2 BACKGROUND

2.1 Model Checking

Model checking is a method to verify whether a model obtained from hardware or software system satisfies a formal specification, written as temporal logic formulas such as *Linear Time Logic* (LTL) [87] or *Computational Tree Logic* (CTL) [88, 89]. The model, capturing the behavior of the system, is expressed as a Kripke Structure as defined below.

Definition 2.1 A Kripke Structure is a quadruple $K=<S,I,T,L>$ where
S, is the finite set of states
$I \subseteq S$, is the set of initial states
$T \subseteq S \times S$, is the set of transitions
$L: S \rightarrow 2^{V}$, is the labeling function with variables set V such that $L(s)$ denotes the set of variables that hold true in state $s \in S$.

Kripke structures do not make any distinction between inputs, outputs or state variables, and transitions are not guarded. Such structures determine the set of computations as an unrolled tree structure from a root state.

Definition 2.2 An LTL formula φ in non-negated form (NNF) is expressed as

$\varphi = true \mid false \mid p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \boldsymbol{X}\, \varphi \mid \boldsymbol{F}\, \varphi \mid \boldsymbol{G}\, \varphi \mid [\varphi_1\, \boldsymbol{U}\, \varphi_2] \mid [\varphi_1\, \boldsymbol{R} \varphi_2] \mid [\varphi_1\, \boldsymbol{W}\, \varphi_2]$, where

p:	Atomic Proposition (AP)		
$\boldsymbol{X}$:	*Next* operator	$\boldsymbol{F}$:	*Eventually* operator
$\boldsymbol{G}$:	*Always* operator	$\boldsymbol{R}$:	*Release* operator
$\boldsymbol{U}$:	*Until* operator	$\boldsymbol{W}$:	*Weak Until* operator

The semantics of an LTL formula is interpreted over a computation path, i.e., sequence of states, $\pi = s_0, s_1, s_2,\ldots$ where for every $j \geq 0$, $s_j \in 2^{AP}$ is the subset of atomic propositions that hold in the j^{th} position of π. Symbolically, we write $\pi \models f$ to denote that f is valid along the path π and is defined recursively as follows (*iff* is shorthand for *if and only if*, and pi^{i} = s_i, s_i+1, s_i+2,....)

$\pi \models true$	$\pi \not\models false$	
$\pi \models p$	*iff*	$p \in s_0$,
$\pi \models \neg p$	*iff*	$p \notin s_0$,
$\pi \models \varphi_1 \wedge \varphi_2$	*iff*	$\pi \models \varphi_1$ and $\pi \models \varphi_2$
$\pi \models \boldsymbol{X}\varphi$	*iff*	$\pi^j \models \varphi$
$\pi \models \boldsymbol{F}\varphi$	*iff*	$\exists i\ \pi^i \models \varphi$
$\pi \models \boldsymbol{G}\varphi$	*iff*	$\forall i\ \pi^i \models \varphi$
$\pi \models \varphi_1 \boldsymbol{U} \varphi_2$	*iff*	$\exists i\ \pi^i \models \varphi_2$ *and* $\forall j, j < i\ \pi^j \models \varphi_1$
$\pi \models \varphi_1 \boldsymbol{R} \varphi_2$	*iff*	$\forall i\ \pi^i \models \varphi_2$ *or* $\exists j, j < i\ \pi^j \models \varphi_1$
$\pi \models \varphi_1 \boldsymbol{W} \varphi_2$	*iff*	$\pi \models \varphi_1 \boldsymbol{U} \varphi_2$ *or* $\pi \models \boldsymbol{G}\varphi_1$

The rewrite rules among ***F***, ***G***, ***R***, and ***U*** operators are as follows:
$\boldsymbol{F}\varphi \equiv true\ \boldsymbol{U}\ \varphi \equiv \neg \boldsymbol{G} \neg \varphi$
$\boldsymbol{G}\varphi \equiv false\ \boldsymbol{R}\ \varphi \equiv \neg \boldsymbol{F} \neg \varphi$
$\varphi_1 \boldsymbol{R}\ \varphi_2 \equiv \neg(\neg\varphi_1\ \boldsymbol{U} \neg\varphi_2)$

An LTL formula φ is *universally valid* in a Kripke Structure M, (symbolically, $M \models A\varphi$) *iff* for all paths π in M such that $s_0 \in I$, $\pi \models \varphi$. Similarly, an LTL formula φ is *existentially valid* in a Kripke Structure M, (symbolically, $M \models E\varphi$) *iff* there exists a path in M, such that $s_0 \in I$, $\pi \models \varphi$. Checking universal validity of an LTL formula φ is same as checking $\neg\varphi$ is *not* existentially valid.

2.1.1 Correctness Properties

We will focus mainly on simple reachable, safety and liveness properties with limited nesting.

- A *reachable property*, denoted as ***EF****p*, specifies that a particular state is reachable from present state.
- A simple *safety property*, denoted as ***AG****p*, specifies that on all paths (***A***) of a system, globally (***G***) in each state of the path, the property *p* holds. Safety properties are used to express "nothing bad will take place in the system, ever".
- A *liveness property*, denoted as ***AF****p*, specifies that on all paths (***A***), eventually *p* will hold in some state in the path. Liveness properties

express behaviors such as "something good will take place eventually, always" [90].

A counter-example to a safety property is a finite length execution trace, while that to a liveness property is an infinite length trace. In the presence of *fairness constraints* — i.e., constraints that a set of states should occur infinitely often along a path — a counter-example trace to a liveness property should also satisfy such fairness constraints. For a finite-state system, such an infinite trace counter-example consists of a loop on states where the "good" never happens and all fairness constraints are satisfied at least once. Safety properties include most common properties like deadlock detection, mutual exclusion, invariants, coverage (state/block/line) checking, and promptness requirements. Liveness properties, on the other hand, include subtle behaviors like starvation freedom, making progress, termination, guaranteed service and receiving service.

2.1.2 Explicit Model Checking

Model checking techniques based on explicit state enumeration such as *Mur*φ [15, 91] and *SPIN* [16, 92] use an explicit representation of states and transitions in the system, and enumerate all reachable states explicitly. To overcome the state explosion, several state reduction techniques are employed such as symmetry detection, reversible rules, repetition constructors, partial order methods and incomplete techniques such as probabilistic verification. Other techniques such as parallelization, caching and using storage disks are also used to make the approach scalable. Explicit model checkers have serious limitations due to explicit enumeration of large state space and are largely unviable for verifying hardware designs.

2.1.3 Symbolic Model Checking

Symbolic model checkers such as *SMV* [17], use symbolic representation of the states using BDDs [18, 19] and traverse the state space symbolically using efficient symbolic algorithms [17, 20-22]. Core steps in symbolic model checking are the *image* and *pre-image* computations, which compute the set of states reachable in single step from/to a given set of states via the transition relations. These operations are shown below:

$$\begin{aligned} Img(\delta, X) &= \exists X,U.\ F(X) \wedge \delta(X,Y,U) \\ PreImg(\delta, Y) &= \exists Y,U.\ T(Y) \wedge \delta(X,Y,U) \end{aligned} \tag{2.1}$$

where the variable sets *X, Y* and *U* denote the present state, next state and primary input variables, respectively; and *F, T*, and δ denote the next states, the current states and the transition relation, respectively. The state sets are represented as BDDs, BEDs [43], or combination [68] of CNF clauses and BDDs, and symbolic computation such as conjoining and existential quantification are carried out on these representations. For large designs, these operations can cause a blow up in a memory size. Use of partitioned transition relation [32] and partitioned symbolic traversal [93] has been shown to improve scalability of these operations to some extent.

