# 3-Valued Circuit SAT for STE with Automatic Refinement

Orna Grumberg, Assaf Schuster, and Avi Yadgar

Computer Science Department, Technion, Haifa, Israel

**Abstract.** Symbolic Trajectory Evaluation (STE) is a powerful technique for hardware model checking. It is based on a 3-valued symbolic simulation, using 0,1 and $X$ ("unknown"), where the $X$ is used to abstract away values of the circuit nodes.
Most STE tools are BDD-based and use a dual rail representation for the three possible values of circuit nodes. SAT-based STE tools typically use two variables for each circuit node, to comply with the dual rail representation.
In this work we present a novel 3-valued Circuit SAT-based algorithm for STE. The STE problem is translated into a Circuit SAT instance. A solution for this instance implies a contradiction between the circuit and the STE assertion. An unSAT instance implies either that the assertion holds, or that the model is too abstract to be verified. In case of a too abstract model, we propose a refinement automatically.
We implemented our 3-Valued Circuit SAT-based STE algorithm and applied it successfully to several STE examples.

## 1 Introduction

Symbolic Trajectory Evaluation (STE) [18] is a powerful model checking technique for hardware verification, which combines symbolic simulation with 3-valued abstraction. Consider a circuit $M$, described as a Directed Acyclic Graph (*DAG*) of nodes that represent gates and latches. For such a circuit, an STE assertion is of the form $A \rightarrow C$, where the *Antecedent A* imposes constraints over nodes of $M$ at different times, and the *Consequent C* imposes requirements on $M$'s nodes at different times.

The antecedent may introduce symbolic Boolean variables, and the assertions it imposes on $M$ depends on them. For each node $n$ and time $t$, STE computes the symbolic representation of $(n, t)$, according to the constraints imposed by $A$, and the behavior of $M$. The nodes that are not restricted by $A$ are initialized by STE to the value $X$ ("unknown"), and thus an $abstraction$ of the checked model is obtained.

For an assertion $A \rightarrow C$ and a circuit $M$, STE may return "pass", "fail", or "unknown" $(X)$ result. If the computed values of all nodes $(n, t)$ comply with the requirements of $C$ for these nodes, then the assertion passes. If, for some requirement of $C$ on $(n, t)$, STE computes a definite value (0 or 1) which contradicts the requirement, then "fail" is returned, together with a counterexample. If, on the other hand, STE computes $X$ for $(n, t)$, though $C$ contains requirements for $(n, t)$, then an "unknown" result is returned. The latter case means that the abstraction induced by $A$ is too coarse, and requires some *refinement*.

STE is successfully used in the hardware industry for verifying very large models with wide data paths [19, 17, 22]. The common method for performing STE is by representing the values of each node in the circuit by *Binary Decision Diagrams (BDDs)* that depend on the symbolic variables [19]. In this method, the *dual rail* representation is used, where two BDDs represent the three possible values of a node. The main drawback of this method is the unpredictability of the BDDs' sizes, and their tendency to explode when a large number of symbolic variables is used. Another limitation in common STE methods is the need for manual refinement, which is time consuming and requires close familiarity with the checked circuit.

For general model checking problems, it has been recognized for quite some time that SAT-based algorithms can often handle much larger models than BDD-based ones. It is therefore very appealing to try and implement SAT-based algorithms for STE as well. However, only a few works took this direction. In [21], *non-canonical Boolean expressions* are used instead of BDDs during the simulation, and a SAT solver is used to check if the resulting expressions meet the requirements of the STE assertion. The Boolean expressions used in this method might be too large to handle, and might require a theorem prover for reducing their size. In [2] and [4], the *dual rail* encoding is used to create a CNF formula for STE. This representation uses two Boolean variables for each node in the circuit, which we avoid in our algorithm. In [15], a 3-valued SAT solver was suggested, which did not perform well. Additionally in [15], an approximation for a 3-valued SAT solver is computed. This approximation, however, does not completely correspond to the semantics of STE. None of the methods discussed above performs automatic refinement. We further elaborate on these works in Section 7.

Particularly interesting for hardware verification is the Circuit-SAT method [8, 11, 10], which gets its input in the form of a circuit rather than a CNF formula. A circuit SAT solver is based on *justification* of nodes, as described in [7]. For a node $n$ in a circuit, and a Boolean value $d$, it searches for a *justification* for $[n, d]$. That is, it looks for a (partial) assignment to some of the circuit inputs, under which $n$ evaluates to $d$.

Our contribution is a novel framework for STE, which is based on a 3-valued justification algorithm. Our algorithm exploits the abstraction induced by using $X$ values, without using the dual rail encoding. It is far less sensitive to the number of symbolic variables than BDD methods. Furthermore, it provides automatic refinement.

For a circuit $M$ and an STE assertion $A \rightarrow C$, we create a circuit that represents $M \wedge A \wedge \neg C$. A justification to the value 1 at the output of the circuit represents a run of $M$ that agrees with the constraints of $A$, and does not satisfy the requirements of $C$. This implies that the STE assertion does not hold on $M$. If no such justification exists, it implies either that $A \rightarrow C$ holds on $M$, or that the abstraction implied by $A$ is too coarse for verifying $A \rightarrow C$. If no justification is found, our algorithm produces a core for the proof of un-justifiability. If this proof does not depend on variables whose values are $X$, then we conclude that $A \rightarrow C$ holds. Otherwise, the core indicates which variables should be refined.

Our algorithm uses a hybrid representation of the problem: as a set of constraints in CNF, and as the *DAG* of the circuit. The CNF representation is used for efficient *Boolean Constraint Propagation* and for learning, as in common SAT solvers [13, 24]. The *DAG* representation is a higher level description of the circuit than the CNF representation. It is used for branching as in [8, 10, 11], for propagating $X$ values, and for deciding termination.

We exploit the fact that for each variable, a Boolean solver holds three possible values, $true$, $false$ and $unspecified$. Thus, we can represent each circuit node by a single variable in the CNF formula. Additional information is used to distinguish between the case the variable has the value $X$ and the case it is unspecified. An $X$ value at a specific node is marked so in the *DAG*. Additionally, it is represented by special constraints added to the CNF formula. New $X$ values can be learnt both on the *DAG* and on the CNF formula. They are used to avoid traversal of abstracted parts of the circuit, thus reducing the amount of work.

We implemented our 3-valued justification algorithm on top of zChaff [13], which is a state of the art CNF SAT solver, and of [9]. We employed our tool for solving several STE problems, and compared it to other methods. Using our algorithm, we managed to solve problems that could not be solved by BDDs, and in most cases it outperformed other SAT based methods. A characterization of such problems is given in Section 6.

The rest of this paper is organized as follows: In Section 2 we present preliminaries. In sections 3 and 4 we describe our justification algorithm, and show how to use it for STE. In Section 5 we explain how automatic refinement can be performed. In Section 6 we present our experimental results, and in Sections 7 and 8 we discuss related work, conclusions, and future research.

## 2 Preliminaries

A hardware model $M$ is a *circuit*, represented by a directed graph. The graph's nodes $\mathcal{N}$ are input and internal nodes, where internal nodes are latches and combinational gates. A combinational gate represents a Boolean operator. The graph of $M$ may contain circles, but not combinational circles. Given a directed edge $(n_1, n_2)$, we say that $n_1$ is an *input* of $n_2$. We denote by $(n, t)$ the value of node $n$ at time $t$. The value of a gate $(n, t)$ is the result of applying its operator on the inputs of $n$ at time $t$. The value of a latch $(n, t)$ is determined by the value of its input at time $t - 1$.

