New Approaches to Model Checking and to 3-Valued Abstraction and Refinement

Avraham Yadgar

New Approaches to Model Checking and to 3-Valued Abstraction and Refinement

Research Thesis

In Partial Fulfillment of the
Requirements for the
Degree of Doctor of Philosophy

Avraham Yadgar

Submitted to the Senate of the
Technion - Israel Institute of Technology

Adar, 5770 Haifa March 2010

The Research Thesis Was Done Under The Supervision of Prof. Orna Grumberg and Prof. Assaf Schuster in the Department of Computer Science.

The generous financial help of the Technion is gratefully acknowledged.

I thank Orna and Assaf for their guidance and help which allowed the creation of this work.

I especially thank my parents, Yardena and Menashe Yadgar, and my wife, Gala Yadgar, for their infinite support during my studies and research.

Contents

Abstract

Model Checking is the problem of verifying the correctness of a system with respect to temporal logic properties [20]. This is an important problem, which is constantly becoming more critical, as software and hardware systems are growing rapidly. Complex systems are more prone to implementation errors, thus demanding a thorough validation. The high time and space requirements of the checking makes the verification of large systems hard to perform. Despite recent advancement in model checking algorithms and tools, efficient model checking remains a critical problem, due to the growing complexity of software and hardware systems.

In this work we present new approaches to symbolic model checking of hardware systems. Our approaches are aimed at increasing the performance of model checkers. We address the size of the systems that are being checked, the verification runtime, and its memory requirements. Our work is also aimed at increasing the automatization of model checking methodologies, thus decreasing human effort and required expertise of the verification engineer.

In our work we propose a memory efficient hybrid SAT and BDD based algorithm for model checking of circuits. This algorithm exploits the advantages of both SAT and BDD approaches, while avoiding their drawbacks as much as possible.

We also present a 3-valued-SAT based approach for Symbolic Trajectory Evaluation (STE). In this approach, STE is performed by using a novel 3-valued SAT solver, instead of the conventional, BDD-based, approach.

A common approach for addressing the capacity problem in model checking is using *abstraction* and *refinement*. In this work we present different methods for automatic refinement in STE, instead of the common manual refinement. These methods require less user effort and expertise, and are less error prone.

We also present an automata theoretic approach to 3-Valued model checking, which can be used for explicit and symbolic model checking. We implement this approach in 3-valued *Bounded Model Checking* (BMC). Our BMC abstraction outperforms conventional abstraction in BMC both in the sizes of the systems being checked, and in the runtime of the verification tool. We propose a new methodology for using BMC for very large systems, based on our 3-valued abstraction. Our methodology requires significantly less user effort and expertise than the common methodologies for BMC of large designs, and is less error prone.

Abbreviations and Notations

AP	—	Set of atomic propositions
$M = \langle S, I_0, R, L \rangle$	—	Kripke structure
1	—	Truth value 'true'
0	—	Truth value 'false'
X	—	Truth value 'unknown'
$\perp$	—	Truth value 'bottom'
AP_2	—	$\{p = 0, p = 1 \mid p \in AP\}$
AP_3	—	$\{p = 0, p = 1, p = X \mid p \in AP\}$
$C = \langle \mathcal{V}, I_0, PI, F \rangle$	—	Hardware circuit
$\mathcal{N}$	—	Circuit nodes
(n, t)	—	Value of circuit node n at time t
$[n, d]$	—	Circuit node n and a quaternary value d
TRG	—	Transition relation graph of a circuit
COI	—	Cone of influence
BCOI	—	Bounded cone of influence
LTL	—	Linear time logic
P	—	LTL property
ψ	—	LTL path formula
$\models$	—	Satisfies
$\mathcal{B}= \langle \Sigma, Q, q_{in}, \rho, \alpha \rangle$	—	Büchi automaton
$\mathcal{L}(\mathcal{B})$	—	The language of automaton $\mathcal{B}$
$\mathcal{B}_{\neg\psi}$	—	Automaton that accepts runs that contradict the path formula ψ
E	—	Product of a Kripke structure with a Büchi automaton
BMC	—	Bounded model checking
UBMC	—	Unbounded model checking
$\mathcal{Q}$	—	Quaternary domain $\{0, 1, X, \perp\}$
$\mathcal{T}$	—	Ternary domain $\{0, 1, X\}$
$\sqsupseteq$	—	Partial order in the quaternary and ternary domains
$\hat{f}$	—	Ternary function obtained by syntactically replacing Boolean variables and operators in a Boolean function f by ternary

		variables and opertators
$\succeq$	—	Abstraction relation
STE	—	Symbolic Trajectory Evaluation
X-possible(node)	—	Exists a run for which the node evaluates to X
φ	—	Propositional formula
$\vert_{!0}$	—	$not - 0$. A ternary variable that cannot be assigned 0
$\vert_{!1}$	—	$not - 0$. A ternary variable that cannot be assigned 1
dr	—	Degree of Responsibility
$C_0(n)$	—	Number of changes in the environment that imply $n = 0$
$C_1(n)$	—	Number of changes in the environment that imply $n = 1$

Chapter 1

Introduction

In this work we present new approaches to symbolic model checking of hardware systems. Our approaches are aimed at increasing the performance of model checkers. We address the size of the systems that are being checked, the verification runtime, and its memory requirements. Our work is also aimed at increasing the automatization of model checking methodologies, thus decreasing human effort and required expertise of the verification engineer.

In our work we propose a memory-efficient SAT and BDD based algorithm for model checking of circuits, and a SAT based approach for Symbolic Trajectory Evaluation. We also address the capacity problem of model checkers by proposing new approaches to abstraction and refinement. We present automatic refinement methods for symbolic trajectory evaluation, instead of the common manual refinement. We also present an automata theoretic approach to 3-Valued model checking, and propose a new methodology for using Bounded Model Checking based on this approach.

Model Checking is the problem of verifying the correctness of a system with respect to temporal logic properties [20]. This is an important problem, which is constantly becoming more critical, as software and hardware systems are growing rapidly. Complex systems are more prone to implementation errors, thus demanding a thorough validation. The high time and space requirements of the checking makes the verification of large systems hard to perform. Despite recent advancement in model checking algorithms and tools, model checking remains a critical problem, due to the growing complexity of software and hardware systems.

In this work we discuss *symbolic model checking* of hardware designs. Symbolic algorithms hold some symbolic representation of the system's states and transitions, and perform operations over this representation. Often, symbolic algorithms can handle systems for which the explicit representation is too large to fit into a given machine's memory [10, 5]. Common approaches to symbolic model checking are BDD and SAT based algorithms.

Each of these approaches has different advantages and disadvantages.

Binary Decision Diagrams (BDDs)[9] are a compact representation of boolean functions. BDDs are used to represent sets of states, and the system's transition relation. The model checking algorithms can then perform boolean operations, as well as quantifications on these sets. BDD-based model checking algorithms are usually quite efficient when the transition relation and the set of system states can be represented with BDDs [10]. This is because applying Boolean operations is polynomial in the size of the BDDs. However, quantification is exponential in the size of the BDDs. Therefore, BDDs are quite unpredictable and tend to explode on intermediate results of operations which require quantification.

An alternative symbolic approach exploits the significant recent improvement of SAT solvers. SAT solvers are memory efficient, and are less sensitive to the size of the problem at hand than BDDs. Sets of states, and the system's transition relation are represented as boolean formulae, usually given in CNF. These formulae are given as an input to a SAT solver when performing Bounded Model Checking (BMC) [5], induction based model checking [74, 6], or interpolation based model checking [58]. Several works [57, 12, 31, 44] perform symbolic model checking by using an All-SAT solver, which instantiates all the solutions for a given Boolean formula.

Apart from the approaches described above, other works such as [35] make a hybrid use of both BDDs and SAT methods for model checking.

In this work we present a hybrid BDD and all-SAT approach to model checking of hardware circuits. In our approach, the SAT algorithm uses a propositional symbolic representation of the system, as well as a graph representation of the circuit being checked. By using the combination of BDD, propositional, and graph representations of the system, we are able to exploit the benefits of each representation, while avoiding their drawbacks as much as possible. We also presented this approach in [83].

One of the most efficient approaches for increasing the capacity of model checking tools is by *abstraction*. In this approach, some of the information about the system is abstracted out, resulting in a new *abstract* system, which does not accurately describe the original *concrete* system. The abstract system is smaller than the concrete one, and therefore is easier to represent or manipulate. In many cases, the abstract system is enough for reasoning about the concrete one.

Different works such as [47, 19, 13, 21, 27] suggested algorithms for *abstraction* and *refinement*. Most of the abstraction-refinement algorithms construct an over-approximated system and follow the *CounterExample Guided Abstraction Refinement* (CEGAR) approach [47, 19]. In this approach, model checking starts with a small abstract system. Checking such a system may yield a spurious counterexample. This is an erroneous trace of the abstract system, which does not represent an actual behavior of the concrete system. In that case, the abstract system is refined according to the spurious counterexample. That is, some of the details of the system, that were abstracted out, are added back. This results in a larger abstract system, which describes the concrete system more accurately.

In this work we discuss 3-valued abstraction and refinement in model checking. In this abstraction, circuit nodes are ternary instead of Boolean. The third value, X, stands for "*unknown*". Abstraction is performed by assigning the value X to nodes, thus abstracting out the logic that drives them. We discuss 3-valued abstraction in *Symbolic Trajectory Evaluation* (STE), and in automata theoretic based model checking.

A 3-Valued abstraction-refinement framework for CTL was introduced in [75], and for μ-calculaus in [29] and [30]. 3-valued model checking has been used in other works both for abstracting out parts of the model [8, 84, 37] and for indicating that it is impossible to give a definite result [69, 14, 71]. In [71], 3-valued bounded model checking of $\mu-$calculus properties is presented, where the third value $\perp$ represents indefinite results due to the bound. 3-Valued abstraction-refinement is also used for symbolic simulation in STE [72] and SAT-based STE

STE [72] is a successful method for formally verifying very large hardware circuits with wide data paths [73, 70, 86]. STE is a symbolic symbolic simulation which uses 3-valued abstraction. In STE, X is used for abstracting out nodes, and the logic that drives them. STE is traditionally performed by using BDDs to represent the values of circuit nodes at different times. In order to represent ternary circuit nodes, the *dual rail* encoding is used. In this method, two BDDs are used for representing the value of each node. Refinement in STE is a critical task, which is traditionally done manually. This task is labor intensive, and requires user's expertise.

In this work we present a SAT-based approach for STE. Our approach makes use of a novel 3-valued circuit SAT algorithm, which is capable of representing ternary circuit nodes without using the dual rail encoding. In many cases, using our SAT solver increases the capability of STE, compared to using of BDDs. We also presented this work in [32]

We also present two approaches for automatic refinement in STE. The first approach is based on our 3-valued circuit SAT solver, and is applicable for our SAT-based STE algorithm. We presented this approach in [32]. The second approach uses the notion of *responsibility* [17] in order to refine too abstract systems. We also presented this approach in [16].

Finally, we present an automata theoretic approach to 3-Valued model checking. Similarly to STE, our ternary domain includes the value X, which is used for abstracting out nodes and the logic that drives them. For our 3-valued model checking approach, we follow the automata theoretic approach to model checking [80], and adapt it to the ternary domain. Our 3-valued approach is applicable both for explicit and symbolic model checking.