A basic algorithm *Model_check* for symbolic model checking simple safety properties can be formulated as shown in Figure 2.1. Let *B* be the bad states, in which the property *p* does not hold, and *I* the set of initial states. The forward reachability analysis starts from the initial states *I* and computes the image operations (*Img*) in successive states. If the bad states intersect with explored states so far, the algorithm returns *CEX* indicating "violation found" else it continues till it explores all reachable states without any violation, indicating a proof of correctness. Backward reachability analysis is similar to the forward analysis, except that the exploration starts backwards from the bad states and uses symbolic *preimage* computations.

```
1.  Synopsis:  Model check simple safety property
2.  Input:     Initial states I, Transition relation δ,
3.             Bad States B
4.  Output:    CEX/Proof
5.  Procedure: Model_check
6.
7.  R=F=N=I;
8.  while (N ≠ ∅ ) {// fixed point?
9.    if (B ∩ N ≠ ∅ )
10.     return CEX; // Violation found
11.   F = Img(δ, N);
12.   N = R \ F;
13.   R = R ∪ N; //Forward reachable states
14. }
15. return Proof; // No Violation Possible
```

Fig. 2.1: Symbolic forward traversal algorithm for safety property

2.2 Notations

We describe the basic notations used in BDD-based and SAT-based model checking applications, described later.

Let V be a finite set of n variables over the set of Boolean values $B\in\{0,1\}$. A *literal* is a variable $v\in V$ or its negation $\neg v$. A *cube* c is a product of one or more distinct literals. A *minterm* over a set of m variables is a product of m distinct literals in the set. A minterm m is contained in a cube c, $m\in c$ iff $c\wedge m=m$. A *clause* is a disjunction of one or more distinct literals and is identified as a set of literals. In the sequel, clause and cube will be used interchangeably to represent the same set of literals. A cube, minterm, or clause are considered trivial if they have both v and $\neg v$. In the sequel, we consider only non-trivial cubes, minterms, and clauses. A *CNF formula* is a conjunction of clauses and is identified as a set of clauses.

A *Boolean function* f is a mapping between Boolean spaces f: $B^n\rightarrow B$. Let $f(v_1\ldots v_n)$ be a Boolean function of n variables. We use *supp(f)* to denote the support set $\{v_1\ldots v_n\}$ of f. In the sequel, we will use function to denote a Boolean function. The positive and negative cofactors of $f(v_1\ldots v\ldots v_n)$ with respect to a variable v are $f_v=f(v_1\ldots 1\ldots v_n)$ and $f_{\neg v}=f(v_1\ldots 0\ldots v_n)$, respectively. Existential (Universal) quantification of $f(v_1\ldots v\ldots,v_n)$ with respect to a variable v is $\exists v\, f=f_v\vee f_{\neg v}$ ($\forall v\, f=f_v\wedge f_{\neg v}$). The cofactor f_c of f by a cube c is obtained by applying a sequence of positive (negative) cofactoring on f with respect to variable v ($\neg v$) in c. A function is a sum of minterms over its support variables and is identified with a set of minterms. The *onset* of a function f is a set of minterms m_i s.t. $f(m_i)=1$. A sum and product of functions over the same domain can be viewed as the union ($\cup$) and intersection ($\cap$) of their onsets, respectively. A function f is said to imply ($\Rightarrow$) another function g, if the onset of f is a subset ($\subseteq$) of the onset of g. In the sequel, we use $f\subseteq g$ to denote that f implies g.

For a finite state machine, let $U=\{u_1\ldots u_k\}$ denote the set of k input variables and $X=\{x_1\ldots x_m\}$ denote the set of m state variables, where $U\cup X=V$ and $U\cap X=\varnothing$. In the sequel, we denote a latch (state holding element) with a state variable. A state $s\in S$ (S is a finite set of states) is a minterm over X. A state cube (and a state clause) consists of only state variables. A characteristic function Ω_Q: $X^m\rightarrow B$ represents a set of states Q ($\subseteq S$), i.e., $Q=\{s \mid \Omega_Q(s)=1 \wedge s\in S\}$. An assignment is a function $\alpha: V_\alpha\rightarrow B$ where V_α ($\equiv dom(\alpha)$) $\subseteq V$. An assignment α_T is called *total* when $V_\alpha=V$; otherwise, it is called *partial*. An assignment α is a *complete satisfying assignment* for a formula f when f evaluates to 1 after applying $v=\alpha(v)$ for all $v\in V_\alpha$. We sometimes refer to a complete satisfying assignment as a *satisfying solution*. A satisfying solution is called *total* if the satisfying assignment is *total*, else called *partial*. We equate an assignment α with a cube that contains all v such that $\alpha(v)=1$ and all $\neg v$ such that $\alpha(v)=0$. We extend the domain of α to literals, i.e., by $\alpha(\neg v)=1$ we mean $\alpha(v)=0$. A *satisfying state cube* is $\alpha\downarrow X$ (projection of α onto state variables X) and a *satisfying input cube* is $\alpha\downarrow U$ (projection of α onto input variables U).

2.3 Binary Decision Diagrams

We discuss a popular graph-based representation called Binary Decision Diagrams (BDDs) [18, 19] used for representing Boolean functions. A BDD is a directed acyclic graph in which each vertex is labeled by a Boolean variable and has outgoing arcs labeled as "then" and "else" branches. Each arc leads either to another vertex or to a terminal labeled 0 or 1. The value of the function for a given assignment to the inputs is determined by traversing from the root down to a terminal label, each time following the then (else) branch corresponding to the value 1 (0) assigned to the variable specified by the vertex label. The value of the function then equals the terminal value.

In an ordered BDD, all vertex labels must occur according to a total ordering of the variables. In a reduced ordered BDD (ROBDD), besides the ordering criteria, two more conditions are imposed:

a) two nodes with isomorphic BDDs are merged, and
b) any node with identical children is removed.

These conditions make a reduced ordered BDD a canonical representation for Boolean functions. In the sequel, we will use the term BDD to refer to a ROBDD.

Researchers [24-27] have developed several heuristics to obtain a good variable ordering that produce compact BDD representation. Even though finding a good BDD variable ordering is not easy [28, 29], for some functions such as integer multiplication, there does not exist an ordering that gives a sub-exponential size representation [28].

We show the effect of two different variable orderings on the size of a BDD for a given function $f = a_1b_1 + a_2b_2 + a_3b_3$; (example from [19]). As shown in Figure 2.2(a), the BDD requires 8 vertices for an ordering $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$; whereas for an ordering $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$, 16 vertices are required to represent the same function as shown in Figure 2.2(b).