### 2.1 Symbolic Trajectory Evaluation (STE)

In STE, a node can get a value in a quaternary domain $\mathcal{Q} = \{0, 1, X, \perp\}$. $X$(”unknown”) is given to a node whose value cannot be determined by its inputs. $\perp$ is used to describe an over constrained node. This might occur when there is a contradiction between an external assumption on the circuit and its actual behavior.

A *state* $s$ in $M$ is an assignment of values from $\mathcal{Q}$ to every node, $s : \mathcal{N} \to \mathcal{Q}$.

A *trajectory* $\pi$ is an infinite series of states, describing a run of $M$. We denote by $\pi(i), i \in \mathbb{N}$, the state at time $i$ in $\pi$, and by $\pi(i)(n), i \in \mathbb{N}, n \in \mathcal{N}$, the value of node $n$ in the state $\pi(i)$. $\pi^i, i \in \mathbb{N}$, denotes the suffix of $\pi$ starting at time $i$.

Let $\mathcal{V}$ be a set of *symbolic Boolean variables* over the domain $\{0, 1\}$. A *symbolic expression* over $\mathcal{V}$ is an expression consisting of quaternary operations, applied to $\mathcal{V} \cup \mathcal{Q}$. The truth tables of the quaternary operators is given in Figure 1. A *symbolic state* over $\mathcal{V}$ is a mapping from each node of $M$ to a symbolic expression. A symbolic state represents a set of states, one for each assignment to $\mathcal{V}$. A *symbolic trajectory* over $\mathcal{V}$ is a series of symbolic states, compatible with the circuit. It represents a set of trajectories, one for each assignment to $\mathcal{V}$. Given a symbolic trajectory $\pi$ and an assignment $\phi$ to $\mathcal{V}$, $\phi(\pi)$ denotes the trajectory that is received by applying $\phi$ to all of the symbolic expressions in $\pi$.

| AND | $X$ | 0 | 1 | $\perp$ |
|---|---|---|---|---|
| $X$ | $X$ | 0 | $X$ | $\perp$ |
| 0 | 0 | 0 | 0 | $\perp$ |
| 1 | $X$ | 0 | 1 | $\perp$ |
| $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ |

| OR | $X$ | 0 | 1 | $\perp$ |
|---|---|---|---|---|
| $X$ | $X$ | $X$ | 1 | $\perp$ |
| 0 | $X$ | 0 | 1 | $\perp$ |
| 1 | 1 | 1 | 1 | $\perp$ |
| $\perp$ | $\perp$ | $\perp$ | $\perp$ | $\perp$ |

| NOT | |
|---|---|
| $X$ | $X$ |
| 0 | 1 |
| 1 | 0 |
| $\perp$ | $\perp$ |

**Fig. 1.** Quaternary Operations

A *Trajectory Evaluation Logic* (TEL) formula is defined recursively over $\mathcal{V}$ as follows: $f ::= n\ is\ p \mid f_1 \wedge f_2 \mid p \to f \mid \mathbf{N}f$, where $n \in \mathcal{N}$, $p$ is a Boolean expression over $\mathcal{V}$, and $\mathbf{N}$ is the next time operator. The *maximal depth* of a TEL formula $f$ is the maximal time $t$ for which a constraint exists in $f$ on some node $n$, plus 1.

Given a TEL formula $f$ over $\mathcal{V}$, a symbolic trajectory $\pi$ over $\mathcal{V}$, and an assignment $\phi$ to $\mathcal{V}$, we define the satisfaction of $f$ as defined in [20]:
$[\phi, \pi \models f] = \bot \leftrightarrow \exists i \geq 0, n \in \mathcal{N} : \phi(\pi)(i)(n) = \bot$. Otherwise:
$[\phi, \pi \models n \text{ is } p] = 1 \leftrightarrow \phi(\pi)(0)(n) = \phi(p)$
$[\phi, \pi \models n \text{ is } p] = 0 \leftrightarrow \phi(\pi)(0)(n) \neq \phi(p)$ and $\phi(\pi)(0)(n) \in \{0,1\}$
$[\phi, \pi \models n \text{ is } p] = X \leftrightarrow \phi(\pi)(0)(n) = X$ $\quad\phi, \pi \models p \rightarrow f \equiv \neg\phi(p) \vee \phi, \pi \models f$
$\phi, \pi \models f_1 \wedge f_2 \equiv (\phi, \pi \models f_1 \wedge \phi, \pi \models f_2)$ $\quad\phi, \pi \models \mathbf{N}f \equiv \phi, \pi^1 \models f$
Note that given an assignment $\phi$ to $\mathcal{V}$, $\phi(p)$ is a constant (0 or 1).
We define the truth value of $\pi \models f$ as follows:
$[\pi \models f] = 0 \leftrightarrow \exists\phi : [\phi, \pi \models f] = 0$
$[\pi \models f] = X \leftrightarrow \forall\phi : [\phi, \pi \models f] \neq 0$ and $\exists\phi : [\phi, \pi \models f] = X$
$[\pi \models f] = 1 \leftrightarrow \forall\phi : [\phi, \pi \models f] \notin \{0, X\}$ and $\exists\phi : [\phi, \pi \models f] = 1$
$[\pi \models f] = \bot \leftrightarrow \forall\phi : [\phi, \pi \models f] = \bot$

This definition creates an order of importance between $0$ and $X$. If there exists an assignment such that $[\phi, \pi \models f] = 0$, the truth value of $\pi \models f$ is $0$, even if there are other assignments such that $[\phi, \pi \models f] = X$.

| $t$ | $n_1$ | $n_2$ | $n_3$ | $n_9$ | $n_5$ | $n_6$ |
|---|---|---|---|---|---|---|
| 1 | $X$ | $X$ | $X$ | 0 | $X$ | 0 |
| 2 | $X$ | $X$ | 0 | $X$ | 1 | $X$ |

**Fig. 2.** Symbolic Simulation

STE assertions are of the form $A \rightarrow C$, where $A$ (the antecedent) and $C$ (the consequent) are TEL formulae. $A$ expresses constraints on circuit nodes at specific times, and $C$ expresses requirements that should hold on circuit nodes at specific times. $M \models (A \rightarrow C)$ iff for all trajectories $\pi$ of $M$ and assignments $\phi$ to $\mathcal{V}$, $[\phi, \pi \models A] = 1$ implies that $[\phi, \pi \models C] = 1$. When applying $A$ to $M$, if a node $n$ is evaluated to $X$, but is also constrained to a Boolean value 0 or 1 by $A$, then $n$ is assigned with the value imposed by $A$. If $n$ is evaluated to $0(1)$ and $A$ constraints it to $1(0)$, then n is assigned $\bot$. As in [20], an *antecedent failure* is a case where for every trajectory $\pi$ and every assignment $\phi$ to the symbolic variables, there is a node $n$ and time $t$ such that $(n, t)$ is over constrained by $\pi, \phi$ and $A$. Consider the circuit in Figure 3(a), and the STE assertion $A \rightarrow C$, where $A = (n_4, 1)\ is\ 0$ and $C = (n_5, 2)\ is\ 1$. The table in Figure 2 corresponds to a symbolic simulation of this assertion. $(n_5, 1)$ is evaluated to 1, and thus the assertion holds.