In this work we use our 3-valued approach for a SAT-based, 3-valued, BMC algorithm. Our 3-valued BMC algorithm outperforms current abstraction methods for BMC, in terms of the sizes of the systems that can be checked, and in terms of runtime. We also present a new methodology for abstraction in BMC of very large systems, based on this approach. Our methodology requires significantly less expertise and familiarity of the user with the system, compared to the common methodologies. Therefore, the validation process requires significantly less effort and time, and is much less error prone.

The rest of this work is organized as follows. In Chapter 2 we give the background to our work. In Chapter 3 we present a hybrid BDD and all-SAT approach to model checking. In Chapter 4 we present a SAT-based STE approach, and in Chapter 5 we present approaches for automatic refinement in STE. In Chapter 6 we present our automata theoretic approach to 3-valued model checking, and its BMC implementation, and conclude our work in Chapter 7.

Chapter 2

Preliminaries

Given a model M, and a specification φ, the *model checking problem* is the problem of deciding whether M satisfies φ or not. In Part I of this chapter we present the definitions of models and the specification language, and describe various approaches to model checking. In Part II of this chapter, we describe data structures and algorithms that are used for solving the model checking problem.

Part I: Model Checking

2.1 Kripke Structures

In this work we discuss verification of systems that are modelled as *Kripke structures.* Given a set of atomic propositions AP, we slightly change the standard notion, and define $AP_2 = \{p = 0, p = 1 | p \in AP\}$. A *Kripke structure* is a tuple $M = \langle S, I_0, R, L\rangle$, where S is a finite set of states, and $I_0 \subseteq S$ is a set of initial states. $R \subseteq S \times S$ is a *total* transition relation. That is, $\forall s \in S\ \exists s' \in S,\ (s, s') \in R$. $L : S \to \mathcal{P}(AP_2)$ is a labelling function such that for every $s \in S$, and for every $p \in AP$, exactly one of $p = 0, p = 1$ is in $L(s)$. A *path* in M is an infinite set of states $s_0, s_1 \ldots$ such that $\forall i,\ (s_i, s_{i+1}) \in R$. For a path $\pi = s_0, s_1 \ldots$, π^i is the suffix $s_i, s_{i+1} \ldots$ of π.

2.2 Circuits

A hardware *circuit* C is a directed graph. The graph's nodes $\mathcal{N}$ are primary input and internal nodes, where internal nodes are latches and combinational gates. A node can get a Boolean value, and a combinational gate represents a Boolean operator. There is a directed edge from node n_1 to node n_2 if n_1 is an *input* to n_2 in the circuit. The value of a node n is the result of applying its operator on its inputs at each clock cycle. The graph of C

may contain circles, but not combinational circles. We formally define a circuit $C = \langle \mathcal{V}, I_0, PI, F\rangle$, where $\mathcal{V}$ is the set of latches in C, I_0 is a set of possible initial values to $\mathcal{V}$, PI is the set of primary inputs, and F is a set of transition functions such that $\forall v_i \in \mathcal{V},\ f_i : 2^{\mathcal{V}} \times 2^{PI} \to \{0,1\}$ defines the value of v_i in the next state as a function of the current values of $\mathcal{V}$ and PI. Since F consists of a set of functions, it is called a *partitioned transition relation*. We give an example of a partitioned transition relation in Figure 2.1(a).

The *cone of influence* (COI) of a node n, C_n, is the set of all the nodes in the circuit that may influence the value of n, and is defined recursively as follows: the COI of a combinational node is the union of the COI of its inputs, and the COI of a latch or an input is the empty set.

We denote by (n,t) the value of node n at time t. The *bounded cone of influence* (BCOI) of a node n at time t contains all (n',t') with $t' \leq t$ that may influence the value of (n,t), and is defined recursively as follows: the BCOI of a combinational node at time t is the union of the BCOI of its inputs at time t, and the BCOI of a latch at time t is the union of the BCOI of its inputs at time $t-1$. The BCOI of an input is the empty set.

A circuit can be viewed as a Kripke structure $M = \langle S, I_0, R, L\rangle$ where $AP = \mathcal{V}$. A *state* s in M is an assignment of values to every latch, $s : \mathcal{V} \to \{0,1\}$. The transition relation R is given by F, where $(s,s') \in R \Leftrightarrow \exists in \in 2^{PI}, \forall v_i \in \mathcal{V},\ s'(v_i) = f_i(s,in)$. We abuse the notation of F and use it as a function $F : S \times 2^{PI} \to S$, where $s' = F(s,in)$. $L : S \to \mathcal{P}(AP_2)$ is a labelling function such that $(v_i = c) \in L(s) \Leftrightarrow s(v_i) = c$, for $c \in \{0,1\}$. A *path* $\pi \in M$ describes a run of the circuit C.

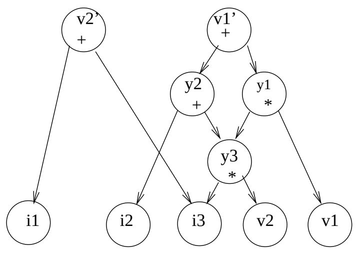
2.2.1 Transition Relation Graphs

As described above, a partitioned transition relation consists of a set of functions F, such that for each next-state variable $v'_i \in \overline{v'}$, $v'_i = f_i(\overline{v}, \overline{I})$. The transition relation can be represented as a DAG, where the roots are the next-state variables $\overline{v'}$, and the terminal nodes are the current-state variables $\overline{v}$, and the input variables $\overline{I}$. Each internal node in the graph is associated with a Boolean variable, and corresponds to a subexpression of the formulae that define the transition relation. Each internal node is associated with a Boolean operation, and the node's value is the result of applying the corresponding operation on its successors. The graph in Figure 2.1(b) corresponds to the transition relation given in Figure 2.1(a). Nodes y_1, y_2 and y_3 are auxiliary variables for the subexpressions of the function that computes $\overline{v'}$ out of $\overline{v}$ and $\overline{I}$.

We refer to the graph of the transition relation as a *Transition Relation Graph* (TRG). We refer to a node in the TRG by the name of its corresponding variable.

$v'_1 = (v_1 \wedge (v_2 \wedge i_3)) \vee ((v_2 \wedge i_3) \vee i_2)$
$v'_2 = i_3 \vee i_1$

(a) Transition Relation



(b) TRG

Figure 2.1: (a) Partitioned Transition Relation. v'_1 and v'_2 are given as functions of v_1, v_2, i_1, i_2, i_3. (b) The TRG corresponding to the transition relation given in (a). $\overline{v} = \{v_1, v_2\}$, $\overline{I} = \{i_1, i_2, i_3\}$ and $\overline{v'} = \{v'_1, v'_2\}$. y_1, y_2 and y_3 are auxiliary variables representing the subexpressions of the transition relation.

2.3 Linear Time Logic

Next we define Linear Time Logic (LTL)[65] and its semantics. An LTL *path formula* ψ is recursively defined as follows, where ψ_1 and ψ_2 are LTL path formulae:

$$\psi = p|\neg\psi_1|\psi_1 \wedge \psi_2|N\psi_1|\psi_1 U \psi_2$$

$p \in AP$, U stands for the "Until" time operator, and N stands for the "Next" time operator. Note that throughout this work we denote the "Next" time operator by N, rather than the common notation X.

For a path $\pi = s_0, s_1 \ldots$, π *satisfies* a path formula ψ, denoted by $\pi \models \psi$, is defined as follow:
$\pi \models p \in AP \Leftrightarrow p = 1 \in L(s_0)$
$\pi \models \neg\psi_1 \Leftrightarrow \neg[\pi \models \psi_1]$
$\pi \models \psi_1 \wedge \psi_2 \Leftrightarrow \pi \models \psi_1$ and $\pi \models \psi_2$
$\pi \models N\psi_1 \Leftrightarrow \pi^1 \models \psi_1$
$\pi \models \psi_1 U \psi_2 \Leftrightarrow \exists j,\ \pi^j \models \psi_2 \wedge (\forall i\ 0 \leq i < j,\ \pi^i \models \psi_1)$

An *LTL formula* is of the form $P = A\psi$, where ψ is a path formula. A Kripke structure M satisfies P, denoted by $M \models P$, iff $\forall \pi = s_0, s_1 \ldots \in M$ such that $s_0 \in I_0$, $\pi \models \psi$.

2.4 Checking Safety Properties

Temporal properties are typically divided to *safety* and *liveness* properties. Checking safety properties is considered to be simpler than checking liveness properties, and therefore translation from liveness properties to safety properties has been suggested [4]. In this section we discuss checking of safety properties of the form $P = AGe$, where e is a Boolean expression over

```
$ModelCheck(I_0, R, \neg E)$
1) $S^* \leftarrow \phi$
2) $new \leftarrow \neg E$
3) $while\ (new \neq \phi)$ {
4)    $if\ new \cap I_0 \neq \phi$
5)       $return$ 0
6)    $S^* \leftarrow S^* \cup new$
7)    $new \leftarrow Pre - Image(new) \setminus S^*$
8) }
9) $return$ 1
```

Figure 2.2: Model Checking Algorithm for $P = AGe$. The pre-image computation at line 7 is performed only for the newly found states, in order not to search for the predecessors of a state more than once. The algorithm returns 1 if $M \models P$, and 0 otherwise.

the atomic formulae. Checking of a large class of safety properties can be reduced to checking of invariants of this form [2].

Let $M = (S, I_0, R, L)$ be a Kripke structure, over a set of atomic formulae AP. For a given set of states S', the set of all the predecessors of states in S', denoted *Pre-Image*(S'), is

$$Pre - Image(S') = \{s \mid \exists s' \in S', R(s, s')\} \tag{2.1}$$

Using the pre-image operation, the symbolic algorithm for model checking given in Figure 2.2 can be constructed. The input for the Algorithm is I_0, R, and $\neg E$, which is the set of states where the Boolean expression e does not hold. Note that $\neg E$ is trivial to compute. The model checking algorithm determines whether there is a path from some $s_0 \in I_0$ to some $s_{\neg e} \in \neg E$. If such a path exist, we conclude that $M \not\models P$. Otherwise, $M \models P$.

In order to find such a path, the algorithm performs an iterative *backward search*, starting from $\neg E$. In each iteration, pre-image computation is done for the states which were found in the previous iteration. This way, there is a path from every state which is found by the algorithm, to some state in $\neg E$. The algorithm terminates if at some point, a state $s_0 \in I_0$ is found, which implies that $M \not\models P$, or if no new states are found, which implies that there is no path from a state in I_0, to an error state in $\neg E$, and thus $M \models P$.

2.5 The Automata-Theoretic Approach to Model Checking

2.5.1 Büchi Automata

An infinite word w over an alphabet Σ is an infinite sequence $w = \sigma_0, \sigma_1 \ldots$, where $\sigma_i \in \Sigma$. A *Büchi automaton on infinite words* is a tuple $\mathcal{B}$=

$\langle \Sigma, Q, q_{in}, \rho, \alpha \rangle$ where Σ is the input alphabet, Q is a finite set of states, $\rho : Q \times \Sigma \rightarrow 2^Q$ is the transition function, $q_{in} \in Q$ is an initial state, and $\alpha \subseteq Q$ is an acceptance condition. $\rho(q, \sigma)$ is the set of states that $\mathcal{B}$ can move to from state q, with the input letter σ.

A run of $\mathcal{B}$ on a word $w = \sigma_0, \sigma_1 \ldots$ is a function $r : \mathbb{N} \rightarrow Q$ where $r(0) = q_{in}$, and $\forall i \geq 0$, $r(i+1) \in \rho(r(i), \sigma_i)$. That is, a run is a series of states starting from the initial state and obeying the transition function. Since Q is finite, there is a set of states that appear infinitely often in r, denoted by $inf(r)$. $inf(r) = \{q \in Q |$ for infinitely many $i \in \mathbb{N}$, $r(i) = q\}$.