BDDs have many desirable properties for symbolic Boolean manipulation [17, 20-22]. Many binary operations such as AND and OR on Boolean functions can be implemented efficiently as graph algorithms applied to BDDs. Given two BDDs of size m and n, the complexity of a binary operation is $O(m.n)$. Thus, symbolic manipulations involving quantifications $\exists$ and $\forall$ have polynomial complexity in the sizes of BDDs. As BDDs are canonical, tautology checking and complementation can be achieved in constant time. Due to these nice features, BDDs have become primary workhorse for symbolic model checking [17, 20-22]. BDDs are used to represent transition system and state sets as characteristic functions. The primary limitation of the BDD-based approaches, however, is the scalability, i.e., BDDs constructed in course of verification grow extremely large,

resulting in space-outs or severe performance degradation due to paging [23]. Moreover, BDDs are not very good representations of state sets, especially when the sharing of the nodes is limited. Choosing a right variable ordering for obtaining compact BDDs is very important. Finding a good ordering is often time consuming and/or requires good design insight, which is not always feasible. Several variations of BDDs such as Free BDDs [30], zBDDs [31], partitioned-BDDs [32] and subset-BDDs [33] have also been proposed to target domain-specific application; however, in practice, they have not scaled adequately for industry applications. Therefore, BDD-based approaches are limited to designs containing of the order of a few hundred state holding elements; this is not even at the level of an individual designer subsystem.

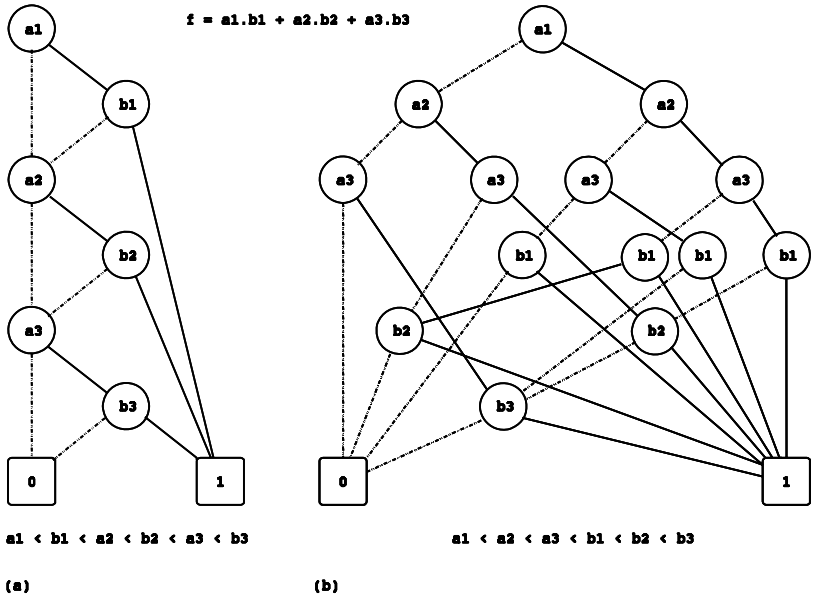

Fig. 2.2: Effect of variable ordering on BDD size: (a) with ordering $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$, and (b) with ordering $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$. A solid line denotes a then-branch and a dashed line denotes an else-branch

2.4 Boolean Satisfiability Problem

The Boolean Satisfiability (SAT) problem is a well-known constraint satisfaction problem, with many applications in the fields of VLSI Computer-Aided Design (CAD) and Artificial Intelligence. Given a propositional formula, the Boolean Satisfiability problem is to determine, whether there exists a variable assignment under which the formula

evaluates to true, or to prove that no such assignment exists. The SAT problem is known to be NP-Complete [94]. In practice, there has been tremendous progress in SAT solver technology over the years, summarized in a survey [34].

Most SAT solvers use a Conjunctive Normal Form (CNF) representation of the Boolean formula. In CNF, the formula is represented as a conjunction of clauses, each clause is a disjunction of literals, and a literal is a variable or its negation. Note that in order for the CNF formula to be satisfied, each clause must also be satisfied, i.e., at least one literal in the clause has to be *true*. A clause with at least two unassigned literals is called *unsatisfied clause*. A *unit clause* is a clause where all literals are false, except one which is unassigned, referred to as the unit literal. A *conflicting clause* is a clause where all literals are false. A Boolean circuit can be encoded as a satisfiability equivalent CNF formula [95, 96]. Alternatively, for SAT applications arising from the circuit domain, it may be more efficient to modify the SAT solver to work directly on the Boolean circuit representation.

Most modern SAT solvers are based on a DPLL-style [35] procedure, shown as *SAT_solve* in Figure 2.3, with three main engines: *decision, deduction*, and *diagnosis*. Here, we focus on the state-of-the-art techniques involved that have allowed the basic SAT algorithm to scale to problem instances with as many as millions of variables. The basic SAT algorithm is a branch-and-search algorithm. A variable that is unassigned, is called a *free* variable. Initially, all variables are unassigned.

```
1. Synopsis:  Check Boolean Satisfiability
2. Input:     Boolean Problem
3. Output:    SAT/UNSAT
4. Procedure: SAT_solve
5.
6. if (Deduce()=CONFLICT) return UNSAT; //Pre-process
7. while(Decide()=SUCCESS) { //branch
8.   while(Deduce()=CONFLICT) { //constraint propagation
9.     blevel = Diagnose(); // conflict-driven learning
10.    if (blevel == 0) return UNSAT;
11.    else backtrack(blevel); //backjump to blevel
12.  }
13.}
14.return SAT;
```

Fig. 2.3: DPLL-style SAT Solver

Pre-processing of the formula is done by a *deduction* engine without making any decision. The *deduction* engine applies the *unit clause rule* [35] repeatedly on *unit* clauses until no such clause exists or a conflicting clause is

detected. As per this rule, the unit literal is set to *true* in order to satisfy the corresponding unit clause. Consecutive application of this rule is also called *unit propagation* or *Boolean Constraint Propagation* (BCP). The deduction engine can also optionally apply *pure literal rule*, where a variable that appears as only positive (negative) literal in remaining unsatisfied clauses is set to *true* (*false*) [35].

The main loop (lines 7-13) in the procedure *SAT_solve* begins with invocation of *decision* engine, wherein a *free* variable is assigned a value (also called *branching literal*). The decision on the branching literal is critical to the performance of the SAT solver. After the branch, the *deduction* engine is called to apply BCP until no more *unit* clause exists or a *conflict* occurs, i.e., conflicting values are implied on a variable. In the former case, the *decision* engine is called again to make a decision at next level; otherwise, a *diagnosis* engine is called to resolve the conflict that involves learning reasons for conflict, and *backtracking*. Note that backtracking corresponds to undoing the assignments made during BCP up to a previous level on the decision stack derived from conflict analysis. Advanced SAT solvers use backjumping or non-chronological backtracking [36-38, 40, 97] during conflict analysis and derive backtrack level and conflict-driven learnt clauses based on implication graph. Diagnosis engine, in some sense, is a correction procedure to the decision strategy that provides guidance to the search procedure. Effectiveness of the guidance is measured by number of backtracks before the problem is resolved. A bad decision strategy will cause more frequent backtracks, affecting the SAT performance overall. Backtrack to decision level 0 indicates that the problem is unsatisfiable (UNSAT). On the other hand the problem is said to be satisfiable (SAT) if the decision engine is unsuccessful in finding any free variable to branch on.