Most STE implementations use the *dual rail* encoding in order to represent the 4 values. In this encoding, the value of each node $(n, t)$ is determined by the evaluations of two Boolean functions $f^1_{n,t}, f^2_{n,t} : \mathcal{V} \rightarrow \{0, 1\}$ over the set of symbolic variables $\mathcal{V}$.

## 2.2 The SAT Problem

The *Boolean satisfiability problem* (SAT) is the problem of finding an assignment $\phi$ to a set of Boolean variables $V$ such that a Boolean formula $\varphi(V)$ will have the value '1' under this assignment. $\phi$ is called a *satisfying assignment* for $\varphi$.

We discuss formulae presented in the conjunctive normal form (CNF). That is, $\varphi$ is a conjunction of clauses, where each clause is a disjunction of literals over $V$. A literal $l$ is an instance of a variable or its negation: $l \in \{v, \neg v \mid v \in V\}$. We shall consider a clause as a set of literals, and a formula as a set of clauses.

A clause $cl$ is satisfied under an assignment $\phi$ iff $\exists l \in cl, \phi(l) = 1$. For a formula $\varphi$ given in CNF, an assignment satisfies $\varphi$ iff it satisfies all of its clauses. Hence, if, under an assignment $\phi$ (or a partial assignment), all of the literals of some clause in $\varphi$ are 0, than $\phi$ does not satisfy $\varphi$. We call this situation a *conflict*.

For an unsatisfiable formula $\varphi = \mathcal{C}$, where $\mathcal{C}$ is a set of clauses, an *unSAT core* $\mathcal{C}'$ is a set of clauses $\mathcal{C}' \subseteq \mathcal{C}$ such that $\mathcal{C}'$ is unSAT.

For two clauses $cl_1 = (w_1, v_1 \dots v_n)$ and $cl_2 = (\neg w_1, z_1 \dots z_m)$ ($(v_1 \dots v_n)$ and $(z_1 \dots z_m)$ are not necessarily disjoint), their *resolvent* is $cl_{res} = (v_1 \dots v_n) \cup (z_1 \dots z_m)$. It is easy to show that $cl_1 \wedge cl_2 \wedge cl_{res} \equiv cl_1 \wedge cl_2$. For an unSAT formula, there exists a series of resolutions that leads to the empty clause. This series is the proof of the formula's unsatisfiability. This series is called *resolution tree*, where the root is the empty clause, and the rest of the nodes are the clauses in the series that led to it. The antecedents of a node are the clauses that were involved in the resolution that create it. The leaves are a subset of the original clauses of the formula. This subset of clauses is an unSAT core.

**Davis-Putnam-Logemann-Loveland Backtrack Search (DPLL)** We begin by describing the *Boolean Constraint Propagation* (*bcp*). Given a partial assignment $\phi$ and a clause $cl$, if there is one literal $l \in cl$ with no value, while the rest of the literals are all 0, then in order to avoid a conflict, $\phi$ must be extended such that $\phi(l) = 1$. $cl$ is called a *unit clause*, and the assignment to $l$ is called an *implication*. *bcp* computes all possible implications at a given moment. This procedure is efficiently implemented in [13, 23, 12].

The DPLL algorithm [6, 5] iteratively chooses an assignment to some variable, and computes its implications. If no conflict occurs, a new assignment is chosen, and so on. If a conflict occurs, the algorithm invalidates the last chosen assignment, and tries another one instead. Choosing an assignment to a variable is called *branching*, and invalidating a decision is called *backtracking*. DPLL terminates in the following cases: If all of the variables are assigned without causing a conflict, $\varphi$ is satisfiable, and the current assignment to the variables is a satisfying assignment. On the other hand, if a conflict occurs but there are no decisions to invalidate, it is concluded that $\varphi$ is unsatisfiable.

**Optimizing DPLL**

Modern SAT solvers apply several optimization on the basic DPLL backtrack search. Such optimizations are conflict based learning, conflict driven backtracking, restarts and more. These optimizations result in a significant speedup of the SAT solving tools.

Learning is performed upon the occurrence of a conflict. At this point, two clauses implicate conflicting values to the same variable. The resolution of the clauses describes the cause to the conflict. Thus, it is added to the formula, and prevents the conflict from reoccurring. Different learning strategies yield different conflict clauses. *1UIP* is a common and very efficient strategy[24].

## 2.3 Bounded Model Checking

We shall briefly describe Bounded Model Checking (BMC)[1] for a model $M$ and a property $P$, which is a commonly used model checking technique. In BMC, the transition relation of $M$ is described as a Boolean formula $R(\overline{x}, \overline{x}')$, where $\overline{x}$ and $\overline{x}'$ are the current and next state variables, respectively. The property $P$ is also described as a Boolean formula $P(\overline{x})$. Additionally, the set of initial states of $M$ is described by a boolean formula $I_0(\overline{x})$.

BMC is an iterative algorithm. At iteration $k$, the formula $\varphi_k = I_0(\overline{x_0}) \bigwedge_{i=0}^{k-1} R(\overline{x_i}, \overline{x_{i+1}}) \wedge \neg P(\overline{x_k})$ is constructed, and is given to a SAT solver. A solution for $\varphi_k$ represents a path of length $k$ from an initial state in $M$, along which the property $P$ does not hold. Thus, a solution represents a bug in the model. If $\varphi_k$ is unSAT, then no such path of length $k$ exists, and the algorithm continues to the next iteration.

If $P$ describes only finite paths, BMC terminates when $k$ reaches the length of the longest path in $P$. Otherwise, BMC terminates when $k$ reaches the diameter of $M$. In practice, the diameter of the model is not reached, and BMC stops due to memory limits or timeouts.

### 2.4 Circuit SAT Solvers

**Justification of Assignments** For a circuit node $n$ and value $d$, we say that $[n, d]$ is *justified* by the inputs to $n$ if $d$ is implied by them according to the semantics of $n$. On that case, we say that $n$ is justified by its inputs. For example, consider a node $n$, associated with an "AND" operator, and its inputs $in_1 \dots in_m$. $[n, 0]$ is justified iff $\exists\ i\ s.t.\ in_i = 0$, regardless of the values of the rest of the inputs. $[n, 1]$ is justified iff $\forall\ i,\ in_i = 1$. We generalize this definition for the set of nodes in the graph that may effect the value of $n$. When given a (partial) assignment to the inputs of a circuit, we say that $[n, d]$ is justified if $d$ is implied by those inputs. An input is thus trivially justified. Throughout the rest of this paper, an *assignment* is considered a *partial assignment*.

**Circuit SAT** A *Circuit SAT Solver* [8, 11, 10] is a solver that uses a graph representation of the circuit instead of a CNF formula. Given a circuit, a node $n$ and a value $d$. A circuit SAT solver returns a justification for $[n, d]$ if one exists, or "unjustifiable" otherwise. Branching, *bcp*, learning and other procedures are performed over the graph.