A run r on a word w *accepts* w iff $inf(r) \cap \alpha \neq \emptyset$. Since $\mathcal{B}$ is nondeterministic, there are multiple runs on w. A Büchi automaton accepts w if there exists a run that accepts w. The language of an automaton $\mathcal{B}$, denoted by $\mathcal{L}(\mathcal{B})$, is the set of words that $\mathcal{B}$ accepts.

Throughout this work we assume that the states of Büchi automata are represented by a Boolean encoding. Let $\mathcal{B}= \langle \Sigma, Q, q_{in}, \rho, \alpha \rangle$ be a Büchi automaton, and let Y be a set of Boolean state variables. A state $q \in Q$ is an assignment $q : Y \rightarrow \{0, 1\}$ to the variables in Y.

For practical encoding of $\mathcal{B}$, it is useful to represent ρ as a Boolean function, rather than a relation, over the states Q and the alphabet Σ. It is possible to construct this representation of the transition relation by using an additional integer variable d. The value of d determines which of the possible transitions will be taken next. Let $D = \{1 \ldots |Q|\}$. The transition function is now defined as $\rho_D : Q \times \Sigma \times D \rightarrow Q$. ρ_D can also be represented by a set of Boolean functions F such that $\forall y_i \in Y$, the value of y_i in the next state q', denoted by $q'(y_i)$, is given by a Boolean function $f_{y_i} : Q \times \Sigma \times D \rightarrow \{0, 1\}$.

2.5.2 Büchi Automata for LTL

For a path formula ψ over a set of atomic propositions AP, it is possible to construct a Büchi automaton over infinite words $\mathcal{A}= \langle \Sigma, Q, q_{in}, \rho, \alpha \rangle$ such that $\mathcal{L}(\mathcal{A})$ is the set of all infinite words over Σ that satisfy ψ [80]. $\Sigma \subseteq \mathcal{P}(AP_2)$, such that for every $\sigma \in \Sigma$, and for every $p \in AP$, exactly one of $p = 0, p = 1$ is in σ.

In order to reduce the size of $\mathcal{A}$, its transition function is constructed such that for every letter $\sigma \in \Sigma$, and for each transition, it considers only the atomic propositions that are required in order to determine if w should be accepted, rather than all the atomic propositions in σ. If for some $p \in AP$, neither $p = 0$ nor $p = 1$ appears in σ, it means that p can take any value in w [55]. We later show that the reduced alphabet is helpful for our work.

2.5.3 Model Checking using Büchi Automata

Let $M = \langle S_M, I_0^M, R_M, L_M\rangle$ be a Kripke structure over a set of atomic formulae AP, and let $P = A\psi$ be an LTL formula over AP.

- A counterexample for P is a path $\pi \in M$ such that $\pi \models \neg\psi$.
- Let $\mathcal{B}_{\neg\psi} = \langle \Sigma, Q, q_{in}, \rho, \alpha\rangle$ be the Büchi automaton corresponding to $\neg\psi$.
- Let $E = \langle S_E, I_0^E, R_E, L_E, \alpha_E\rangle$ be the product of M and $\mathcal{B}_{\neg\psi}$, where $S_E = S_M \times Q$, $I_0^E = I_0^M \times \{q_{in}\}$, $R_E = \{((s,q),(s',q'))|(s,s') \in R_M,\ q' \in \rho(q, L_M(s))\}$, $L_E(s,q) = L_M(s)$ and $\alpha_E = S_M \times \alpha$.
- A *fair* path $\pi \in E$ is a path such that $inf(\pi) \cap \alpha_E \neq \emptyset$.

A fair path in E represents a path in M, and a word accepted by $\mathcal{B}_{\neg\psi}$. It therefore represents a counterexample for P in M. Thus, the model checking problem is reduced to finding a fair path in E.

2.6 Bounded Model Checking

Bounded Model Checking (BMC) [5] is an iterative process for checking models against LTL formulae. Let M and $P = A\psi$ be a Kripke structure and an LTL property, respectively. For BMC, M and P are represented by propositional formulae. At iteration i, BMC uses a SAT solver for finding a counterexample to P in M, consisting of i states, or for determining that there is no such counterexample. If no counterexample is found, BMC terminates when the desired bound k is reached, thus proving that there is no counterexample of k or less states in M.

Unbounded model checking is possible by using induction [74, 6] or interpolation [58]. However, in many practical cases these methods are infeasible, and BMC terminates when the conclusion if there is a bug of length i becomes too hard to compute.

Next we show how BMC is used to implement automata-based model checking. Let $E = \langle S_E, I_0^E, R_E, L, \alpha_E\rangle$ be the product of M and $\mathcal{B}_{\neg\psi}$, as described in Section 2.5.3. For ease of presentation in the following sections, we suggest a slightly different description of E. Following the definition of E, R can be represented as

- $R_E = R_M^E \cap R_B^E$, where $R_M^E = \{((s,q),(s',q'))|(s,s') \in R_M\}$, and $R_B^E = \{((s,q),(s',q'))|q' \in \rho(q, L(s))\}$

Similarly,

- $I_0^E = I_0^{ME} \cap I_0^{BE}$, where $I_0^{ME} = I_0^M \times Q$, and $I_0^{BE} = S_M \times \{q_{in}\}$

For BMC, $I_0^{ME}, I_0^{BE}, R_M^E, R_B^E$ and α_E are encoded as propositional formulae, as described in [5]. When clear from the context, we do not distinguish between these sets and their propositional representation.

We define the propositional formula φ_i as follows:

$$fair_i(e_0 \dots e_i) = \bigvee_{0 \leq l < i} \left((e_l = e_i) \wedge \bigvee_{l \leq j < i} \alpha_E(e_j) \right) \tag{2.2}$$

$$\varphi_{ei}(e_0 \dots e_i) = I_0^{BE}(e_0) \wedge \bigwedge_{0 \leq j < i} R_B^E(e_j, e_{j+1}) \wedge fair_i(e_0 \dots e_i) \tag{2.3}$$

$$\varphi_{\pi i}(e_0 \dots e_i) = I_0^{ME}(e_0) \wedge \bigwedge_{0 \leq j < i} R_M^E(e_j, e_{j+1}) \tag{2.4}$$

$$\varphi_i(e_0 \dots e_i) = \varphi_{\pi i}(e_0 \dots e_i) \wedge \varphi_{ei}(e_0 \dots e_i) \tag{2.5}$$

A satisfying assignment to φ_i is a path π in M, which is also a path in E. π passes infinitely often in states in α_E, and therefore is a fair path in E. Thus, π represents a path in M where ψ does not hold. Consequently, φ_i represents all the erroneous runs with up to i different states in M. In the i^{th} iteration of BMC, a SAT solver is used to determine if φ_i is satisfiable. If $SAT(\varphi_i)$, then BMC terminates and returns the satisfying assignment as a counterexample to P. Otherwise, there is no such counterexample, and i is increased.

2.7 Unbounded Model Checking

We give a brief description of SAT-based *Unbounded Model Checking* (UBMC) of safety properties, as presented in [74] and [6]. While BMC is mainly used for bug hunting, UBMC can provide full verification of LTL properties. As before, let $M = \langle S, I_0, R, L \rangle$ be a Kripke structure over a set of atomic propositions AP. Let $P = AGexp$ be a safety property such that exp is a Boolean expression over the atomic formulae. For an index k, φ_b^k and φ_{ind}^k are defined as follows:

$$\varphi_b^k(s_0 \dots s_k) = I_0(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg exp(s_i) \tag{2.6}$$

$$loop\_free^k(s_0 \dots s_k) = \bigwedge_{\substack{0 \leq i,j \leq k \\ i \neq j}} (s_i \neq s_j) \tag{2.7}$$

$$\varphi_{ind}^k(s_0 \dots s_{k+1}) = \bigwedge_{i=0}^{k} (R(s_i, s_{i+1}) \wedge exp(s_i)) \wedge loop\_free^k(s_0 \dots s_k) \wedge \neg exp(s_{k+1}) \tag{2.8}$$

$\wedge$	X	0	1	$\perp$
X	X	0	X	$\perp$
0	0	0	0	$\perp$
1	X	0	1	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$\vee$	X	0	1	$\perp$
X	X	X	1	$\perp$
0	X	0	1	$\perp$
1	1	1	1	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$\neg$	
X	X
0	1
1	0
$\perp$	$\perp$

Figure 2.3: Quaternary Operations

UBMC is an iterative algorithm. At iteration k , satisfiability of φ_b^k represents a bug in M. Unsatisfiability of φ_b^k implies that there is no path of length k from an initial state with a state that does not satisfy *exp*. Unsatisfiability of φ_{ind}^k implies that for every loop free path (not necessarily from an initial state) of length $k+1$, if *exp* is satisfied along the first k states, then *exp* is satisfied in the $k+1$ state. If both φ_b^k and φ_{ind}^k are unsatisfiable, then there is no path from an initial state in M, for which *Gexp* does not hold, and therefore $M \models AGexp$. If φ_{ind}^k is satisfiable, the algorithm proceeds to the next iteration.

2.8 Quaternary Logic

Let $\mathcal{Q}$ be a quaternary domain $\mathcal{Q} = \{0, 1, X, \perp\}$, where X is "unknown" and $\perp$ is "over constrained". Let $\wedge$, $\vee$ and $\neg$ be quaternary operators, with the truth tables given in Figure 2.3. Let $\sqsupseteq$ be a partial order on $\mathcal{Q}$, defined by $X \sqsupseteq 0$, $X \sqsupseteq 1$, $0 \sqsupseteq \perp$, and $1 \sqsupseteq \perp$. $(\sqsupseteq, \mathcal{Q})$ is the lattice depicted in Figure 2.4. The quaternary operators are monotonic with respect to $\sqsupseteq$. That is, if $d_1' \sqsupseteq d_1$ and $d_2' \sqsupseteq d_2$, then $d_1' \wedge d_2' \sqsupseteq d_1 \wedge d_2$, $d_1' \vee d_2' \sqsupseteq d_1 \vee d_2$, and $\neg d_1' \sqsupseteq \neg d_1$. Throughout this work, when referring to monotonicity of functions, it is with respect to $\sqsupseteq$.

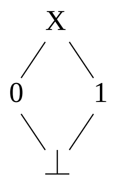


Figure 2.4: Quaternary Lattice

2.9 Symbolic Trajectory Evaluation (STE)

Symbolic Trajectory Evaluation (STE) [72] is a symbolic simulation of hardware circuits with abstraction. In STE, a circuit node can get a value in a quaternary domain $\mathcal{Q} = \{0, 1, X, \perp\}$. A node whose value cannot be determined by its inputs is given the value X("unknown"). $\perp$ 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.

For a circuit C in STE, a *state* s in C is an assignment of values from $\mathcal{Q}$ to every node, $s : \mathcal{N} \to \mathcal{Q}$. Note that this definition of a state is different than the general definition given in Section 2.2, where a state is defined only over the latches of a circuit. Note also, that the values of the latches and primary inputs of a circuit, determine the values of the rest of its nodes. A *trajectory* π 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 π, 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 π starting at time i.