We now discuss various techniques developed to improve each of the SAT engines.

2.4.1 Decision Engine

The choice of the branching literal is critical to the performance of a SAT solver. Many heuristics have been proposed in the past [98, 99] which seem to work on certain classes of problems More recently, a decision strategy called *Variable State Independent Decaying Sum* (VSIDS) [38] has been shown to be both effective and efficient for many classes of problem instances. This strategy improves upon a previously proposed strategy on the use of literal count [98], wherein the current branching literal has the highest occurrence in currently unresolved clauses. In VSIDS strategy, the positive and negative scores of variables are computed initially based on the

occurrence of positive and negative literals in the clauses. Whenever a clause is learned and added, the score of its variable is increased dynamically by a constant. Further, scores of variables are periodically divided by a constant, i.e., decreased, to give more weight to the recently added learned clauses. A branching literal is the highest (positive or negative) scoring variable with that phase. As opposed to [98], VSIDS score does not exactly reflect the occurrence of literal in currently unresolved clauses, but has been found to be very effective and efficient. In another variation [40] of VSIDS, one can also increase the score of all the literals in the clauses that cause a conflict, instead of only those that appear in a conflict-driven learned clause. Further, one can decide to satisfy those conflict-driven clauses added, starting with the most recent ones [40]. This scheme has been shown to be quite effective and robust.

2.4.2 Deduction Engine

BCP constitutes the core of SAT algorithm, taking about 80% of the total time in SAT. Any improvement in BCP directly translates into overall performance gain. We discuss a successful and highly optimized BCP scheme called *lazy update*, first introduced in Chaff [38], which is an improvement over BCP scheme based on head/tail pointer proposed in SATO [36]. In lazy update scheme,

1. For each clause, only two *non-zero literals* are watched.
2. The clause state is updated lazily when the watched literals coincide.
3. Re-positioning of watch pointers after backtrack is not required as they are guaranteed to watch non-zero literals after backtrack.

Note, a literal assignment does not necessarily imply re-positioning of watch pointers in a clause as that literal may not be watched. Also, backtrack cost, modulo variable unassignment, is constant time.

The implications generated during BCP are stored in a FIFO queue or assignment queue. In other words, implications are processed in the order in which they are generated. Since the actual assignment is postponed until it is de-queued, such a scheme is also called *lazy assignment* [100]. In contrast, one can also process the implications as they are generated, essentially in a LIFO order. Such processing of implications is also referred to as *eager assignment* [100]. In lazy assignment, the watch pointers may point to a literal that is implied but not yet assigned value *false* in the queue. Also, watch pointers can get updated even if the clause is satisfied. Though detection of unit and conflict clauses is easy, detection of satisfiable clauses requires additional book-keeping and invariant maintenance which increase

the backtracking cost. One can avoid such overhead in head/tail pointer scheme [36] at the cost of backtrack. Recently, Biere *et al.* have also proposed [100] an efficient way to access the other watched literal to determine if the clause is satisfied. Experimental results, however, have shown that gain due to zero-cost backtrack using simple structure is significantly higher than the overhead of repositioning of the watch pointers using strict invariants [36] or use of additional pointer de-referencing [100].

In the following, we illustrate the lazy update with an example as shown in Figure 2.4. The clause ($-V_1 + V_4 + -V_7 + V_{11} + V_{15}$) has *watch pointers* (shown as arrow)on literals V_4 and V_{15}. With variable assignment V_{15}=0 at decision level 4 (denoted as @4), the right watch pointer moves left to point at V_{12}. With assignments V_7=1@5 and V_{11}=0@5, watch pointers are not repositioned. With V_4=0@5, the clause becomes unit clause with unit literal V_{12}=1. Later, with backtrack to level 4, all variables get unassigned in the clause except V_1 (assuming V_1 got assigned before level 4). Note, the watch pointers are not repositioned during backtrack and thus, backtrack is effectively zero-cost process. Further when V_{12}=1@5, the right watch pointer is not moved. However, with V_4=0@5, the left pointer moves even though the clause is satisfied with V_{12}=1.

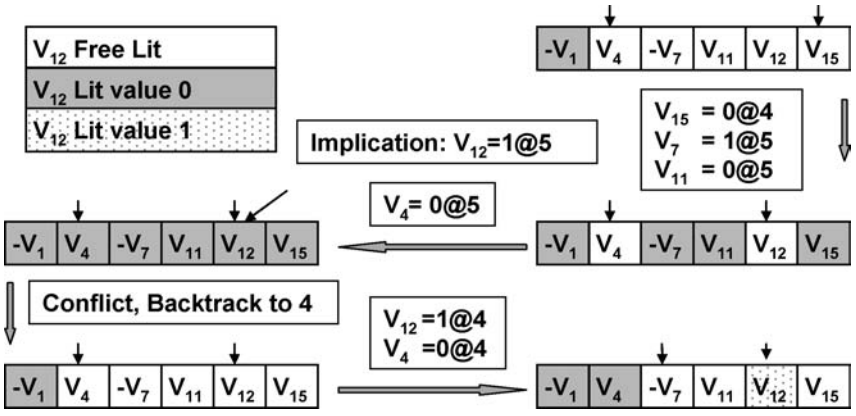

Fig. 2.4: Lazy update of a CNF clause during BCP

For clauses with many literals, a lazy update scheme performs significantly better than a head/tail pointer scheme. This also works well in practice, as conflict-driven learning tends to add very long clauses. Other forms of deduction such as *pure literal rule* [35] and equivalence reasoning [101] can also be applied but are not so robust and efficient in practice.

2.4.3 Diagnosis Engine

A conflict occurs when opposite values are implied during BCP on a variable. When such a conflict is detected, the conflict analysis routine is called to resolve the conflict. The analysis could be as simple as flipping the last decision variable that has not been previously flipped, and sometimes referred to as *chronological backtracking*. It is not very effective in pruning the search space. In contrast, conflict analysis could be more involved, wherein resolution process is applied on the implication graph [36, 37, 97] to learn conflict clauses for *conflict-driven learning* (for learning conflict clauses) and *non-chronological backtracking* [37, 102] (i.e., back-jumping to an earlier decision level).

The conflict-driven learned clauses are generated by applying the following *resolution rule*: $(x+ y)\wedge(-x + z) \Rightarrow (y + z)$. As per unit clause rule, when a clause becomes a unit clause, it *implies* a true value on the unit literal. Such a clause becomes the *antecedent* of the corresponding variable. Thus, every implied variable will have an antecedent clause. The implications are typically stored in the form of an *implication graph*, a digraph with the vertices as variables and edges between the variables corresponding to the antecedent clause of the *to-vertex*. Vertices, with no incoming edges, denote decision variables. A conflict occurs whenever the graph has two vertices for the same variable with opposite values. A small example of an implication graph leading up to a conflict is shown in Figure 2.5.