## 3 3-Valued Justification

In this section we describe our 3-valued algorithm for justifying a value of a node in a circuit. Our algorithm uses a dual representation of the circuit. The first is a graph $G$ of the circuit's gates and latches, and the other is a CNF description of it, denoted $\varphi$. $\varphi$ is built as described in [1]. $\psi^1_{and}$ in Figure 6 is an example for a CNF description of an "AND" gate $n$, with inputs $in_1$ and $in_2$. There is a 1-1 mapping between the variables of $\varphi$ and the nodes of $G$. Thus, we can refer to a node by its corresponding variable and vice versa. The graph and the CNF representations are maintained throughout the computation in order to keep the correlation between them. The pseudo code of our algorithm is given in Figure 4. Throughout this Section we refer to the example of circuit $t_1$ in Figure 3(b).

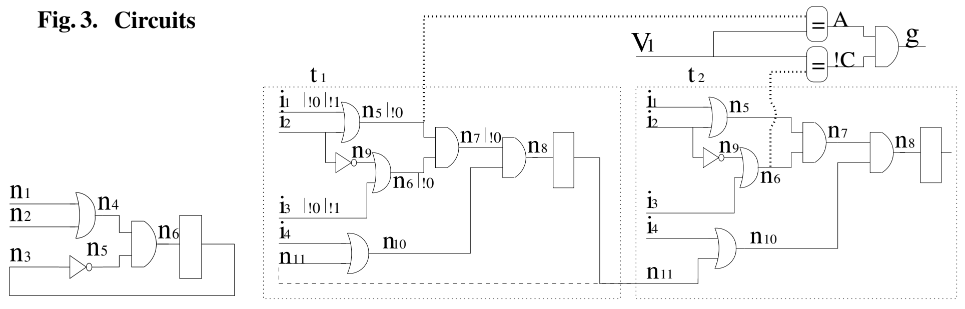
### 3.1 *not-0* and *not-1* Variables

When working in a 3-valued domain, a variable being *not-1* does not imply being 0, and vice versa. Therefore, we introduce the notions of *not-0* and *not-1*. A variables is *not-0* or *not-1* if it is not allowed to be assigned 0 or 1, respectively. Consequently, a node which is both *not-0* and *not-1* can only be assigned $X$. Such restrictions can be derived from external constraints, or learned during the search. We denote *not-0* and *not-1* by $|_{!0}$ and $|_{!1}$, respectively.

In the graph representation $G$ we have a mechanism for marking $|_{!0}$ and $|_{!1}$ nodes. We need a mechanism for marking and manipulating $|_{!0}$ and $|_{!1}$ variables in $\varphi$. Therefore, we do not consider the clauses to be sets of literals, as defined in Section 2.2. Instead, we consider the clauses to be *multi-sets* of literals. The definition of a conflict and constraint propagation remain as in Section 2.2. $|_{!0}$ and $|_{!1}$ variables are marked by adding the clauses $(n \vee n)$ and $(\neg n \vee \neg n)$, respectively. When applying constraint propagation, each of these clauses causes a conflict if we try to assign $n$ with a value 0 or 1, respectively. However, since they never become *unit clauses*, neither of the clauses forces any value on $n$. In the Boolean domain, the propositional formula $(n \vee n) \wedge (\neg n \vee \neg n)$ is not *satisfiable*. In contrast, in our algorithm, these clauses correctly represent $X$ values, since our algorithm does not necessarily *satisfy* all the clauses, as we describe in Section 3.2. In particular, our algorithm does not assign a value to a variable that is both $|_{!0}$ and $|_{!1}$. Note that though a variable may have multiple instances in a clause, we only have to distinguish between single and multiple instances. Thus, if a variable has more than one instance in a clause, we only keep two instances.

$|_{!0}$ and $|_{!1}$ restrictions are propagated on $G$. Consider $n_5$ in the example. $i_1$ is $|_{!0}$. Therefore, $n_5$ cannot be assigned 0, and is also $|_{!0}$. Similarly, $n_6$ is also $|_{!0}$. In addition, since all the inputs to $n_7$ are $|_{!0}$, $n_7$ is also $|_{!0}$. We do not propagate the restrictions directly on $\varphi$. However, when propagating them on $G$, we also create the appropriate clauses for the implied restrictions, and add them to $\varphi$. For example, $i_1$ is $|_{!0}$ implies that $n_5$ is $|_{!0}$. Thus we add the clause $(n_5, n_5)$. Additionally, $i_1$ being $|_{!1}$ changes the relation between $i_2$ and $n_5$: Since $i_1$ is $|_{!1}$, $n_5 = 1$ implies $i_2 = 1$. This new relation is expressed by the clause $(i_2, \neg n_5, \neg n_5)$. Note that $i_2$ is only one of the inputs to an "OR" gate, and therefore $i_2 = 0$ *should not* imply $n_5 = 0$. The two instances of $\neg n_5$ prevent that. Similarly, the clause $(n_9, \neg n_6, \neg n_6)$ is created.

**Fig. 3. Circuits**



(a) A circuit (b) An Unrolled Circuit

In section 2.2 we defined the *resolution tree* for clauses that are created by resolution. In our context, clauses can be created by propagating $|_{!0}$ and $|_{!1}$ on $G$. The propagation on $G$ corresponds to the semantics of the nodes, which is also expressed by the clauses of the nodes in $\varphi$. Thus, the generated clauses are considered as the result of applying resolution on the relevant clauses in $\varphi$. In the example, the clause $(n_5, n_5)$ can be created by applying resolution on the clauses $(i_1, i_1)$ and $(\neg i_1, n_5)$, and on their resolvent and $(\neg i_1, n_5)$ again. The definition of the resolution tree thus remains unchanged.

## 3.2 3-Valued Justification Algorithm

Given a *DAG* $G$ of a circuit, a CNF description of it $\varphi$, a node $r \in G$, and a Boolean value $d$, our 3-valued justification algorithm (3VJA) returns a justifying assignment for $[r, d]$, or *unjustifiable* if $[r, d]$ is not justifiable. We call $r$ the *root* of $G$. 3VJA performs an iterative backtrack search over $G$. The information in $G$ about the structure of the model is used for branching during the search, and allows propagation of $|_{!0}$ and $|_{!1}$ restrictions. It is also used for correct termination of the algorithm. The CNF representation $\varphi$ is used for efficient constraint propagation, detection of conflicts and for learning. $X$ values are described by using clauses representing the $|_{!0}$ and $|_{!1}$ constraints, and can be learned during the solving process. Next we describe and explain 3VJA.

We begin by describing the branching procedure, which is a 3-valued variation of the justification procedure described in [7]. Our branching procedure traverses $G$, assigning the nodes with values in a pre-order manner, starting from the root. For each node it chooses values only to its inputs that are needed in order to justify it. The rest of the input nodes are not assigned and are not traversed. The branching procedure does not assign $|_{!0}$ and $|_{!1}$ nodes with the values 0 and 1, respectively. In the example, justification of $[n_8, 0]$ will not be done by assigning $n_7 = 0$. If it is impossible to justify a node with any of its inputs, 3VJA invalidates the last branching and tries another path. The root of $G$ is assigned either 1 or 0. Therefore, we never justify an $X$ value, nor do we have to assign a node with the value $X$ for justification of a node.

```
3VGA (G,φ,n,d)
1)  while true
2)    if (¬branch on G)
3)      return justification
4)    if (bcp on φ ⇒conflict){
5)      learn conflict clause
6)      if learned X clause {
7)        mark X on G
8)        propagate X on G
9)        add clauses to φ
10)     }
11)     if possible
12)       backtrack
13)     else
14)       return unjustifiable
15) }
```

**Fig. 4.** 3VJA. Lines 2, 7 and 8 are executed on $G$. Lines 4, 5 and 9 are executed on $\varphi$.

After each branching, the assigned value has to be propagated through the variables. We exploit the fact that a value of a variable in a Boolean SAT solver can be either 1, 0, or $unassigned$ in order to represent 3 values in a Boolean context. Thus, we use $\varphi$ to propagate the branching assignment through the circuit. The propagation and the definition of a *conflict* remain as defined for Boolean SAT. If the propagation does not cause a conflict, 3VJA continues to the next iteration. If a conflict occurs, 3VJA learns a new conflict clause, and backtracks accordingly.

