \oplus b_0)) \ \vee & \\
& & b_1 \wedge \neg a_1 \wedge b_0 \wedge \neg a_0 &) \wedge \\
T(s_1, s_2): & & ((a_2 \leftrightarrow \neg a_1) \wedge (b_2 \leftrightarrow (a_1 \oplus b_1)) \ \vee & \\
& & b_2 \wedge \neg a_2 \wedge b_1 \wedge \neg a_1 &) \wedge \\
T(s_2, s_3): & & ((a_3 \leftrightarrow \neg a_2) \wedge (b_3 \leftrightarrow (a_2 \oplus b_2)) \ \vee & \\
& & b_3 \wedge \neg a_3 \wedge b_2 \wedge \neg a_2 &) \wedge \\
s_3 = s_0: & (& (a_3 \leftrightarrow a_0) \wedge (b_3 \leftrightarrow b_0) \quad \vee & \\
s_3 = s_1: & & (a_3 \leftrightarrow a_1) \wedge (b_3 \leftrightarrow b_1) \quad \vee & \\
s_3 = s_2: & & (a_3 \leftrightarrow a_2) \wedge (b_3 \leftrightarrow b_2) &) \wedge \\
p(s_0): & (& \neg a_0 \vee \neg b_0 &) \wedge \\
p(s_1): & (& \neg a_1 \vee \neg b_1 &) \wedge \\
p(s_2): & (& \neg a_2 \vee \neg b_2 &)
\end{array}$$

Procedure $bool - to - cnf$

- Given a Boolean formula f, $bool - to - cnf(f, true)$ returns a set of clauses C which is satisfiable iff f is satisfiable.

- The procedure traverses the syntactical structure of f, introduces a new variable for each subexpression, and generates clauses that relate the new variables.

- v_f, v_g, v_h are new variables introduced for f, g and h. C_1 and C_2 are sets of clauses.

- Procedure clause( ) translates a Boolean formula into clause form.

- Clause( ) will never be called with more than 3 literals. Thus, in practice, the cost of this procedure is quite acceptable.

procedure bool-to-cnf(f, v_f)

{

  case

    <u>cached(f) == v:</u>

      return clause($v_f \leftrightarrow v$);

    <u>atomic(f):</u>

      return clause($f \leftrightarrow v_f$);

    <u>$f == h \circ g$:</u>

      C_1 = bool-to-cnf(h, v_h);

      C_2 = bool-to-cnf(g, v_g);

      cached(f) = v_f;

      return clause($v_f \leftrightarrow v_h \circ v_g$) $\cup C_1 \cup C_2$;

  esac;

}