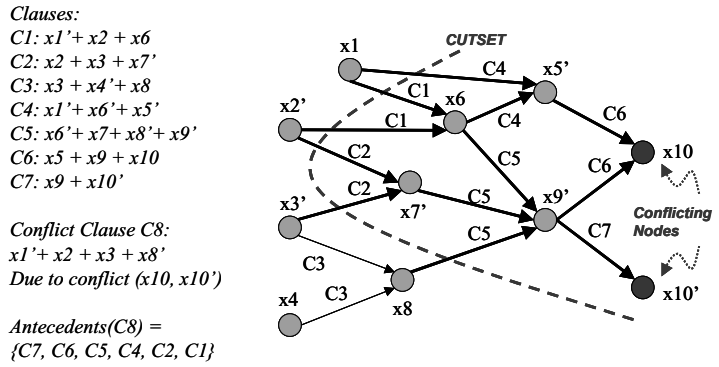

Fig. 2.5: Example for Conflict Analysis

Conflict analysis works on the implication graph by first bipartitioning the graph into a reason and a conflict side, as shown in Figure 2.5. All the vertices on the reason side that have at least one edge to the conflict side

constitute the reasons for the conflict. Such a bipartition is called a *cut.* Each cut corresponds to various learning schemes [103]. Conflict analysis then traverses back the edges leading to the conflicting nodes up to any cutset in this graph. A conflict clause is then derived from the variables feeding into the chosen *cutset*. A key feature of a learned conflict clause is that it is also the result of resolution on all the antecedent clauses, which are traversed up to the chosen *cutset*. In order to make the conflict clause an *asserting clause (*equivalently, *unit clause)*, a cut is chosen such that the unit literal corresponds to the variable with the current decision level, and backtrack level chosen is the maximum decision level (except current decision level) of the variables in the conflict clause.

An *unique implification point* (UIP) [37] is a vertex at the current decision level that dominates both the vertices corresponding to the conflicting variable. (A variable x is said to dominate y *iff* any path from the decision variable of the decision level of x to y needs to go through x.) In *firstUIP* learning scheme, only the UIP at the current decision level is computed, which becomes the unit literal in the conflict clause after backtrack. Other learning schemes such *lastUIP*, *allUIP* and min-cuts requires additional resources for computation and have not been found to be so effective [103] as *firstUIP*.

Conflict-driven learning results in addition of conflict clauses to the SAT problem in order to prevent the same conflict from occurring again during the search. Additionally, information recorded during conflict analysis has been used very effectively to provide a *proof of unsatisfiability*, as described next.

2.4.4 Proof of Unsatisfiability

There have been several independent efforts aimed at extracting resolution-based proofs of unsatisfiability from a CNF-based SAT solver [48, 77, 78, 104, 105]. They are based on recording additional information during conflict analysis and conflict-driven learning such as antecedents (reasons) and their association with learned clauses. As part of the check, these techniques also identify a *subset* of clauses from the original problem, called the *unsatisfiable core*, such that the clauses are sufficient for implying the unsatisfiability. Similar SAT-based proof analysis techniques have also been proposed independently in the context of refinement, and abstraction-based verification methods [48, 77, 78]. We describe the generation of a proof of unsatisfiability in the following.

When a SAT solver determines that a given problem is unsatisfiable, it does so because there is a conflict without any decisions being taken. A conflict analysis can again be used to record the antecedents for this *final*

conflict. This time, the learned conflict clause, i.e., a resolution of all its antecedents is an empty clause. Therefore, this *final resolution proof tree* (also called *final conflict graph*) constitutes a proof of unsatifiability [35], except that it may include some learned clauses. Recall that a learned clause is itself a resolution of the antecedents associated with it. Therefore, by recursively substituting the resolution trees corresponding to the learned clauses into the final resolution tree, a resolution proof only on the original clauses can be obtained. These original clauses constitute an *unsatisfiable core,* i.e., they are sufficient for implying the unsatisfiability. In practice, the resolution tree is created only if a check is needed for the unsatisfiability result [104, 105]. For the purpose of identifying an *unsatisfiable core*, a marking procedure is used, which starts from the antecedents of the final conflict graph, and recursively marks the antecedents associated with any marked conflict clause. At the end, the set of marked original clauses constitutes an *unsatisfiable core*.

The existing resolution-based proof analysis techniques have been described for SAT solvers that use a CNF representation of the Boolean problem. In Chapter 4, we discuss how these techniques can be adapted for hybrid SAT solvers that work directly on circuit structures.

2.4.5 Further Improvements

There has been ongoing research [106] on further improving the SAT solvers. We briefly mention some of the most relevant efforts that have shown promising results on certain problem instances and involve low overhead.

Frequent Restarts

The state-of-the-art SAT solvers also employ a technique called *random restart* [38] for greater robustness. The first few decisions are very important in the SAT solver. A bad choice could make it very hard for the solver to exit a local non-useful search space. Since it is very hard to determine *a priori* what a good choice might be for decisions, the restart mechanism periodically undoes all decisions and starts afresh. The learned clauses are preserved between restarts; therefore, the search conducted in previous rounds is not lost. By utilizing such randomization, a SAT solver can minimize local fruitless search.

Non-conflict-driven Back-jumping

This refers to a backjump to an earlier decision level (not necessarily to level 0), without detecting a conflict [107]. It is a variation of frequent restarts strategy, but guided by the number of conflict-driven backtracks seen so far. The goal is to quickly get out of a "local conflict zone" when the number of backtracks occurring between two decision levels exceeds a certain threshold. For hard problem instances, such strategy has shown promising results.

Frequent Clause Deletion

Conflict-driven learned clauses are redundant, and therefore, deleting them does not affect satisfiability of the problem. Conflict clauses, though useful can become an overhead especially due to increased BCP time and due to large memory usage. Such clauses can be deleted based on their relevance metric [38] which is computed based on number of unassigned literals. One can also compute relevance of a clause based on its frequent involvement in conflict [40].

Clause Shrinking

Effectiveness of conflict-driven learning scheme is very hard to determine *a priori.* In general, a shorter conflict-clause is useful, as it prunes a larger search space. One scheme is to shrink the conflict clause by identifying a sufficient subset of the literals required to generate the conflict [108]. Using the conflict literals as decision variables, one applies BCP and stops as soon as a conflict is detected. In many cases, fewer literals in the conflict clause are involved.

Early Conflict Detection in Implication Queue

The implication queue (in lazy assignment) stores the newly implied variables during BCP. If a newly implied variable is already in the queue with an opposite implied value, one can detect conflict early without doing BCP further [100].

Shorter Reasons First

Several unit clauses can imply the same value on a variable at a given decision level. With an intuition that shorter unit clauses decrease the size of

implication graph, and hence, the size of conflict clauses, the implications due shorter clauses are given preference [100].

2.5 SAT-based Bounded Model Checking (BMC)

Bounded Model Checking refers to a model checking approach where the number of steps in forward traversal of the state space are bounded. The approach reports either "violation found" or "no violation possible within the bounded depth". Symbolic simulation can be considered as bounded model checking, restricted to invariants checking. Such an approach does not mandate storing state space and hence, is found to be more scalable and useful.