When a conflict occurs, the resolvent of the clauses that were involved in the conflict is added to the problem. In the 3-valued context, we define the resolvent of clauses $cl_1 = (w_1, v_1 \dots v_n)$ and $cl_2 = (\neg w_1, z_1 \dots z_m)$ to be $cl^3_{res} = (v_1 \dots v_n, z_1 \dots z_m)$. Note that the clauses are considered to be *multi-sets*, and clauses may have multiple instances of a variable. For example, the resolvent of $(v_1, v_2, v_3)$ and $(\neg v_1, v_3, v_4)$ is $(v_2, v_3, v_3, v_4)$. Adding a resolvent as defined above to $\varphi$ does not change the set of justifying assignments to $[r, d]$. Due to space limitations, we omit the proof of this claim. This resolution is similar to the resolution described in [15]. We elaborate on this in Section 7.

It is possible to learn conflict clauses such as $(n, n)$ and $(\neg n, \neg n)$. When learning such clauses, we mark the corresponding nodes in $G$ as $|_{!0}$ and $|_{!1}$, respectively. We propagate this information on $G$, thus extracting additional information from the learned conflict clause. We then generate the appropriate clauses, as described in Section 3.1, and thus maintain the correlation between $G$ and $\varphi$.

By learning $(n, n)$ and $(\neg n, \neg n)$ clauses we can conclude that a node is forced to $X$ even if such a conclusion can not be explicitly derived from $G$. This is an important result, because it prevents the branching procedure from trying to use the constrained node for justification in the future. It also helps detecting conflicts earlier. We demonstrate this on our example. Assume that the branching procedure assigned $n_8 = 1$. A possible series of implications from this assignment is $n_7 = 1, n_5 = 1, n_6 = 1, n_9 = 1, i_2 = 0$. Other series could be computed, depending on the order of computing the implications. The result of these implication is that all the literals in the clause $(i_2, \neg n_5, \neg n_5)$ are 0. That is, a conflict has occurred. We show the series of resolutions that is performed upon this occurrence in

Figure 5. The learned conflict clause is $(\neg n_7, \neg n_7)$, and it is added to $\varphi$. We also mark that $n_7$ is $|_{!1}$ in $G$ and propagate it, implying $n_8$ is $|_{!1}$ and $n_{11}$ is $|_{!1}$. These implications are also added as the clauses $(\neg n_8, \neg n_8)$ and $(\neg n_{11}, \neg n_{11})$ to $\varphi$. The result is that $n_7$ is assigned $X$, and $n_8$ and $n_{11}$ are $|_{!1}$.

| | | |
|---|---|---|
| 1. $(i_2, \neg n_5, \neg n_5)$ | 5. $(\neg n_7, n_5)$ | 9. $(8 \uplus 4) \uplus 4 = (\neg n_7, \neg n_7, \neg n_5, \neg n_5)$ |
| 2. $(\neg n_9, \neg i_2)$ | 6. $(\neg n_8, n_7)$ | 10. $(9 \uplus 5) \uplus 5 = (\neg n_7, \neg n_7)$ |
| 3. $(n_9, \neg n_6, \neg n_6)$ | 7. $1 \uplus 2 = (\neg n_9, \neg n_5, \neg n_5)$ | 11. $(\neg n_8, \neg n_8)$ |
| 4. $(\neg n_7, n_6)$ | 8. $7 \uplus 3 = (\neg n_5, \neg n_5, \neg n_6, \neg n_6)$ | 12. $(\neg n_{11}, \neg n_{11})$ |

**Fig. 5.** Learning $X$ clauses. $\uplus$ denotes the *resolution* operation. Refer to the circuit $t_1$ in Figure 3(b). $n_8 = 1$, implies $n_7 = 1, n_5 = 1, n_6 = 1, n_9 = 1, i_2 = 0$, by using clauses $1 - 6$. Clauses $1, 3$ originate from propagating $|_{!1}$ for $i_1$ and $i_3$, respectively, as described in Section 3.1. Clause 2 is a part of the description of the "NOT" gate. Clauses $4, 5$ and 6 are the relevant clauses of the "AND" and "OR" gates. Clauses $7 - 10$ are created by applying *resolution* on the original clauses. Clause 10 is the conflict clause derived by the *1UIP* strategy. Having $n_7$ is $|_{!1}$ on $G$ implies that $n_8$ and $n_{11}$ are $|_{!1}$, and thus clauses 11 and 12 are created.

Note that unlike implications computed by constraint propagation, nodes that are assigned $X$ remain $X$ throughout the solving process, and are not effected by backtracking. This is because the conclusion about $X$ nodes is derived from the problem itself, regardless of the current partial assignment. Therefore, a mechanism for invalidating $X$ assignments is not required.

A justifying assignment for $[r, d]$ is found when we complete the traversal of $G$. This traversal does not necessarily include all the nodes in $G$, but rather only the nodes that were required for this justification. Alternatively, if the traversal can not be completed, we conclude that $[r, d]$ can not be justified.

## 4 STE with 3-Valued Justification

In this section we show how to employ our 3-valued justification algorithm for STE. We start by describing the construction of circuits to represent an STE problem, and then show how to use the algorithm from Section 3 for solving it.

### 4.1 Constructing Circuits for STE Assertions

Consider a model circuit $M$, and an STE assertion $A \to C$. $A$ and $C$ are given in $TEL$, as described in Section 2.1. In order to prove or falsify the assertion, $M$ has to be simulated $k$ times, where $k$ is the *maximal depth* of $A$ and $C$.

We create a new graph by unrolling $M$ $k$ times. Each node $n \in M$ has $k$ instances in the new graph. The $i^{th}$ instance of node $n$ represents node $n$ at time $i$. In the new graph, the connectivity of the input and gate nodes remains the same. The latches are connected such that the input to a latch at time $t$ are the nodes at time $t - 1$, and the latch is an input to nodes at time $t$. Due to the new connectivity of the latches, and since $M$ does not have combinational circuits, the unrolled graph is a *DAG*. The inputs to the new graph are $k$ instances of each of the inputs to the circuit. In Figure 3(b), an unrolling of a circuit is presented. $t_1$ and $t_2$ are two instances of the circuit. The inputs to the latch $n_{11}$ in $t_2$ are the nodes of $t_1$, thus eliminating the circle in $t_1$. The inputs to the new circuit are the two instances of $i_1 - i_4$. From herein we denote by $M$ the unrolled graph of the circuit.