Let $\mathcal{W}$ be a set of *symbolic Boolean variables* over the domain $\{0,1\}$. A *symbolic expression* over $\mathcal{W}$ is an expression consisting of quaternary operations, applied to $\mathcal{W} \cup \mathcal{Q}$. The truth tables of the quaternary operators are given in Figure 2.3. A *symbolic state* over $\mathcal{W}$ 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 $\mathcal{W}$. A *symbolic trajectory* over $\mathcal{W}$ is an infinite series of symbolic states, compatible with the circuit. It represents a set of trajectories, one for each assignment to $\mathcal{W}$. Given a symbolic trajectory π and an assignment ϕ to $\mathcal{W}$, $\phi(\pi)$ denotes the trajectory that is received by applying ϕ to all the symbolic expressions in π.

For specification in STE we use *Trajectory Evaluation Logic* (TEL). A formula in TEL is defined recursively over $\mathcal{W}$ as follows:

$$f ::= n \text{ is } p \mid f_1 \wedge f_2 \mid p \rightarrow f \mid \mathbf{N}f$$

where $n \in \mathcal{N}$, p is a Boolean expression over $\mathcal{W}$, and $\mathbf{N}$ is the next time operator. The *maximal depth* of a TEL formula f is the maximal number of nested N operators in f plus 1.

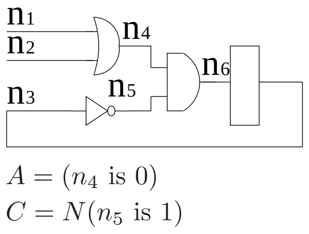
Given a TEL formula f over $\mathcal{W}$, a symbolic trajectory π over $\mathcal{W}$, and an assignment ϕ to $\mathcal{W}$, we define the satisfaction of f as in [79]:

$[\phi, \pi \models f] = \bot \ \leftrightarrow\ \exists i \geq 0, n \in \mathcal{N} : \phi(\pi)(i)(n) = \bot$. 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 $\phi(\pi)(0)(n) \in \{0,1\}$
$[\phi, \pi \models n \text{ is } p] = X \ \leftrightarrow\ \phi(\pi)(0)(n) = X$
$\phi, \pi \models p \rightarrow f \equiv \neg\phi(p) \vee \phi, \pi \models f$
$\phi, \pi \models f_1 \wedge f_2 \equiv (\phi, \pi \models f_1 \wedge \phi, \pi \models f_2)$
$\phi, \pi \models \mathbf{N}f \equiv \phi, \pi^1 \models f$

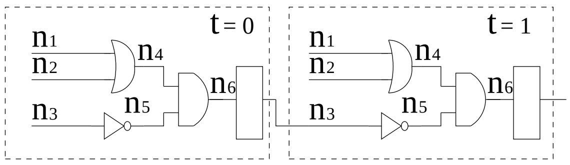
Note that given an assignment ϕ to $\mathcal{W}$, $\phi(p)$ is a constant (0 or 1).
We define the truth value of $\pi \models f$ as follows:
$[\pi \models f] = 0 \ \leftrightarrow\ \exists\phi : [\phi, \pi \models f] = 0$
$[\pi \models f] = X \ \leftrightarrow\ \forall\phi : [\phi, \pi \models f] \neq 0$ 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 levels 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 $[\phi, \pi \models f] = X$.

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



(a) A circuit M



(b) An Unrolling of M to depth 2

Figure 2.5: Circuits in STE. n_4 is an "AND" gate, n_6 is an "OR" gate, n_5 is a "NOT" gate, and n_6 is a latch.

t	n_1	n_2	n_3	n_4	n_5	n_6
0	X	X	X	0	X	0
1	X	X	0	X	1	X

Figure 2.6: Symbolic Simulation

nodes at specific times, and C expresses requirements that should hold on circuit nodes at specific times. We define the truth value of $[M \models A \Rightarrow C]$ as follows:

$[M\models A\Rightarrow C]=\bot \leftrightarrow \forall\pi:[\pi\models A]=\bot$
$[M\models A\Rightarrow C]=1 \leftrightarrow [M\models A\Rightarrow C]\neq\bot$ and $\forall\pi:[\pi\models A]=1$ implies $[\pi\models C]=1$
$[M\models A\Rightarrow C]=0 \leftrightarrow \exists\pi:[\pi\models A]=1$ and $[\pi\models C]=0$
$[M\models A\Rightarrow C]=X \leftrightarrow [M\models A\Rightarrow C]\neq 0$ and $\exists\pi:[\pi\models A]=1$ and $[\pi\models C]=X$

We denote by (n,t) the value of node n at time t. When applying A to M, if a node (n,t) is evaluated to X, but is also constrained to a Boolean value 0 or 1 by A, then (n,t) is assigned with the value imposed by A. As in [79], an *antecedent failure* is the case where $[M \models A \Rightarrow C] = \bot$. for a node n at time t we say that "(n,t) is X-possible" if there exists a trajectory π and an assignment ϕ such that $\phi(\pi)(t)(n)$, the value of n at time t, is X. If (n,t) is *X-possible*, and is also constrained to a Boolean value by C, then we say that (n,t) is *undecided*. This is the case where $[M \models A \Rightarrow C] = X$. Consider the circuit in Figure 2.5(a), and the STE assertion $A \Rightarrow C$, where $A = ((n4,0)\ is\ 0)$ and $C = ((n5,1)\ is\ 1))$. The table in Figure 2.6 corresponds to the symbolic simulation of this assertion. n_5 at time 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 $f^1_{n,t}, f^2_{n,t} : \mathcal{W} \rightarrow \{0,1\}$ over the set of symbolic variables $\mathcal{W}$. We further discuss dual rail encoding in Section 2.15.

Part II: Tools and Implementation

2.10 Binary Decision Diagrams

We give a short description of *Binary Decision Diagrams* (BDDs) and the way they are used to represent models.

A BDD is a data structure which is used to represent sets. A Boolean function $f : \{0,1\}^{|\overline{v}|} \rightarrow \{0,1\}$ over a set of variables $\overline{v}$, represents the set of assignments for which it evaluates to 'true'. A BDD represents such a function, and thus represents the corresponding set of assignments. We denote by $S(\overline{v})$, a BDD of a set of assignments to a set of variables $\overline{v}$. The order of the variables in $\overline{v}$ is of no importance for the evaluation of the Boolean function[1]. Therefore, throughout this work, we shall refer to $\overline{v}$ as a set of variables.

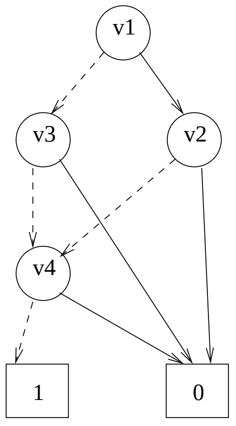
A BDD is a directed Acyclic graph (DAG) where the following holds:

- There is one root node, with no incoming edges.
- There are one or two terminal nodes with no outgoing edges. A terminal node is labeled '0' or '1', and there are no two terminal nodes with the same label.
- All the nodes except for the terminal nodes have two outgoing edges. Each such node is associated with a variable of the function represented by the BDD.
- Each path from the root to one of the terminal nodes contains no more than one node which is associated with a given variable.
- The nodes in all the paths from the root node to the terminal nodes are associated with variables in the same (partial) order.

The outgoing edges of a node are labeled 'right' and 'left', and are associated with the values '1' and '0' of the corresponding variable respectively. A path from the root of the BDD to one of the terminal nodes represents an assignment to the variables which are associated with the nodes in the path. If the terminal node of the path is labled '1', then $f = 1$ for the corresponding assignment. If the terminal node is labeled '0', then $f = 0$ for the corresponding assignment. Thus, by the value of the terminal node of a path, we know if the corresponding assignment is in the set which is represented by the function.

Note that a given path may include nodes associated with only a subset of the function's variables. In that case, the value of the formula is determined by the corresponding partial assignment. This partial assignment

[1]The order of the variables in the BDD effects its size. However, this order is not related to the order of the variables in the definition of the Boolean function.



(a)

$$\begin{array}{ll} \pi_1 : \neg v_4, \neg v_2, v_1 & \left\{ \begin{array}{l} a_{1,1} : \neg v_4, \neg v_3, \neg v_2, v_1 \\ a_{1,2} : \neg v_4, v_3, \neg v_2, v_1 \end{array} \right. \\ \pi_2 : \neg v_4, \neg v_3, \neg v_1 & \left\{ \begin{array}{l} a_{2,1} : \neg v_4, \neg v_3, \neg v_2, \neg v_1 \\ a_{2,2} : \neg v_4, \neg v_3, v_2, \neg v_1 \end{array} \right. \end{array}$$

(b)

Figure 2.7: (a) A BDD representing the function $f = \neg v_4 \wedge ((v_1 \wedge \neg v_2) \vee (\neg v_1 \wedge \neg v_3))$. Solid lines represent the 'right' successors of nodes, and dashed lines represent the 'left' successors of nodes. (b) The assignments corresponding to the paths from the root to the terminal node '1'.

represents all the complete assignments which agree with it on the values of the variables on the path. This way, multiple assignments are represented by a single path. An example of a BDD, and the assignments it represents is given in Figure 2.7.

Given a model, a common way of representing its states is by an encoding over a set of Boolean state variables $\mathcal{V}$. In that case, a function $f(\mathcal{V})$ evaluates to 1 for values of $\mathcal{V}$ that represent states in the model. f is then represented by a BDD over $\mathcal{V}$. Similarly, a transition relation of a model is represented by a BDD which represents pairs of current and next states.

Ordered BDD - An Ordered BDD (OBDD) is a BDD where on all the paths from the root node to a terminal node, the variables agree with a given order $v_1 < v_2 < \cdots < v_n$.

Reduced OBDD - A Reduced OBDD (ROBDD) is an OBDD where the following holds:

- No two nodes are associated with the same variable and agree on the right and left successors.
- No node has the same right and left successors.

Note that the given conditions eliminate redundant nodes and isomorphic subgraphs, and thus reduce the size of the BDD. In this work, when referring to a BDD, we refer to an ROBDD. Though BDDs are considered a compact representation for sets, we do not have a bound on their size tighter than $2^{|v|}$. The order of the variables within a BDD has a major effect on its size. Therefore, a great effort is made for choosing such an order which would yield a compact BDD.

Applying the logical operators $\neg$, $\wedge$ and $\vee$ is polynomial in the size of the BDDs. Universal and existential quantification might be exponential in the size of the BDD. For S_1 and S_2, sets of assignments, which are represented by the BDDs B_1 and B_2 respectively, the set $S_1 \cup S_2$ is represented by the BDD $B_1 \vee B_2$, and the set $S_1 \cap S_2$ is represented by the BDD $B_1 \wedge B_2$

2.11 The SAT Problem

The *Boolean satisfiability problem* (SAT) is defined as follows: Given a Boolean formula ϕ over a set of Boolean variables V, find an assignment A to V such that $\phi(V)$ has the value 'true' under this assignment. A is called a *satisfying assignment*, or a *solution*, for ϕ. If no such assignment exists, we say that ϕ is *unsatisfiable*, denoted by *unSAT*.

We shall discuss formulae presented in Conjunctive Normal Form (CNF). That is, ϕ 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 regard a clause as a set of literals, and a formula as a set of clauses.

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

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 $cl_1 \wedge cl_2 \wedge cl_{res}$ is satisfiable iff $cl_1 \wedge cl_2$ is satisfiable. 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 are involved in the resolution that creates it. The leaves are a subset of the original clauses of the formula. This subset of clauses is called an unSAT core.

We refer to an assignment by the values it assigns to the variables. That is, the assignment $\{x_0 = 1, x_1 = 0, x_2 = 0\}$ is referred to as $\{x_0, \neg x_1, \neg x_2\}$.