In SAT-based BMC [66, 67], the specifications are expressed in LTL (Linear Temporal Logic). To keep the discussion simple, we consider formulas of the existential type $\boldsymbol{E}f$ where f is a temporal formula with no path quantifiers. In the sequel, we sometimes drop the quantifier $\boldsymbol{E}$ and implicitly imply an existential LTL formula, if not obvious from the context. Given a Kripke structure M, an LTL formula f, and a bound k, the translation task in BMC, i.e., henceforth denoted as *BMC(M,f,k),* is to construct a propositional formula $[M, f]_k$, such that the formula is satisfiable if and only if there exists a witness of length k. $[M, f]_k$ is defined as follows:

$$
\begin{aligned}
&[M, f]_k = [M]_k \wedge [f]_k, \text{ where} \\
&[M]_k = I(s_0) \wedge \wedge_{0 \le i < k} T(s_i, s_{i+1}) :: \textit{initial and circuit constraints} \\
&[f]_k = ((\neg L_k \wedge [f]_k^0) \vee (\vee_{0 \le l \le k} ({}_lL_k \wedge {}_l[f]_k^0))) :: \textit{property translation} \\
&{}_lL_k = T(s_k, s_l) :: \textit{loop transition from state } s_k \textit{ to } s_l \ (l \le k) \\
&L_k = \vee_{0 \le l \le k} {}_lL_k :: \textit{disjunction of loops of length up to } k
\end{aligned}
\tag{2.2}
$$

Since the set of states is finite, a symbolic encoding in terms of Boolean variables denoted as s is used to represent a state, and $s_0 \ldots s_k$ is used to represent a finite (k+1)-length sequence of states.

The first set of constraints, denoted $[M]_k$, ensures that this sequence is a valid path in M. The first part of this formula imposes the constraint that the first state in the sequence should be an initial state I. The second part of this formula imposes the constraint that every successive pair of states should be related according to the transition relation T. Note that this second part corresponds to an unrolling of the sequential design for k steps, and results in an increasing SAT problem size with increasing k.

The next set of constraints $[f]_k$ ensures that this valid path in M satisfies the LTL formula f. This involves a case split, depending upon whether or not

the path is a *(k, l)* -loop, as shown in Figure 2.6 (taken from [66]). The case of a loop from state k to state l can be translated as ${}_lL_k = T(s_k, s_l)$. The case where there is no loop can be translated as $\neg L_k$ where $L_k = \vee_{0 \le l \le k}\, {}_lL_k$. Let $[f]_k^0$ denote the translation for formula f in the no loop case, and ${}_l[f]_k^0$ denote the translation for f in the *(k, l)*-loop case (detailed definitions are in [66]).

(a) (b)

Fig. 2.6: BMC check (a) No loop (b) Loop

For our discussion later, we call the first conjunct $[M]_k$ as the *circuit constraints*, and the second conjunct $[f]_k$ as *property translation*. Note that the general translation is monolithic, i.e., the entire propositional formula is generated for a given k. This formula is then checked for satisfiability using standard SAT solvers [35].

In practice, the search for longer witnesses is conducted by incrementing the bound k. This works well when a witness does exist. However, in case there is no witness, an additional proof technique is needed to conclude that the property is indeed false. In particular, it is sufficient to examine all k up to the diameter of the finite state machine [66]. Use of additional constraints such as loop-free paths, shortest paths, and inductive invariants [73], have also been proposed in a similar setting for proving safety properties [67, 109].

2.5.1 BMC formulation: Safety and Liveness Properties

A counter-example to a safety property is a finite trace. Most used safety properties are simple, i.e., $\boldsymbol{G}(\neg p)$ where p is a propositional atom. A standard translation for the negation of the property can omit the loop condition since a finite trace is sufficient for falsifying the safety property, as shown below:

$$[\neg \boldsymbol{G}(\neg p)]_k = [\boldsymbol{F}(p)]_k = [M]_k \wedge [\vee_{0 \le t \le k}\ p^t]_k \;;;\; p^t \equiv \textit{node p at depth t} \tag{2.3}$$

Recall, a liveness property expresses that eventually "something good will take place" along an execution path. Presence of fairness constraints $B = \{f_1,\ldots,f_b\}$ puts additional requirement that each $f_r \in B$ should hold infinitely often along the execution path. Liveness properties $\boldsymbol{F}(\neg q)$, $\boldsymbol{G}(p \rightarrow \boldsymbol{F}(\neg q))$ in the presence of fairness constraints B expressed as $\boldsymbol{G}(\wedge_{1 \le r \le b} \boldsymbol{F}(f_r))$ are:

$$\boldsymbol{G}(\wedge_{1\le r\le b}\boldsymbol{F}(f_r)) \rightarrow \boldsymbol{F}(\neg q)$$
$$\boldsymbol{G}(\wedge_{1\le r\le b}\boldsymbol{F}(f_r)) \rightarrow \boldsymbol{G}(p\rightarrow\boldsymbol{F}(\neg q))$$
(2.4)

The standard LTL translations [66, 110, 111] for the negation of the above properties where *p, q* are propositional atoms are as follows:

$$[\neg(\boldsymbol{G}(\wedge_{1\le r\le b}\boldsymbol{F}(f_r)) \rightarrow \boldsymbol{F}(\neg q))]_k = [\boldsymbol{G}(q \wedge (\wedge_{1\le r\le b}\boldsymbol{F}(f_r)))]_k$$
$$=[M]_k \wedge [\vee_{0\le l\le k}({}_lL_k \wedge (\wedge_{1\le r\le b}(\vee_{l\le j\le k} f_r^j)) \wedge (\wedge_{0\le i\le k} q^i))]_k$$
(2.5)

$$[\neg(\boldsymbol{G}(\wedge_{1\le r\le b}\boldsymbol{F}(f_r))\rightarrow\boldsymbol{G}(p\rightarrow\boldsymbol{F}(\neg q)))]_k$$
$$=[\boldsymbol{F}(p\wedge\boldsymbol{G}(q\wedge(\wedge_{1\le r\le b}\boldsymbol{F}(f_r))))]_k$$
$$=[M]_k\wedge [\vee_{0\le l\le k}({}_lL_k\wedge(\vee_{0\le t\le k}(p^t\wedge(\wedge_{min(t,l)\le i\le k} q^i)\wedge(\wedge_{1\le r\le b}(\vee_{min(t,l)\le j\le k} f_r^j)))))]_k$$
(2.6)

A counter-example to a liveness property $\boldsymbol{G}(\boldsymbol{F}(f))\rightarrow \boldsymbol{G}(p\rightarrow\boldsymbol{F}(\neg q))$ in the presence of a fairness constraint *f* is shown in Figure 2.7. Note, the fairness requirement is that *f* should hold at least once between the looping states i.e., s_l and s_k.

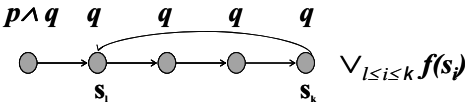

Fig. 2.7: Debug trace for $\boldsymbol{G}(p\rightarrow\boldsymbol{F}(\neg q))$ with fairness *f*

Another form of liveness property that is commonly used is *bounded liveness*, which is typically expressed as $(q\wedge_{1\le n\le M}\boldsymbol{X}{:}n(q)$ and $\boldsymbol{G}(p\rightarrow q\wedge_{1\le n\le M}\boldsymbol{X}{:}n(q))$ where *M* is the liveness bound and $\boldsymbol{X}{:}n$ is shorthand hand for $\boldsymbol{X}\wedge\ldots\wedge\boldsymbol{X}$, repeated n times.