As mentioned before, $A$ and $C$ are given in TEL. Therefore, we can construct combinational circuits that represent them. The inputs to these circuits are nodes in $M$, and new constructed nodes that represent the symbolic variables of the STE assertion. The output of each circuit, denoted the *root* of the circuit, equals to the evaluation of the corresponding TEL formula. For example, for a TEL formula $A = (n, i)\ is\ V_1$, the input to the circuit of $A$ is the $i^{th}$ instance of the circuit node $n$ in $M$, and a node associated with the symbolic variable $V_1$. The root of the circuit is 1 if the input values are equal, and 0 otherwise. The construction of circuits for $n\ is\ p$, $f_1 \wedge f_2$ and $p \rightarrow f$ are trivial. The circuit for $f = Nf'$ is derived by constructing the circuit for $f'$, and replacing each of its input nodes $(n, t)$ by the node $(n, t + 1)$. Note that each symbolic variable has only one instance. Also note, that the constructed circuits for $A$ and $C$ are also *DAG*s. From herein we denote by $A$ and $C$ the corresponding circuits, respectively. Also, we refer to a node $n$ at time $i$ by the name of the $i^{th}$ instance of $n$ in $M$, instead of by $(n, i)$.

We construct a circuit for $M \wedge A$ by connecting the relevant nodes in $M$ to the inputs of $A$. The inputs to $M \wedge A$ are the $k$ instances of the inputs to the hardware model, and the symbolic variables defined in $A$. The assertions that are imposed by $A$ are in fact assumptions on the circuit. As defined in Section 2.1, a node $n$ is assigned the Boolean value imposed on it by $A$, even if its evaluation on the circuit is $X$. In our algorithm, this means that the values of $n$ do not have to be justified, and should not propagate from $n$ to its inputs. We mark asserted nodes in the graph, such that $X$ values do not propagate through them, and the branching procedure considers them justified, not trying to assign their inputs. Additionally, we construct the CNF clauses for an asserted node such that they allow forward propagation only. This is demonstrated in Figure 6. For a node $n = in_1 \wedge in_2$, we create $\psi^2_{and}$ instead of $\psi^1_{and}$. For example, if $n$ is assigned the value 1 by $A$, none of the clauses propagates this value to $in_1$ and $in_2$. On the other hand, forward propagation is still implied.

$$\psi^1_{and} = (n, \neg in_1, \neg in_2) \wedge (\neg n, in_1) \wedge (\neg n, in_2)$$
$$\psi^2_{and} = (n, \neg in_1, \neg in_1, \neg in_2, \neg in_2) \wedge (\neg n, in_1, in_1) \wedge (\neg n, in_2, in_2)$$

**Fig. 6.** A CNF representation of an "AND" gate $n = in_1 \wedge in_2$. $\psi^1_{and}$ propagates implications in both directions. $\psi^2_{and}$ propagates implications only forwards.

We construct the circuit $\Gamma = M \wedge A \wedge \neg C$ by connecting the relevant nodes in $M \wedge A$ to the inputs of $C$. As with $M \wedge A$, the inputs to the new circuit are the inputs to $M$ and the symbolic variables. We create a new "AND" node such that its inputs are the roots of $A$ and $\neg C$. This node is considered the root of $\Gamma$. An example for such a construction is given in Figure 3(b). The node associated with "=" represents a combinational circuit that evaluate to 1 if the values in the inputs are equal, and 0 otherwise. Consider an assertion $A \rightarrow C$ such that $A = (n_5, 1)\ is\ V_1$, and $C = (n_6, 2)\ is\ \neg V_1$. $t_1$ and $t_2$ are the unrolled circuit. The node $A$ is the root of the circuit that corresponds to $A$. The inputs to this circuit are $(n_5, 1)$ and $V_1$. $!C$ is the root of the circuit that corresponds to $\neg C$. The inputs to this circuit are $(n_6, 2)$ and $V_1$. The node $g$ evaluates to $\Gamma$.

## 4.2 Running STE

We first verify that $A$ does not cause an *antecedent failure* with $M$. Therefore, we have to verify that there is at least one run of the model that does not conflict with the assertions from $A$. Consider the circuit $M \wedge A$, described in the previous section. We apply 3VJA for justifying $[a, 1]$, where $a$ is the root of $A$. A justifying assignment for this problem represents a run of $M$ that satisfies the constraints imposed by $A$. Therefore, if such an assignment is found, we conclude that there is no antecedent failure. If the problem is unjustifiable, then no such run exists, which means an antecedent failure.

Assuming no antecedent failure was found,we apply 3VJA on $[\gamma, 1]$, where $\gamma$ is the root of $\Gamma$, defined in the previous section.

If a justifying assignment is found, it represents an assignment to the inputs of $\Gamma$ that makes $\gamma$ evaluate to 1. This assignment represents a run of $M$ that satisfies the constraints imposed by $A$, but contradicts the requirements of $C$. Such an assignment means that the STE assertion $A \to C$ does not hold in $M$.

If $[\gamma, 1]$ is unjustifiable, an empty clause was learned. We extract the unSAT core from the resolution tree of the empty clause, and check if it contains clauses for $|_{!0}$ or $|_{!1}$ nodes, that originate from $A$. If there are no such clauses in the core, then no $X$ value has participated in proving the unjustifiability of $[\gamma, 1]$. Therefore, we conclude that there is no run of $M$ that complies with the restrictions of $A$, but does not satisfy the requirements of $C$. That is, the STE assertion $A \to C$ holds in $M$. On the other hand, if the unSAT core includes clauses for $|_{!0}$ or $|_{!1}$ nodes that originate from $A$, then the proof for unjustifiability depends on $X$ values. In that case, it might be that we did not find a counter example for $A \to C$ due to a too coarse abstraction. Therefore, we have to refine the model in order to prove or falsify the STE assertion.

Note that in case of an unjustifiability proof that depends on $X$ values from $A$, another proof that does not depend on $X$ values might exist. Therefore, it might be possible to prove the STE assertion without refining the model. We could avoid this by changing the justification and traversing larger portions of the circuit. We then have a trade off between light-weight justification with more refinements, and heavy-weight justification. In our current algorithm, we choose to perform the light-weight justification and refine the model if needed. We discuss refinement in Section 5.

## 5 Refinement

A major strength of STE is the use of abstraction. The abstraction is determined by assigning nodes in the model $M$ with the value $X$ by $A$, the antecedent of the STE assertion. However, if the abstraction is too coarse, there is not enough information for proving or falsifying the STE assertion. We present a "CEGAR" [3] approach for refining such assertions.

For an unjustifiable instance given to 3VJA, the resolution tree, derived for it, is the proof that the instance is unjustifiable. We define a *spurious* proof to be a resolution tree such that the unSAT core defined by it includes clauses for $X$ nodes, that originate from the antecedent $A$ of the STE assertion.

In our STE method, a too coarse abstraction results in an unjustifiable instance with a spurious proof of unjustifiability. By associating the $X$ nodes in the unSAT core with symbolic variables, we refine the model and invalidate the current spurious proof.