2.12 Davis-Putnam-Logemann-Loveland Backtrack Search (DPLL)

In this section we describe the *Davis-Putnam-Logemann-Loveland* Backtrack Search (DPLL) [23, 22], which is the basis for most of the modern SAT solving algorithms and tools.

We begin by describing the *Boolean Constraint Propagation* (*bcp()*) procedure. During SAT solving, given a partial assignment A and a clause cl, if there is one literal $l \in cl$ with no value, while the rest of the literals are all false, then in order to avoid a conflict, A must be extended such that $A(l) = 1$. cl is called a *unit clause* or an *asserting clause*, and the assignment to l is called an *implication.* The *bcp()* procedure iteratively finds all the implications at a given moment. This procedure is efficiently implemented in [59, 28, 87, 56, 52].

The DPLL algorithm walks the binary tree that describes the variables space. At each step, the algorithm assigns a value to one of the variables, thus *branching* in the tree. Each branch is assigned with a new *decision level.* After branching, the algorithm uses the *bcp()* procedure to compute all its implications. All the implications are assigned with the corresponding decision level. Note that at a given time, both truth values of a given variable may be implied by two different clauses. In that case, only one of the clauses can be satisfied, and we reach a conflict. If a conflict is reached, the algorithm backtracks in the tree, and chooses a new value for the most recent decision variable not yet tried both ways. The algorithm terminates if one of the leaves is reached with no conflict, or if the whole tree was searched, but no leaf was reached. In the first case, the path from the root to the leaf describes a satisfying assignment for ϕ. In the latter case, ϕ is unsatisfiable.

Pseudo code of DPLL is shown in Figure 2.8.

2.12.1 Optimizing DPLL

Modern SAT solvers apply several optimization on the basic DPLL backtrack search. Such optimizations are conflict based learning, conflict driven backtracking, non-chronological backtracking for empty sub-spaces, restarts and more. These optimizations result in a significant speedup of the SAT solving tools. We shall not describe these optimizations in this work. However, our All-SAT tool employs conflict based learning, and non-chronological backtracking over empty sub-spaces.

2.13 The All-SAT Problem

Given a Boolean formula presented in CNF, the *All-SAT problem* is to find all of its solutions as defined in the SAT problem.

2.13.1 Blocking by Clauses

A straightforward method to find all of the formula's solutions is to modify the DPLL SAT solving algorithm such that when a solution is found, a blocking clause describing its negation is added to the solver. This clause prevents the solver from reaching the same solution again. The last branch

```
bool satisfiability_decide() {
  while (true) {
    if (!branch())
      return SATISFIABLE
    while (bcp() == CONFLICT) {
      if (current_level==0)
        return UNSATISFAIABLE
      back-track one level
    }
  }
}
```

Figure 2.8: DPLL Backtrack Search. The function branch() chooses a value for one of the unassigned variables, or returns 'false' if all the variables are already assigned.

is then invalidated, and the search is continued with a new branch. At any given time, the solver holds the original formula, conjuncted with the negation of all of the solutions found so far. Once all the solutions are found, there is no satisfying assignment to the current formula, and the algorithm terminates.

This algorithm suffers from a rapid space growth as it adds a clause of the size of $\overline{v}$ for every solution that is found. Another problem is that the increasing number of clauses in the system slows down the *bcp()* procedure, which has to compute implications in an increasing number of clauses.

An improvement for this algorithm can be achieved by generating a clause consisting of the negation of the branching literals only. This is correct because all the rest of the assignments are implications, and are forced by the assignment of the branches. This reduces the size of the generated clauses dramatically (1-2 orders of magnitude) but does not affect their number.

Pseudo code of the blocking clauses algorithm with the described optimization is shown in Figure 2.9. Variations of this algorithm are used in [57, 12, 44], where the All-SAT problem is solved for model checking.

2.13.2 Blocking by BDDs

Another method for preventing repetitive instantiation of the same solution is by a BDD.

First we show how to constraint a SAT solver by using a BDD. Given a BDD B, defined over a subset of the variables of the SAT problem, we would like the solution found by the SAT solver to agree with B. That is, we would like the projection of the solution over the variables of B to be in B. During the SAT solving, whenever a variable which exists in B is assigned, we search B for a path from its root to the terminal node labelled '1', which

```
Blocking Clauses All-SAT:
$A \leftarrow \phi$
while (true){
  if (!branch()) {
    if (handle_solution() == EXHAUSTED)
      return A
  }
  while (bcp() == CONFLICT) {
    if (current_level==0)
      return A
    back-track one level
  }
}
```

```
handle_solution() {
  $A \leftarrow A \cup \{$current assignment$\}$
  if (current level == 0)
    return EXHAUSTED
  cl ← create a clause with the
    negation of all of the
    assignments in the branches
  add cl to the solver
  backtrack one level
  return NOT_EXHAUSTED
}
```

Figure 2.9: All SAT algorithm using blocking clauses to prevent multiple instantiations of the same solution. The procedure *bcp()* and branch() are the same as defined for DPLL in section 2.12.

agrees with the current partial assignment of the SAT solver. The existence of such a path means that the current partial assignment agrees with B. If no such path exists, the SAT solver backtracks. We call this procedure *BDD_agree(B)*, where B is the BDD which constraints the search. This procedure has to be invoked at the end of the *bcp()* procedure of DPLL, if any variable which appears in B was given an assignment.

Similarly to the blocking clauses methods, we can use a blocking BDD in order to instantiate all the solutions of a SAT problem. We initialize a BDD B to the empty set. During the search, the SAT solver executes *BDD_agree(*$\neg B$*)* at the end of each *bcp()*. Whenever a solution is found, it is added to B. The solver then backtracks, and continues the search. This way, no solution is found more than once.

The BDD representation of B is more compact than its CNF representation which is used in the blocking clauses method. This method has been presented and used in [33] and [35].

2.13.3 "Front" Based All-SAT

A different approach to solving the all-SAT problem is a "Front" Based All-SAT [31]. In this approach, a more restricted order is imposed on the branching scheme of a SAT solver, such that the solver holds the "front" of the search within the search space. This way, finding the next solution does not require storing the previous solutions that were found, and they do not have to be held within the solver. This method significantly reduces the memory requirements of the all-SAT solver, and consequently reduces its time requirements.

2.14 Circuit SAT Solvers

2.14.1 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. In 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 $\exists\ i$ such that $in_i = 0$, 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 work, we do not distinguish between an *assignment* and a *partial assignment.* We further discuss justification of assignments in Section 3.1.2.

2.14.2 Circuit SAT

A *Circuit SAT Solver* [25, 54, 42] 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.

The graph is a higher level description of the problem than CNF. Therefore, for many practical BMC instances, a circuit solver is more efficient than a CNF SAT solver. The main advantage of a circuit solver is the ability to follow the justification of the roots of the circuit as a branching heuristic. By doing so, the solver is able to make "smarter" branchings, and assign values only to the subset of the circuit nodes that is required for the justification. The rest of the nodes remain unassigned.

2.15 Dual Rail Encoding

In Section 6.2.3 we use a propositional representation of 3-valued variables and operators. This representation is then processed by a Boolean SAT solver. In order to represent 3-valued variables in a Boolean SAT solver context, we use a *dual rail* encoding. In this encoding, a ternary variable v is represented by two Boolean variables (v_h, v_l), such that $(v_h, v_l) = (0, 1) \Leftrightarrow v = 0, (v_h, v_l) = (1, 0) \Leftrightarrow v = 1$ and $(v_h, v_l) = (1, 1) \Leftrightarrow v = X$. Dual railed propositional formulae were used in [7] and [67] for STE. We assume that all the formulae throughout the rest of this work are dual rail encoded.

The expression $e = u$ is equivalent to the expression $(e \rightarrow u) \wedge (u \rightarrow e)$. For a ternary expression e, the value of the expression $e = X$ is always X, regardless of the value of e. This is because X is "unknown", and therefore it is impossible to determine if it is equal to the value of e. For e and u represented in dual rail, we say that the value of the Boolean expression $e =_b u$ is 1 if $e_h = u_h \wedge e_l = u_l$, and 0 otherwise. For example, the value of $X =_b X$ is 1. We use $=_b$ for checking if the value of a variable is X, which cannot be done with the operator $=$.

Chapter 3

Hybrid BDD and All-SAT Model Checking

In the following sections we present a hybrid, symbolic model checking algorithm for temporal safety properties, composed of both BDD and All-SAT procedures. This algorithm exploits the strengths of both BDD-based and SAT-based approaches, while trying to avoid their respective drawbacks. In addition to the common representation of the model being checked as a CNF formula or as a BDD structure, we also make use of a graph representation, which we use to prune the search space and improve performance in several ways.

We first suggest an efficient implementation of the pre-image computation using the All-SAT procedure. We then use the pre-image computation in a backward search algorithm to perform full model checking of temporal safety properties. The resulting algorithm uses BDDs for all operations, except for the pre-image computation, where the All-SAT method is used instead.

SAT-based methods for image and pre-image computation [57, 12, 31] are based on All-SAT engines, which return the set of all the solutions to a given formula (all satisfying assignments). The All-SAT engine for pre-image computation receives as input a propositional formula describing the application of a transition relation R to a set of *"next-states"* S'. The resulting set of solutions represents the pre-image of S', which is the set of all predecessors for states in S', also referred to as *"current-states"*.

Most modern SAT solvers implement the DPLL [22] backtrack search. These solvers learn and add conflict clauses to the formula in order to block searching in subspaces that are known to contain no solution. SAT solvers also implement efficient Boolean propagation procedures, as in [59]. However, when used for model checking, these algorithms do not make use of available knowledge about the structure of the model which is being checked. Thus, SAT-based model checking still suffers from the exponential complexity of search procedures that explore too many potential assignments.

We propose an All-SAT algorithm that makes use of two representations of the model's transition relation R: a propositional CNF formula and a graph of the hardware gates. We exploit the two representations for efficient search in the pre-image computation: the CNF representation is used for the usual backtrack search of SAT algorithms, whereas the graph representation is used to extract information about the structure of the design. We dynamically modify the graph representation according to the currently searched sub-space, thus exploiting more information than by static analysis of the model.

Our algorithm uses the information extracted from the structure of the model to do the following:

1. Process whole sets of next states instead of processing them one by one, unlike other All-SAT based image/pre-image algorithms [57, 12, 63, 31].

2. Each set of next-states is represented by a partial assignment to the next-state variables. The values of these variables can be justified by using only a subset of the current-state variables of the model. In every iteration of the model checking, when pre-image computation is performed, our algorithm assigns values only to variables that are required for the justification. This is done by our algorithm without computational overhead.

3. Similar to [12, 62, 53, 41, 45, 43], our algorithm uses the graph representation of the model to find partial assignments to the current state variables, instead of complete ones, thus saving time and space. However, unlike other works, the required analysis of the graph transition relation is carried on-the-fly, costing $O(1)$ operations for the branching procedure.

4. Detect independent sub-spaces and solve them independently.

5. Detect sub-spaces where solving SAT instead of All-SAT problem is sufficient.

Built on top of a SAT algorithm, our All-SAT algorithm benefits from the mechanisms incorporated in the original SAT algorithm for learning conflict clauses. Moreover, when used for pre-image computation, our All-SAT algorithm is capable of learning conflict clauses incrementally. Thus, in each iteration of the backward search model checking, the algorithm exploits the knowledge that was gained in earlier iterations.

The set of states S' is given to the All-SAT engine in the form of a BDD. Similar to [35], our All-SAT algorithm stores the negation of the solutions which were already found in a BDD structure. Consequently, both S' and

S, the input and the output of our All-SAT algorithm, consist of BDD structures only. This allows us to easily use the All-SAT algorithm in the BDD based model checking algorithm.