Safety properties have generated a lot of interest among researchers due to their relative importance in practice. Recently, there have been several effective SAT-based techniques – BMC [66, 110], induction [67, 73], unbounded model checking (UMC) [47, 68, 71, 112], and proof-based abstraction [48, 78] – that can handle safety properties very efficiently compared to BDD-based approaches like symbolic model checking [17]. Though many of these techniques can be applied to checking all properties in principle, there has been a general lack of efforts dedicated to liveness, barring a few described later in Chapters 5 and 10.

As reported in a recent workshop [113], one school of thought believes that rigorous reasoning about liveness is tedious and therefore, it is not worth the effort given finite resource for verification. Another school of thought, along similar lines, believes that liveness without a bound is not useful and

bounded liveness can be reduced to a safety property; therefore, we should use safety checking only. We believe, like an alternative school of thought, that liveness is important and that it too deserves dedicated model checking techniques. We list some of the main reasons cited:

- If we restrict verification to only safety checking, it would be impossible to detect subtle design errors that prevent some behavior from happening eventually.
- Translation of bounded liveness to safety is expensive especially when the bound is large. Moreover, designers may not always provide a safe bound.
- Fairness constraints are used to capture the interaction of the design with the environment. Since formal verification is usually applied at the block level, fairness is needed to model the environment correctly. Therefore, liveness with fairness is important to capture subtle system behaviors.
- Model checking for all LTL properties can be reduced to checking of a fairness constraint.

In general, liveness is handled as a standard LTL property, by converting the negated formula to a Buchi automaton and checking for language emptiness of its product with the design model [114]. Such a general translation may not be very effective, and many improved translations [115-117] have been proposed. SAT-based BMC [66] handles liveness as part of a standard translation of an LTL formula to a propositional satisfiability problem where the challenge is to find a state loop. BMC is incomplete without a *completeness bound* [66, 67, 72], and generally it is difficult to obtain such a bound. However for safety checking, the completeness has been addressed by several other methods: induction [67, 73], SAT-based UMC [47, 68, 71, 112], and proof-based abstraction [48, 78]. This inspired some researchers [118] to reduce the liveness property to a safety property and use the complete safety checking algorithms. However, this translation is based on a state recording approach that doubles the number of state bits in the design. Such a translation adds significant overhead to safety checking algorithms and is therefore, not very practical. In another recent work [119], authors reason about a model composed with Buchi automata obtained from translation of the negated LTL formula, and provide completeness bound for a fairness constraint based on loop-free path analysis. Such bounds may be quite large in practice, and are therefore not very useful.

Recently we have proposed [46] SAT-based techniques geared towards verifying liveness properties of the form $\boldsymbol{F}(q)$ and $\boldsymbol{G}(p \rightarrow \boldsymbol{F}(q))$ with fairness B. These techniques can be extended to handle commonly used LTL properties. We use dedicated SAT-based translations for bounded model

checking of non-safety properties, and determine the completeness bounds for liveness using SAT-based unbounded analysis. This is discussed in more detail in Chapters 5 and 10.

2.5.2 Clocked LTL Specifications

Property variables that involve gates with support from state elements in multiple clock domains require the use of clocks in the formula to avoid ambiguities. The Property Specification Language (PSL) standardized by Accellera [83] has formal semantics for specifying clocked properties using the clock operator @, based largely on the work of Eisner *et al.* [120]. A clocked LTL specification *(f)@clk*, expressed under the context *clk* using clock operator @, can be equivalently translated [120] into an un-clocked LTL specification (with an implicit global clock tick) $T^{clk}(f)(\equiv (f)@clk)$ where $T^{clk}(f)$ is defined recursively using the following rules *R1-6*:

$$
\begin{aligned}
&R1: T^{clk}(p) = \neg clk\ \mathbf{U}\ (clk \wedge p)\ ;;f \text{ is propositional atom } p\\
&R2: T^{clk}(\neg f) = \neg T^{clk}(f)\\
&R3: T^{clk}(f_1 \wedge f_2) = T^{clk}(f_1) \wedge T^{clk}(f_2)\\
&R4: T^{clk}(\mathbf{X} f) = \neg clk\ \mathbf{U}\ (clk \wedge \mathbf{X}(\neg clk\ \mathbf{U}\ (clk \wedge T^{clk}(f))))\\
&R5: T^{clk}(f_1 \mathbf{U} f_2) = (clk \rightarrow T^{clk}(f_1))\ \mathbf{U}\ (clk \wedge T^{clk}(f_2))\\
&R6: T^{clk}((f)@clk1) = T^{clk1}(f)
\end{aligned}
\tag{2.7}
$$

Rules for other LTL operators ***F***, ***G*** and ***W*** can be derived from the above rules. Note, that in [120], the authors have differentiated temporal operators and propositional atoms as weak or strong using the strength operator **!**. A strong atomic proposition *p***!** does not hold on an *empty path*, compared to a weak atomic proposition *p* which does. For traditional temporal semantics, there is always a current state and hence the problem of an empty path does not arise. However, for clocked semantics, there may not be a current state when a clock in the specification stops ticking. For most practical scenario we will assume, henceforth, that clocks in the specifications are always ticking. Under that assumption, the strength operator **!** can be dropped. We first make some crucial observations regarding the rules *R1*, *R4* and *R6*.

As per rule *R1*, *p@clk* holds at the current state, if

a) *p* holds and *clk* ticks at the current state, or
b) *clk* ticks next at a state where *p* also holds.

Similarly, as per rule *R4*, (***X****f)@clk* takes us two *clk* ticks into the future if the *clk* does not hold in the current state. The rule *R6* disallows accumulation of clocks in presence of nesting, allowing only the innermost specified clock to supersede the outer ones.

For example, a clocked LTL formula

$$\boldsymbol{F}(p \wedge (\boldsymbol{X}q@clk1)@clk)$$

gets translated into an equivalent unclocked LTL formula

$$\boldsymbol{F}(p \wedge (\neg clk \boldsymbol{U}(clk \wedge \boldsymbol{X}(\neg clk \boldsymbol{U}(clk \wedge (\neg clk1 \boldsymbol{U}(clk1 \wedge q)))))))$$

The general translation scheme for clocked properties tends to generate large nested LTL formulas that can limit the effectiveness of a standard BMC solver.

Any property specific translation, as discussed later, can get complicated due to subtleties in the clocked specifications, as illustrated in the following example. Consider two clocked specifications, P1 and P2.

P1: $\boldsymbol{F}(ctr2[0] * (\boldsymbol{X}(ctr2[0]))@clk2\_r\_d)$
P2: $\boldsymbol{F}((ctr2[0] * \boldsymbol{X}(ctr2[0]))@clk2\_r\_d)$

In P1, the *ctr2*[0] (bit 0 of *ctr2*) is clocked by the global clock, *gclk*, while in P2, it is clocked by *clk2_r_d* as shown in Figure 2.8. One can verify that the witness state for P1 is at *gclk*=6 where *ctr2*[0]=1 and also two *clk2_r_d* ticks later *ctr2[0]*=1 at *gclk*=13. On the other hand, P2 does not have a witness on the path shown. These subtleties in the clocked specifications add further complexity to the property-specific BMC customization method [46].