Refining only $X$ nodes in the unSAT core, only the variables needed to eliminate the spurious proof are refined. This means that for an $X$ node $n$, we only add variables for $X$ nodes that took part in implying $X$ on $n$, rather than all the $X$ nodes in the cone of influence of $n$. Refer to the example in Figure 3(b), and consider $A$ that assigns, at $t_1$, $X$ to $i_1, i_3$ and $i_4$, and 1 to $n_{11}$. This implies $n_{10} = 1$, and $n_8$ is $|_{!0}$. When trying to justify $n_8 = 1$, as seen in Section 3.2, we learn that $n_7 = X$, and $n_8$ is $|_{!1}$. Therefore, $n_8 = X$. Note that the conclusion that $n_8$ is $|_{!1}$ is independent of $i_2, i_4$ and $n_{10}$. If $n_8 = X$ takes part in the proof that the whole circuit is unjustifiable, $i_1$ and $i_3$ will be in the unSAT core, while $i_2$ and $i_4$ will not. Thus, when refining, we will not add a variables for $i_2$ and $i_4$.

Our refinement eliminates the proof of unjustifiability that was found. Running the justification algorithm again, we either find another spurious proof that has to be refined, a concrete proof of unjustifiability, or proof of justifiability (a justifying assignment).

## 6 Experimental Results

For evaluating our justification algorithm 3VJA, presented in Section 3.2, we implemented it on top of zChaff [13], a state of the art SAT solver, and [9], and used it for STE, as described in Section 4. For comparison, we used the dual rail encoding for solving SAT based STE [15], and *Forte*, a BDD based STE tool by Intel [19]. Additionally, we used BMC for solving the benchmarks, considering the STE assertions as an LTL formulae. For the SAT based STE and for BMC, we used the same SAT solver zChaff, on top of which we implemented our algorithm. All experiments use dedicated computers with 3.2Ghz Intel Pentium CPU, and 3GB RAM, running Linux operating system. Time out was set to one hour.

For our experiments we used the *Memory* and *CAM* circuits from Intel's GSTE tutorial, which are large enough to demonstrate various characteristics of the algorithm. The *Content addressable Memory* (CAM) has 16 entries, 64 bits data width, and 8 bits tag width. The memory circuit has a 6 bits address width and 128 bits data width.

The results of our experiments are presented in Table 6. We verified the *associative read* property of the CAM by using "full", "plain" and "cam" symbolic indexing schemes, as defined in [14]. Additionally, we checked the CAM and the memory against series of multiple write and read operations. Each assertion has a different set of symbolic variables and a different depth. Assertions $1 - 14$ were verified, whereas assertions $15 - 25$ were falsified. Columns 3V, BDD, DR and BMC present the solving time of our 3VJA based STE, BDD based STE, Dual Rail SAT based STE, and BMC, respectively.

3VJA has outperformed the *BDD based algorithm* on most of the assertions, especially the harder ones. Compared to the BDD algorithm, 3VJA is far less sensitive to the number of symbolic variables. Consider assertions $1-3$ and $4-6$. These assertions are different encodings for the associative read operation of CAM, defined for depth 2 and 6, respectively. Each encoding of the assertion requires a different number of symbolic variables. On both depths, the BDD algorithm timed out for "full" and "plain" encodings, while 3VJA solved the problems in seconds. On the other hand, 3VJA is more sensitive to the number of nodes in the circuit, and thus to the depth of the assertions, than BDDs. This is also a characteristic of the other SAT based algorithms, and is demonstrated by assertions $4$ and $10 - 11$, relatively to $1$ and $9$, respectively. In each of these cases, a similar assertion is checked to different depths. The number of symbolic variables is about the same, but the number of nodes in the circuit grows. This affects the SAT based algorithms more than it affects the BDD based algorithm. Note, however, that in case of a failed assertion with many symbolic variables, the BDD method may fail due to the need to compute the values of all nodes up to the depth of the contradiction, while a SAT based algorithm only has to find one erroneous path. This is demonstrated by assertions $15, 16$ and $21 - 25$.

We see that *BMC* outperforms the *dual rail* method in most of the cases, especially for verification. The dual rail representation uses two Boolean variables to represent each node. The result is a very large SAT instance, which is harder to solve. This result matches the results in [15].

3VJA outperforms BMC in most cases, especially in falsification. While not very sensitive to the number of symbolic variables, BMC does not use $X$ values, and thus does not use an abstraction. This makes BMC more sensitive to the width of data paths, and the depth of the assertions. For verification, we expected 3VJA to return “unjustifiable” faster than BMC, since the justification is constrained by the $X$ nodes. However, in a few cases, such as 10, we had to refine the model multiple times until a concrete proof for unjustifiability was found. In these experiments, refinement was performed manually. In 11, we could not find such a proof within the time limit. For falsification, we see a clear advantage to 3VJA. This can be explained by the fact that 3VJA does not try to assign values to $X$ nodes, and thus does not traverse large portions of the circuits. This advantage increases with the number of nodes that are abstracted out by the STE assertion, and is demonstrated by assertions $17-25$.

| Verification | | | | | Time (s) | | | |
|---|---|---|---|---|---|---|---|---|
| | Assertion | D | # vars | #N x$10^3$ | 3V | BDD | DR | BMC |
| 1 | CAM cam | 2 | 124 | 5 | 4 | 0.5 | 5 | 1 |
| 2 | CAM plain | 2 | 204 | 5 | 2 | T.O | 1 | 1 |
| 3 | CAM full | 2 | 1160 | 5 | 1 | T.O | 1 | 1 |
| 4 | CAM cam | 6 | 128 | 15 | 31 | 1 | 94 | 87 |
| 5 | CAM plain | 6 | 208 | 15 | 15 | T.O | 27 | 30 |
| 6 | CAM full | 6 | 1164 | 15 | 14 | T.O | 26 | 34 |
| 7 | CAM 1 | 10 | 152 | 25 | 349 | 5 | 513 | 493 |
| 8 | CAM 2 | 10 | 242 | 25 | 45 | T.O | 537 | 473 |
| 9 | Mem 1 | 2 | 86 | 110 | 5 | 1 | 9 | 2 |
| 10 | Mem 1 | 5 | 104 | 260 | 773 | 3 | 413 | 320 |
| 11 | Mem 1 | 11 | 164 | 550 | T.O. | 9 | T.O | T.O |
| 13 | Mem 2 | 5 | 304 | 260 | 54 | 455 | 72 | 52 |
| 14 | Mem 2 | 11 | 334 | 550 | 77 | 523 | 142 | 81 |

| Falsification | | | | | Time (s) | | | |
|---|---|---|---|---|---|---|---|---|
| | Assertion | D | # Vars | #N x$10^3$ | 3V | BDD | DR | BMC |
| 15 | CAM 3 | 4 | 320 | 10 | 10 | 437 | 5 | 1 |
| 16 | CAM 4 | 4 | 260 | 10 | 14 | 209 | 19 | 13 |
| 16 | CAM 5 | 5 | 72 | 10 | 32 | 3 | 12 | 3 |
| 17 | Mem 3 | 2 | 134 | 110 | 280 | 282 | 832 | 327 |
| 18 | Mem 3 | 5 | 134 | 260 | 536 | 436 | T.O | 2753 |
| 19 | Mem 3 | 10 | 134 | 550 | 1943 | 641 | T.O | T.O |
| 20 | Mem 3 | 15 | 134 | 770 | T.O | 943 | T.O | T.O |
| 21 | Mem 4 | 5 | 168 | 260 | 536 | T.O | 343 | 2854 |
| 22 | Mem 4 | 10 | 168 | 550 | 1765 | T.O | 2248 | 3004 |
| 23 | Mem 4 | 15 | 168 | 770 | 2064 | T.O | 3440 | T.O |
| 24 | Mem 5 | 10 | 670 | 550 | 3276 | T.O | 3555 | T.O |
| 25 | Mem 5 | 15 | 670 | 770 | T.O | T.O | T.O | T.O |

**Table 1.** Experimental Results. D is the depth of the STE assertion, #Vars is the number of symbolic variables, #N is the number of circuit nodes in thousands, and 3V, BDD, DR and BMC are the times required by 3VJA, BDD STE, Dual Rail SAT STE, and BMC, respectively.