3.1 Hybrid BDD and All-SAT Based Pre-Image Computation

In this section we describe an algorithm for an All-SAT based pre-image computation of circuits. This algorithm is based on the DPLL backtrack search [23, 22] and uses conflict based learning of clauses. Apart from the CNF description of the problem, the algorithm uses a graph representation of the model's transition relation. The added information about the structure of the model allows finding sets of solutions instead of instantiating them one by one. Additionally, it is used to speed up the search by reducing the number of sets that are instantiated.

In Section 3.2 we describe our model checking algorithm. In order to implement it using SAT methods, we have to build an All-SAT engine which will perform the pre-image computation by solving Equation 2.1. For a circuit $C = \langle \mathcal{V}, I_0, PI, F \rangle$, we denote by $\overline{v}$ a set of Boolean variables associated with the latches in $\mathcal{V}$. Note that the variables in $\overline{v}$ are not ordered. Let S' be a given set of next-states, and S^* be the set of previously found states in a backward search. In order to perform pre-image computation, we have to find all the solutions of the formula

$$\varphi(\overline{v}) = \exists \overline{v'} \exists \overline{I} [(\overline{v'} = F(\overline{v}, \overline{I})) \wedge S'(\overline{v'}) \wedge \neg S^*(\overline{v})] \tag{3.1}$$

That is, we have to find all the assignments to $\overline{v}$ that can be extended to a solution of the formula

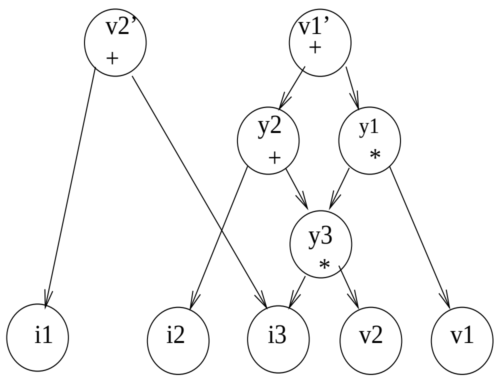
$$\varphi(\overline{v}, \overline{I}, \overline{v'}) = (\overline{v'} = F(\overline{v}, \overline{I})) \wedge S'(\overline{v'}) \wedge \neg S^*(\overline{v}) \tag{3.2}$$

Each solution for this formula represents a current-state $s(\overline{v})$, which is not in S^*, and is a predecessor of some state in S'. The fact that $s(\overline{v})$ is not in S^* implies that the algorithm finds only states which were not found before.

In our model checking algorithm, S' and S^* are given to the All-SAT algorithm as BDDs, and F is given both in CNF and as a graph (see below). The set of solutions $S(\overline{v})$ is returned by the algorithm as a BDD.

3.1.1 Propositional Representation of TRGs

Given a hardware circuit, our algorithm uses its *transition relation graph* (TRG), and requires a propositional reperesntation of it. this representation is constructed as described in [5]. Each subexpression of the transition



(a) TRG

$(y_3 = v_2 \wedge i_3) \equiv$
$(y_3 \vee \neg v_2 \vee \neg i_3) \wedge$
$(\neg y_3 \vee v_2) \wedge$
$(\neg y_3 \vee i_3)$

(b) CNF

Figure 3.1: Transition Relation Graph. (a) The TRG corresponding to the transition relation given in Figure 3.2(a). $\overline{v} = \{v_1, v_2\}$, $\overline{I} = \{i_1, i_2, i_3\}$ and $\overline{v'} = \{v'_1, v'_2\}$. y_1, y_2 and y_3 are auxiliary variables representing the subexpressions of the transition relation. (b) The CNF description for the operator of node y_3.

relation function is associated with an auxiliary variable, which represents it in a CNF formula. These variables correspond to the variables of the internal nodes in the TRG. Thus, for each node in the TRG, the CNF representation of the transition relation has a corresponding variable, and clauses which describe the node's Boolean operation. For example, in the TRG shown in Figure 3.1(a), v_3 is associated with the Boolean operation 'And'. In Figure 3.1(b) we show the clauses corresponding v_3.

3.1.2 Justification of Assignments

As mentioned above, the transition relation of a circuit is a set of functions F, such that for each next-state variable $v'_i \in \overline{v'}$, $v'_i = f_i(\overline{v}, \overline{I})$. For a next-state variable v'_i, f_i depends only on $\widetilde{v_i} \subseteq \overline{v}$ and $\widetilde{I_i} \subseteq \overline{I}$, subsets of the state variables and the inputs to the model, respectively. An example for such a transition relation is given in Figure 3.2(a).

For $v'_i \in \overline{v'}$, the *support* set of v'_i, $supp_{v'_i}$, is:

$$supp_{v'_i} = \widetilde{v_i} \cup \widetilde{I_i}$$

In Figure 3.2(b) we show the support sets of the variables defined in Figure 3.2(a). For an assignment a to v'_i, and an assignment b to $supp_{v'_i}$, we say that

$$a \text{ is } justified \text{ by } b \leftrightarrow a(v_i) = f_i(b(supp_{v'_i}))$$

We generalize this definition for A, an assignment to $\overline{v'}$, and B, an assign-

$$v'_1 = (v_1 \wedge (v_2 \wedge i_3)) \vee ((v_2 \wedge i_3) \vee i_2) \qquad supp_{v'_1} = \{v_1, v_2, i_2, i_3\}$$
$$v'_2 = i_3 \vee i_1 \qquad supp_{v'_2} = \{i_1, i_3\}$$

(a) Transition Relation (b) Support Sets

$$A = \{v'_1, v'_2\} \qquad A = \{v'_1, v'_2\}$$
$$B_1 = \{\neg v_1, v_2, \neg i_1, \neg i_2, i_3\} \qquad B'_1 = \{v_2, i_3\}$$
$$B_2 = \{\neg v_1, v_2, i_1, i_2, \neg i_3\} \qquad B'_2 = \{\neg v_1, v_2, i_1, i_2\}$$
$$B_3 = \{\neg v_1, \neg v_2, i_1, \neg i_2, i_3\} \qquad B'_3 = \{v_2\}$$

(c) Justifying Assignments (d) Maximally Justifying Assignments

Figure 3.2: (a) Partitioned Transition Relation. v'_1 and v'_2 are given as functions of v_1, v_2, i_1, i_2, i_3. (b) The support set of a variable is that set of variables in the function which defines its value. (c) B_1 and B_2 *justify* A. B_3 does not. (d) B'_1 and B'_2 are partial assignments that *justify* A. B'_1 *maximally justifies* A.

ment to $\overline{v} \cup \overline{I}$.

$$A \text{ is } justified \text{ by } B \leftrightarrow \forall v'_i \in \overline{v'},\ A(v'_i) = f_i(B(supp_{v'_i}))$$

In Figure 3.2(c), B_1 and B_2 justify A, and B_3 does not justify it.

A partial assignment b' to $\widehat{supp}_{v'_i} \subseteq supp_{v'_i}$ represents all the assignments to $supp_{v'_i}$ which agree with b' on the assignment to $\widehat{supp}_{v'_i}$. b' justifies an assignment a to v'_i if all the assignments that it represents justify a. In Figure 3.2(d), B'_1 and B'_2 are partial assignments that justify A. B'_3 is a partial assignment which does not justify A. We say that a partial assignment b' *maximally justifies* a if b' justifies a, and for every variable which is assigned by b', removing it from b' will make b' not justify a anymore. Note that such an assignment is not unique. In Figure 3.2(d), B'_1 maximally justifies A, and B'_2 does not.

Recall that we are looking for all the assignments to $\overline{v}$ which can be extended to a solution (a satisfying assignment) of the formula $\overline{v'} = F(\overline{v}, \overline{I}) \wedge S'(\overline{v'}) \wedge \neg S^*(\overline{v})$ (see Equation 3.2). We are actually looking for all the assignments to $\overline{v}$, for which there is an assignment to $\overline{I}$ such that they justify some assignment to $\overline{v'}$, which is in S'. The assignments to $\overline{v}$ should also not conflict with $\neg S^*$. We discuss the last constraint in section 3.1.5.

We find these assignments to $\overline{v}$ by finding all the justifying assignments for all the assignments in S'. Each such justifying assignment is an assignment to $\overline{v} \cup \overline{I}$. However, we only look for assignments that differ in values of $\overline{v}$.

We first describe a SAT based algorithm for finding a single maximally justifying assignment to $\overline{v} \cup \overline{I}$, for a given assignment in S'. We then describe an All-SAT based algorithm which finds *all the maximally justifying assign-*

ments to $\overline{v} \cup \overline{I}$ that differ on the assignment to $\overline{v}$, for a single assignment in S'. Finally, we extend it to find all the maximally justifying assignments to $\overline{v} \cup \overline{I}$ that differ on their assignment to $\overline{v}$, for *all of the assignments* in S'.

3.1.3 Maximally Justifying a Given Assignment to $\overline{v'}$

We incorporate the TRG into a SAT based search algorithm, along with the CNF representation of the transition relation. In this section we describe a SAT based algorithm which uses the TRG for pruning the search space for a SAT engine, such that it will find a maximally justifying assignment to $\overline{v} \cup \overline{I}$, for a given assignment to $\overline{v'}$. The assignment to $\overline{v'}$ is given as an initial partial assignment to the SAT solver before the search begins.

We introduce a new branching procedure into a SAT solver. This procedure uses the TRG to choose the next variable to assign as part of the regular DPLL. The branching procedure also detects that a solution to the problem is found. The rest of the search algorithm is not changed, and uses the CNF representation of the problem. In particular, conflict based learning, and non-chronological backtracking for empty subspaces, are performed as in regular SAT solving. Furthermore, the efficient *bcp()* procedure of the SAT solver is used. Throughout the rest of this work, we use the term *backtrack* to refer to the backtrack of the SAT algorithm in the search tree, as opposed to the operations performed on the TRG or BDDs.

Our new branching procedure is a variation of [24]. It performs a traversal of the TRG, assigning the variables with values in a pre-order manner. Each branch is taken such that it justifies the previous one, until the traversal is completed.

Consider a node v_1 in the TRG, and its successors $y_1 \ldots y_n$. Assume v_1 is associated with the Boolean operator $op \in \{AND, OR, NAND, NOR, NOT\}$, and assume that it was assigned with the Boolean value b at decision level i. According to op and b, there are two possibilities for justifying v_1

- The values of $y_1 \ldots y_n$ are all implied by b at decision level i. For example, $op = AND$ and $b = 1$ imply that $y_1 \ldots y_n$ are 1.

  In that case, v_1 is already justified by its sons at decision level i, and the next branch will be chosen such that it justifies y_1. After completing the traversal of the subgraph of y_1, the branching procedure will move on to justify y_2, and so on.

  We demonstrate this case in Figure 3.3(a). The branch at level i was v_1. This branch implies $y_1 \ldots y_5$. The implications are calculated by the *bcp()* procedure of the SAT algorithm. Since the value v_1 is already justified by its sons, the next branch, at level $i+1$, is z_1, which justifies y_1.

- None of the values of $y_1 \dots y_n$ is implied by b at decision level i, and it is enough to give a value to one of $y_1 \dots y_n$ in order to justify v_1. For example, if $op = AND$ and $b = 0$ then it is enough to give one of the nodes $y_1 \dots y_n$ the value 0 in order to justify b.