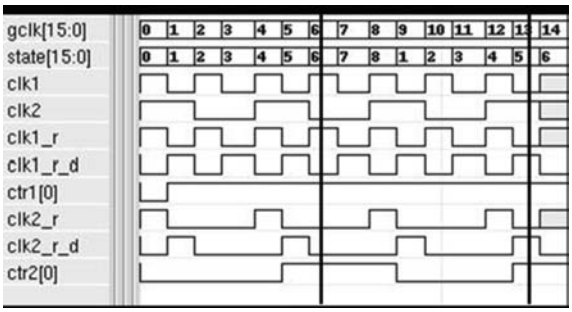

Fig. 2.8: Example: Timing diagram for clocked specification

2.6 SAT-based Unbounded Model Checking

Recall, a standard pre-image computation is given by

$$Pre\text{-}Image(\delta, Y) = \exists_{U,Y}\, \delta(X,Y,U) \wedge T(Y) \tag{2.8}$$

A pre-image computation involves existential quantification of next state variables Y and input variables U. Such an existential quantification can be accomplished using SAT by enumerating all possible solution cubes over X variables in the conjunct $\delta(X,Y,U) \wedge T(Y)$ and adding blocking constraints that exclude previously enumerated solution cubes. A basic algorithm for SAT-based existential quantification based on cube enumeration [68, 71] is shown in Figure 2.9. The procedure *All_sat* takes a function $f(A,B)$ with support variables from sets A and B and returns $C=\exists_B f(A,B)$ after quantifying the variables from the set B. The procedure *SAT_solve* is called repeatedly in line 7 on the constrained problem $f=1 \wedge C=0$ where C represents the set of solutions enumerated so far. Typically, the next call to *SAT_solve* is made from the previous SAT state [47, 69, 71]. The enumerated cube c, in line 9, is obtained by keeping only the A set literals from the assignment cube α in line 8. Successive enumeration of cube c stops in line 7 when C represents $\exists_B f(A,B)$ exactly. The blocking constraints are represented by BDDs [68], zBDDs [71], or CNF [69]. Enlarging the cube c is done by redrawing the implication graph [71] or by applying a justification procedure [121] as a post-processing step. Note that when the procedure *All_sat* is called with substitutions $f \leftarrow \delta(X,Y,U) \wedge T(Y)$, $A \leftarrow X$, and $B \leftarrow U \cup Y$, it returns the *pre-image* of the set $T(Y)$.

```
1.  Synopsis:  SAT-based Quantification (cube enumeration)
2.  Input:     function f, input keep set A,
               input quantifying set B
3.  Output:    ∃B f(A,B)
4.  Procedure: All_sat
5.
6.  C=∅; //initialize constraint
7.  while (SAT_solve(f=1∧C=0)=SAT) {
8.    α=Get_assignment_cube();
9.    c=Get_enumerated_cube(α,B); //obtain, ∃B α
10.   C= C ∨ c;
11. }
12. return C;  // return when no more solution
```

Fig. 2.9: SAT-based existential quantification using cube enumeration

We show a standard least fixed-point computation for temporal logic operator *EF* using the procedure *Fixed_point_EF* in Figure 2.10. $R(X)$ is the set of states that satisfy $\boldsymbol{EF}(f(X))$ and is updated with pre-images in line 8. Note that the exclusion of $R(X)$ in the pre-image computation using *All_sat* as shown in line 10 is equivalent to a typical fixed-point check for $R(X)$.

Note that the number of steps required to reach a fixed-point gives the *longest shortest backward diameter*, a *completeness bound* for BMC [66, 67]. Moreover, one can use the *reachability constraint C(X)* (over-approximated reachable states from initial state) [73], as a *care set* in line 10 (i.e., *All_sat* is called with $f \leftarrow \delta \wedge T \wedge \neg R \wedge C$) to potentially get a fixed-point in less number of pre-image steps than the backward diameter [67].

```
1.  Synopsis:    Least fixed-point using All_sat
2.  Input:       property node f (function of state vars X)
3.  Output:      set of states satisfying EF(f(X))
4.  Procedure:   Fixed_point_EF
5.
6.  i=0; R(X)=∅; T(X)=f(X); //initialize
7.  while (T(X)≠∅) {// fixed-point reached?
8.   R(X)=R(X)∨ T(X); //add i^th pre-image of f
9.   //Compute pre-image state for T but not in R
10.  T(X)=All_sat(δ ∧T(<X←Y>)∧¬R(X), X, U∪Y) ;
11. }
12. return {R(X),i}; // returns states satisfying EF(f(X))
```

Fig. 2.10: Least fixed-point computation using SAT

2.7 SMT-based BMC

In Boolean-level BMC, the translated formula is expressed in propositional logic and a Boolean SAT solver is used for checking satisfiability of the problem. Several state-of-the-art techniques (a survey in [122]) exist for Boolean BMC that have led to its emergence as a mature technology, widely adopted by the industry. However, there are several limitations of a propositional translation and use of a Boolean SAT Solver. Some of these are as follows:

- A propositional translation in the presence of large data-paths leads to a large formula; which is normally detrimental to a SAT-solver due to increased search space.
- Data-path sizes need to be known explicitly *a priori,* before unrolling of the transition relation. For unbounded data-path, additional range-analysis of the program/design is required to obtain conservative but finite data-path sizes.
- High-level information is lost during Boolean translation and therefore, needs to be re-discovered by the Boolean SAT solver, often with a substantial performance penalty.

High-level BMC, as shown in Figure 2.11, overcomes the above limitations of a Boolean-level SAT-based BMC; wherein, a BMC problem is translated typically into a quantifier-free formula in a decidable subset of first order logic, instead of translating it into a propositional formula. A Satisfiability Modulo Theory (SMT) solver is then used to determine the satisfiability of the formula [123-131].

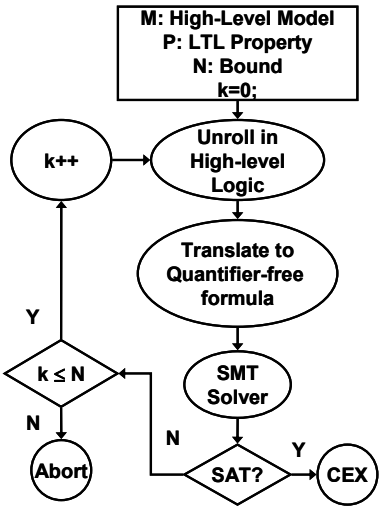

Fig. 2.11: SMT-based BMC

2.8 Notes

We have provided an overview of model checking, bounded model checking, and SAT solvers and their advancements. We require the reader to have a basic understanding of the material described in this chapter. We have also provided many references, which can be followed to get a complete in-depth understanding of the issues and challenges that will be addressed in the subsequent chapters. For ease of understanding, we have attempted to use uniform notation across chapters. Any deviations are either explicitly explained, or are implied from the context.

http://www.springer.com/978-0-387-69166-4

SAT-Based Scalable Formal Verification Solutions

Ganai, M.; Gupta, A.

2007, XXX, 330 p. 118 illus., Hardcover
ISBN: 978-0-387-69166-4