## 7 Related Work

SAT based methods for STE were previously suggested in [2], [21] [4], and [15].

In [21], *non-canonical Boolean expressions* are used to represent the symbolic expressions of the circuit’s nodes during the simulation. At the end of the simulation, a SAT solver is used to check if the resulting expressions meet the requirements of the STE assertion. In this method, the expressions associated with the nodes may grow very large, and even become too large to handle. In such cases, a theorem prover has to be used in order to simplify them. This method is inherently different than 3VJA.

In [2], the *dual rail* encoding is used to create a CNF formula for STE. This construction is referred to in [4] as *simulation based SAT STE*. In [4], a different construction is suggested, and is referred to as *constraint based STE*. The *constraint based* construction is equivalent to the construction presented in [1], that we used in our work. This construction forces propagation of Boolean values through the gates of the circuit. The *simulation based* construction forces propagation of $X$ values as well, and results in much larger CNF formulae. In [4], it is shown that the *constraint based* construction outperforms the *simulation based* construction. We compared 3VJA to the *constraint based* construction in Section 6.

In [15], the constraint based construction is solved by a 3-valued SAT solver. In this work, Boolean variables of a SAT solver represent 3 values, considering an “unassigned” variable as $X$. The definition of satisfiability is changed respectively. In this work, clauses are regarded as multi-sets, and the definition of the resolution is also changed. Note that in our work we do not change the definition of the satisfiability of a formula. Instead, our algorithm does not *satisfy* the formula, but rather *justifies* the root of the graph. Moreover, in our work we distinguish between unassigned nodes and nodes assigned with $X$. This distinction allows us to propagate $X$ values, and to suggest an automatic refinement for too coarse abstractions. Additionally, while the 3-valued resolution defined in our paper is similar to the resolution defined in [15], the reasons for their correctness are different. As described in [15], modifying the SAT solver to fit the new definition of satisfaction and resolution did not yield good performance.

Additionally in [15], an approximation to 3-valued SAT is computed. This algorithm corresponds to a different semantics than the STE semantics, and an assertion that holds by this algorithm might not hold in STE semantics. This algorithm is also not suitable for refining STE assertions.

An automatic refinement scheme was suggested in [20]. This refinement scheme chooses a nodes that is assigned with $X$, and tries to choose a small set of inputs such that this node will evaluate to 0 or 1. In [16], a method for assisting manual refinement is presented. Our refinement scheme eliminates a spurious proof of unjustifiability of the circuit in each iteration, and is inherently different than these methods.

## 8 Conclusions and Future Work

We have presented a 3-valued justification algorithm, 3VJA, that uses a *DAG* and a CNF descriptions of a circuit, and finds a 3-valued justification for the value at the root. We showed how to use 3VJA for STE.

We implemented 3VJA and compared it to other STE tools. It is our opinion that 3VJA is a valuable complement to BDD based STE, especially for falsification, as is the case in other model checking problems. 3VJA is far less sensitive to the number of symbolic variables than BDD methods. Moreover, for falsification, 3VJA may find an erroneous path quickly, while a BDD engine has to compute the values of all the nodes in all the iterations prior to the contradiction.

We compared 3VJA to other SAT based algorithms and in many cases showed a significant speedup. This is the result of introducing the notion of $X$ into the Boolean context, without doubling the number of variables that are used, by propagating $X$ values over a graph representation of the circuit, and by learning $X$ values trough 3-valued resolution. While BMC is a powerful model checking method, it is considered useful mainly for falsification of "shallow" bugs. Exploiting the abstraction used in STE, 3VJA may extend the capabilities of BMC as well.

Last, we showed that 3VJA can be used for an automatic refinement scheme of STE assertions. This scheme takes a "CEGAR" approach, where the spurious counter examples are proofs of unjustifiability of the problem, that depend on $X$ values. The refinement adds symbolic variables to the nodes that are needed in order to eliminate the proof. We intend to research heuristics for minimizing the set of variables that have to be refined for eliminating spurious counter examples.

3VJA also allows us to address the problem of vacuity in STE. Given an STE assertion $A \rightarrow C$, it might hold vacuously if $A$ may never occur in the model. We believe that by applying our justification algorithm, this problem can be solved efficiently.

## References

1. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *DAC*. IEEE Computer Society Press, June 1999.
2. P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *CAV '01*, pages 454–464, London, UK, 2001.
3. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *Journal of the ACM*, 50(5):752–794, 2003.
4. K. Classen and J.-W. Roorda. A new SAT-based algorithm for symbolic trajectory evaluation. In *CHARME*, 2005.
5. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *CACM*, 5(7), July 1962.
6. M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7(3):201–215, July 1960.
7. H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Trans. Computers*, 32(12):1137–1144, 1983.
8. M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining Strengths of Circuit-Based and CNF-Based Algorithms for a High-Performance SAT Solver. In *DAC02*.
9. O. Grumberg, Assaf Schuster, and A. Yadgar. Hybrid BDD and all-sat method for model checking and other application. Technical report, Technion, CS-2007-08, 2007.
10. H. Jin, M. Awedh, and F. Somenzi. CirCUs: A Satisfiability Solver Geared towards Bounded Model Checking. In *CAV'04*.
11. F. Lu, Li-C. Wang, K.-T. Cheng, and R. C-Y Huang. A Circuit SAT Solver With Signal Correlation Guided Learning. In *DATE '03*, page 10892, Washington, DC, USA, 2003. IEEE Computer Society.
12. J.P. Marques-Silva and K.A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *IEEE ICTAI*, 1996.
13. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *39th Design Aotomation Conference (DAC'01)*, 2001.
14. M. Pandey, R.Raimi, R. E. Bryant, and M. S. Abadir. Formal verification of content addressable memories using symbolic trajectory evaluation. *dac*, 00:167, 1997.
15. J-W Roorda. Symbolic trajectory evaluation using a satisfiability solver. Licentiate Thesis, 2005.
16. Jan-Willem Roorda and Koen Claessen. Sat-based assistance in abstraction refinement for symbolic trajectory evaluation. In *CAV'06*, pages 175–189, 2006.
17. T. Schubert. High level formal verification of next-generation microprocessors. In *DAC'03*.
18. C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2), 1995.
19. C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. F. Melham, M. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 24(9), 2005.
20. Rachel Tzoref and Orna Grumberg. Automatic refinement and vacuity detection for symbolic trajectory evaluation. In *CAV06*, pages 190–204, 2006.
21. J. Yang, R. Gil, and E. Singerman. satGSTE: Combining the abstraction of GSTE with the capacity of a SAT solver. In *DCC*, 2004.
22. J. Yang and A. Goel. GSTE through a case study. In *ICCAD*, 2002.
23. H. Zhang. SATO: An efficient propositional prover. In *(CADE)*, 1997.
24. L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, 2001.