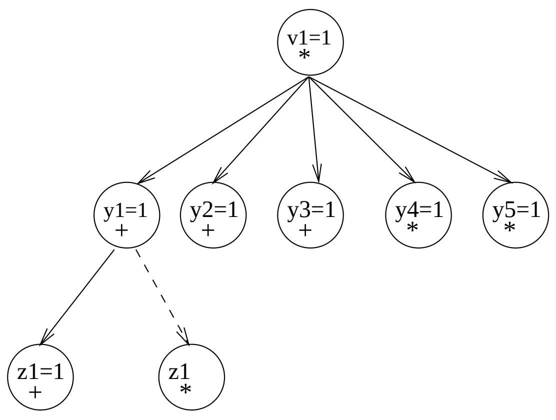
  In that case, the branching procedure assigns y_1 with the appropriate value in decision level $i+1$. The branch in decision level $i+2$ is made such that it justifies the assignment to y_1, and so on. $y_2 \dots y_n$ are not required for justifying v_1, and thus do not have to be assigned with a value. Therefore, they are disconnected from v_1, and do not take part in the rest of the traversal of the TRG. We demonstrate this case in Figure 3.3(b). The branch at level i was $\neg v_1$. The next branch, at level $i+1$ is $\neg y_1$, which is enough in order to justify $\neg v_1$. Therefore, we can remove the edges to y_2 through y_5 from the TRG. When a new branch is required at level $i+2$, we continue the traversal of the TRG over the connected successors only, trying to justify $\neg y_1$.

  When backtracking in the SAT search space, we also go backwards in the traversal of the TRG. If we cannot justify $\neg y_1$, and backtrack to decision level $i+1$, we change the value of y_1 as in regular DPLL. In our example, y_1 will assume the value 1. The new branch at level $i+1$ will be $\neg y_2$, thus justifying $\neg v_1$ by the next successor. This is shown in Figure 3.3(c).

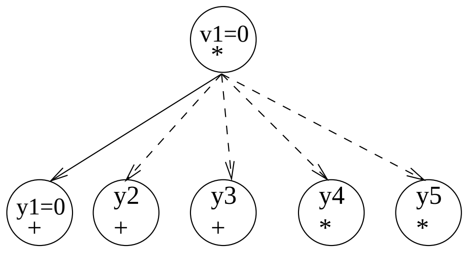
We conclude that a justifying assignment for v_1 was found when we complete the traversal of its subgraph. Note that due to disconnection of nodes during the process, we do not necessarily traverse all the nodes in the subgraph.

Recall that we are trying to find a justifying assignment to a given assignment to $\overline{v'}$. Each $v' \in \overline{v'}$ is a root in the TRG. By justifying all the nodes $v' \in \overline{v'}$, we get a justifying assignment to $\overline{v'}$. The result of this is that instead of satisfying all the clauses of the CNF formula, or assigning values to all of the variables in the TRG, as in regular SAT procedures, we only assign variables that are actually required for the justification. The returned result is the partial assignment to $\overline{v} \cup \overline{I}$, obtained at the termination of the algorithm.

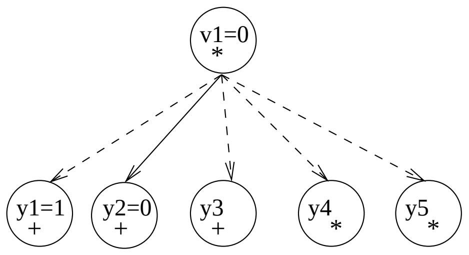
Consider a partial assignment to $\overline{v} \cup \overline{I}$ returned by the algorithm. The branching procedure only traverses parts of the TRG that are required to justify the assignment to $\overline{v'}$. Therefore, each value in the partial assignment takes part in the justification, and removing it would make the partial assignment not justifying the assignment to $\overline{v'}$ anymore. Therefore, this partial assignment maximally justifies the assignment to $\overline{v'}$, and thus we have found the required result.



(a) No Branch

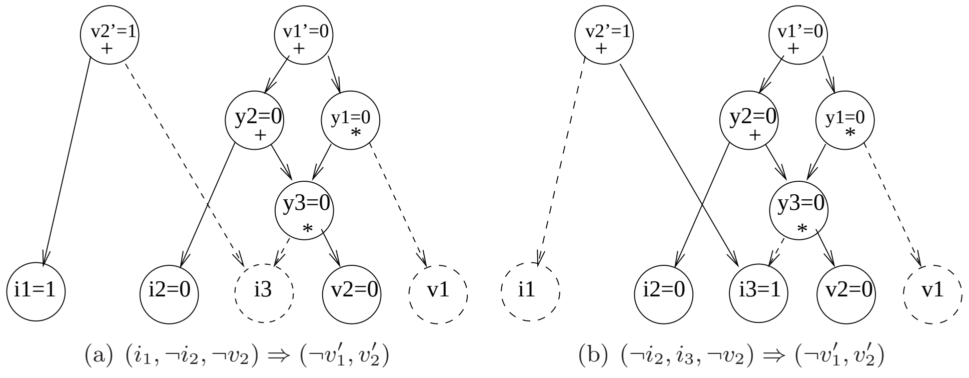


(b) 1^{st} Branch



(c) 2^{nd} Branch

Figure 3.3: Branching over an 'AND' gate. (a) The current assignment v_1 implies $y_1 \dots y_5$. Therefore, no branching over these variables is made. The next branch is then z_1, which justifies y_1. (b) The assignment so far is $\neg v_1$. The branch $\neg y_1$ justifies this assignment. Therefore, the edges to $y_2 \dots y_5$ can be removed. (c) After backtracking, $y_1 = 1$, because all of the solutions that include $y_1 = 0$ were already found. This value to y_1 does not justify $\neg v_1$, and therefore does not have to be justified. The next branch is $\neg y_2$, and it justifies the assignment $\neg v_1$. Again, the other edges can be removed.



v2'=1
+
v1'=0
+
y2=0
+
y1=0
*
y3=0
*
i1=1
i2=0
i3
v2=0
v1
(a) $(i_1, \neg i_2, \neg v_2) \Rightarrow (\neg v'_1, v'_2)$
v2'=1
+
v1'=0
+
y2=0
+
y1=0
*
y3=0
*
i1
i2=0
i3=1
v2=0
v1
(b) $(\neg i_2, i_3, \neg v_2) \Rightarrow (\neg v'_1, v'_2)$

Figure 3.4: Justifying Assignments. Two maximally justifying assignments which differ only in the assignments to $\overline{I}$. The dotted edges are those that were removed from the TRG by the branching procedure, and the dotted nodes are the variables for which a value is not instantiated.

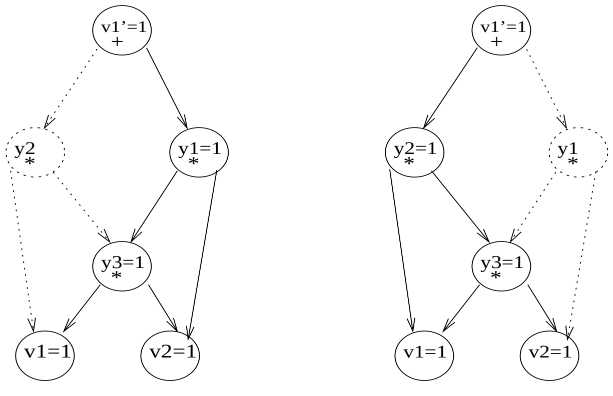
3.1.4 All Justifications for a Given Assignment to $\overline{v'}$

We define a solution for a given assignment a' to $\overline{v'}$ as an assignment to $\overline{v}$, for which there is some assignment to $\overline{I}$, such that they justify a'. In this section we describe an algorithm for finding all the solutions for a given a'.

Note that every extension of a maximally justifying assignment over all the variables in $\overline{v}$ is a solution for a'. Thus, a maximally justifying assignments actually represents a set of solutions to a'.

We build an All-SAT algorithm on top of the SAT algorithm presented in the previous section, in order to find all the maximally justifying assignments to $\overline{v}$, and thus find all the solutions to our problem. This algorithm uses a blocking BDD $S^{not}(\overline{v})$, as in [35], which is initialized to 1. When a justifying assignment is found by the SAT algorithm, we consider its projection a over $\overline{v}$. The conjunction of the literals in a is negated, and then conjuncted with $S^{not}(\overline{v})$. The SAT algorithm backtracks one level, and the search is resumed. We use the procedure $BDD\_agree(S^{not}(\overline{v}))$ in order to check the solutions during their creation, making sure that they agree with S^{not}, meaning that they were not produced before. Thus, this procedure blocks the solver from finding the same partial assignment to $\overline{v}$ again. The algorithm terminates when no new solution is found.

Next we explain the reason for using only the projections of the solutions over $\overline{v}$ in the blocking BDD. Each maximally justifying assignment which is found by the algorithm is a partial assignment to $\overline{v} \cup \overline{I}$. This poses the following problem: Multiple maximally justifying assignments might give the same assignments to the variables in $\overline{v}$, and differ in the assignment to variables in $\overline{I}$ only. This is demonstrated in Figure 3.4. Moreover, we might reach the same maximally justifying assignment by choosing different values for the internal nodes in the TRG. This is demonstrated in Figure 3.5.



(a) $(v_1, x_v) \Rightarrow v_1'$ (b) $(v_1, v_2) \Rightarrow v_1'$

Figure 3.5: Two maximally justifying assignments corresponding to different assignments for the internal nodes. The dotted parts of the graph are those cut by the branching procedure.

However, we are interested in all the maximally justifying assignments which differ on their assignment to $\overline{v}$. Therefore, we use only the assignments to variables in $\overline{v}$ for blocking future solutions.

Note that S^{not} is defined over $\overline{v}$, which constitute about 10% of the total number of variables [76]. This reduces the overhead of using *BDD_agree(S^{not})*. Note, also, that by constructing S^{not}, we actually get the BDD $S(\overline{v})$ of the solutions to the problem. This is because $S = \neg S^{not}$, which is obtained by an O(1) operation. Thus, the algorithm results in $S(\overline{v})$, as required.
This algorithm is given in Figure 3.6.

3.1.5 Justifying All the Assignments to $\overline{v'}$

In the previous section we showed how to find all the solutions for a given assignment to the next-state variables. In this section we find all the solutions to all of the partial assignments represented by a given BDD $S'(\overline{v'})$. The All-SAT algorithm then returns all the maximally satisfying assignments for all the assignments in S', which is the solution to our problem.

Note that the branching procedures that were presented in the previous sections are applicable also to partial assignments to $\overline{v'}$.

We exploit the fact that $S'(\overline{v'})$ is given as a BDD. The assignments to $\overline{v'}$ are represented by paths from the root of the BDD to the terminal node '1'. In Figure 3.7(a,b) we show a BDD and its corresponding assignments. For a path π in the BDD of $S'(\overline{v'})$, $U(\pi)$, the set of variables in π, is a subset of $\overline{v'}$. We can refer to π as an assignment to $U(\pi)$, where a variable $u \in U(\pi)$ is assigned 1 if π includes the edge to the "right" son of u, and 0 otherwise.

By finding all the maximally justifying assignments to every π in the BDD of S', we find representations for all the justifications to all of the

```
Graph All-SAT:
$S^{not}(\overline{v}) \leftarrow '1'$
while (true){
  if (!branch()) {
    if (handle_solution() == EXHAUSTED)
      return $\neg S^{not}(\overline{v})$
  }
  while (bcp() == CONFLICT) {
    if (current_level==0)
      return $\neg S^{not}(\overline{v})$
    backtrack one level
  }
}
```

```
handle_solution() {
  $S^{not}(\overline{v}) \leftarrow S^{not}(\overline{v}) \cap$
    $\{\neg$(current assignment to $\overline{v})\}$
  if (current level == 0)
    return EXHAUSTED
  backtrack one level
  return NOT_EXHAUSTED
}
```

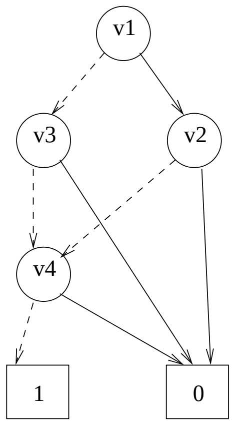
Figure 3.6: All SAT algorithm using a blocking BDD. The procedure *bcp()* also calls $BDD\_agree(S^{not}(\overline{v}))$ if a variable from $\overline{v}$ is assigned, and returns CONFLICT if either the boolean propagation resulted in a conflict, or if $BDD\_agree(S^{not}(\overline{v}))$ finds a conflict between the current partial assignment and the BDD S^{not}. When the computation is completed, the set of solutions is returned by computing $\neg(S^{not})$, which is an O(1) operation.

assignments in S'. We now show how to introduce all the paths in S' into the All-SAT solver.

If we reverse the direction of the edges in S', and traverse it in a DFS-like manner, starting from the terminal node '1', we will get all the paths from the root of S' to the terminal node '1' in reverse order. The value of a node on the path is '1' if we reached it from its 'right' successor, and '0' otherwise. A straightforward approach for finding the maximally justifying assignments for the paths in the BDD is to apply the algorithm given in Section 3.1.4 for each π found in the DFS. However, this may cause unnecessary duplicated work. For example, consider two assignments to $\overline{v'}$, $\pi_1 = \{v'_1, v'_2, v'_3\}$ and $\pi_2 = \{v'_1, v'_2, \neg v'_4\}$. Applying this approach over these assignment would require justifying v'_1 and v'_2 twice, once for each assignment to $\overline{v'}$.

In order to avoid the repetition, we integrate the DFS in the BDD into the All-SAT branching procedure. At decision level $i + 1$, if the branch in level i is not justified yet, the branching procedure uses the TRG for choosing an assignment to a variable that will justify the branch at level i. This is done as described in the Section 3.1.3. Otherwise, if there is no unjustified branch, then a value for one of the (unassigned yet) roots of the TRG is chosen, based on the DFS over the BDD. When backtracking in the All-SAT procedure, we also backtrack in the DFS over the BDD of S' respectively. Thus, a new path in the BDD is explored, and a new assignment to $\overline{(}v')$ is justified. We demonstrate this procedure in Figure 3.7(c,d). The pseudo code for the branching procedure is given in Figure 3.8.

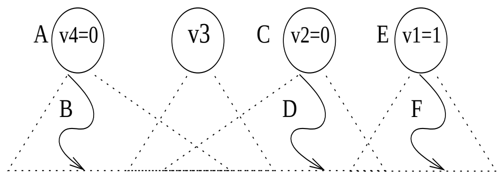
By integrating the DFS with the branching procedure, the All-SAT algo-



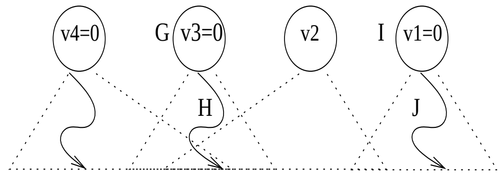
(a) BDD

$$\begin{array}{ll} \pi_1 :\ \neg v_4 \neg v_2, v_1 & \left\{ \begin{array}{l} a_{1,1} :\ \neg v_4, \neg v_3, \neg v_2, v_1 \\ a_{1,2} :\ \neg v_4, v_3, \neg v_2, v_1 \end{array} \right. \\ \pi_2 :\ \neg v_4, \neg v_3, \neg v_1 & \left\{ \begin{array}{l} a_{2,1} :\ \neg v_4, \neg v_3, \neg v_2, \neg v_1 \\ a_{2,2} :\ \neg v_4, \neg v_3, v_2, \neg v_1 \end{array} \right. \end{array}$$

(b) Paths



(c) Branching Sequence



(d) Branching Sequence

Figure 3.7: (a) A BDD representing $S'(\overline{v'})$. Solid lines represent the 'right' successor of a node, while dashed lines represent the 'left' successor of a node. (b) The assignments corresponding to the paths from the root to the terminal node '1'. (c) A,B...F are the branching sequence of the All-SAT. Branches A,C, and E are imposed on the All-SAT engine by the DFS over the BDD. B,D and F represent the branches which are taken for justifying A,C and E respectively. (d) After we backtrack, the branch G corresponds to the DFS search, followed by branching sequences H,I and J. The justification of $\neg v_4$ is performed only once.

```
branch_AND_node():
$n \leftarrow P_{TRG}$
if ($\neg justified$(n)) {                      - $n$ has to be justified
  $n' \leftarrow$ next son($n$)
  $P_{TRG} \leftarrow n'$                          - Point to next son of $n$
  if (value($n$) == 1)                          - $n'$ already implied by $n$
    return branch()                             - Recursive call: justify $n'$
  else                                          - Sons of $n$ are not implied by $\neg n$
    return (value($n'$) $\leftarrow$ 0)                  - Justify $\neg n$ by $\neg n'$
} else {                                        - $n$ already justified
  $P_{TRG} \leftarrow$ next node traversing the TRG    - Continue the DFS
  if ($P_{TRG}$ != $null$) {                         - Next node in DFS is not null
    return branch()                             - Justify the next node
  } else {                                      - Current root justified
    $P_{BDD} \leftarrow$ next node in DFS of the BDD  - Ask BDD for a new root
    if ($P_{BDD}$ == $null$) {                       - No more roots to justify
      return null                               - Solution found
    } else {                                    - BDD returned a root to justify
      $P_{TRG} \leftarrow P_{BDD}$                        - Point to the new root
      return branch()                           - Justify the new root
    }
  }
}
```

Figure 3.8: Branching Procedure over an AND node. P_{TRG} is a pointer for the traversal of the TRG, initialized to null. This algorithm describes the branching procedure when P_{TRG} points to a node which is associated with AND operator. The procedure *branch()* calls the specific branching procedure according the the node pointed by P_{TRG}. P_{BDD} is a pointer for the DFS over the BDD, initialized to the terminal node '1'. The procedure traverses the TRG and the BDD of S' as described in Section 3.1.5, and updates P_{TRG} and P_{BDD} accordingly. The procedure returns a value to some variable, or null, if no further justification is required, and a justifying assignment was found.

rithm assigns $\overline{v'}$ with every assignment corresponding to some π in S', and them only. Since π corresponds to a partial assignment, some of the TRG roots will not be assigned and therefore will not have to be justified. As a result, only the parts of the TRG which are reachable from the assigned roots are traversed, and work is saved. This is similar to using dynamic transition relation, as explained in the next section, with finer resolution. Altogether, the All-SAT algorithm receives partial assignments, representing subsets of next-states (from S') and returns partial assignments, representing subsets of current-states.

Blocking $S^*(\overline{v})$

Recall that we would like the solutions found by our search engine to agree with $\neg S^*(\overline{v})$, as defined in Equation (3.2). $\neg S^*(\overline{v})$ is given to our solver as a BDD. Therefore, we use $BDD\_agree(\neg S^*(\overline{v}))$ to constraint the All-SAT engine. Thus, all the assignments generated by the All-SAT engine agree with $\neg S^*(\overline{v})$ and the requirement is met .

3.1.6 Optimizations

The incorporation of the TRG into the All-SAT solver allows us to apply additional optimizations on the search. In this section we describe how the information extracted from the TRG is used in order to detect independent subproblems, and to reduce All-SAT subproblems to SAT problems.

Independent roots

For a node v in the TRG, and for a subgraph of the TRG, G, we say that v is the root of G if G consists of exactly v and all of its descendants. We say that v is an *Independent Root* of G, if v is the root of G, and all the paths from outside of G into G pass through v. That is, For every v_1, an ancestor of a vertex v_2 in G, v_1 is either a descendant of v, or v is on all the paths from v_1 to v_2. This property can be decided statically before starting the All-SAT solving process. However, when using the TRG for branching during the search, we disconnect nodes that are not required for justifying the current partial assignment. When backtracking in the All-SAT procedure, we reconnect these nodes. Therefore, independent roots should also be detected dynamically during the solving process. Note that the terminal nodes in the TRG are, by their definition, independent roots. Given an assignment to the root v of subgraph G, a *solution* for v in G is an assignment to the terminal nodes in G, which maximally justifies the assignment to v. In Figure 3.10(a), only the terminal nodes are independent roots.

For a node v which is an independent root of a subgraph G, all the nodes in $supp_v$ are in the cone of influence of no other node. Therefore, the assignment to these nodes effect v only. For a given assignment to v, finding all the justification for it is independent of the rest of the nodes in the TRG. Therefore, when we reach an independent root, we can solve the All-SAT problem of its corresponding subgraph independently of the rest of the TRG. The solutions of the complete problem are the product of the partial solutions. In Figure 3.10(b), the edge from v_2' to i_3 is disconnected, and v_3 becomes an independent root. Thus, for any assignment to v_3, i_3 and v_2 should be assigned so that they justify it, regardless of the assignments to the other variables.

A node which is found to be an independent root by the static analysis of the TRG, is always an independent root, regardless of the current partial assignment. On the other hand, a node which was dynamically found to be an independent root, is only independent in the context of the current partial assignment to the variables. This is demonstrated in Figure 3.10(a,b,c).

The algorithm in Figure 3.9 statically determines if a node v_0, a root of a subgraph G, is an independent root. The algorithm performs a DFS over G, starting from v_0. For each node $v \in G$, the algorithm counts the number of edges into v which are reachable from v_0. The *score* of a node v is the number of edges into v, and into all of its descendants, which are not reachable from v_0. Thus, the score of v_0 is the sum of all the edges into G which are not reachable from v_0. If the score of v_0 is 0, then no node in G has an edge not reachable from v_0, and v_0 is an independent node. For complete static analysis, we have to apply this algorithm on all the nodes in the TRG. This is an $O(|TRG|^2)$ operation, which has to be done once, prior to the solving process.

In order to dynamically find independent roots, we dynamically change the score which was calculated in the static analysis. During the solving process, edges are removed from the TRG, or added to it, in correspondence to the current partial assignment. When an edge into a node v is removed, or put back, we decrease/increase the score of v respectively, and notify all the predecessors of v. Each node which is notified updates its score, and notifies its predecessors, until all the ancestors of v are notified. A node updates its score no more than once for each change in the graph, even if it is notified of it by more than one decedent. This operation involves all the ancestors of v.

Note that there is a trade-off between the time spent on dynamically detecting independent roots and the time saved on solving subproblems independently. We elaborate on this in Section 3.3.

Non-Important Roots

We say that a node v in the TRG is a *Non-Important Root* if it is an independent root of a sub-graph G, where all the terminal nodes are input variables (in $\overline{I}$). As with independent roots, this property can be decided statically or dynamically. In Figure 3.10(a), only the terminal nodes of the input variables are non-important roots. In Figure 3.10(b), v_3 is also a

```
$Is\_independent\_root(v_0, G)$ :
$\forall v \in G$
    $v.edges \leftarrow$ #predecessors of $v$
    $v.visited \leftarrow false$
$v_0.edges \leftarrow 0$
$dec\_ante\_edges(v_0)$
$\forall v \in G$
    $v.score \leftarrow 0$
    $v.visited \leftarrow false$
$sum\_scores(v_0)$
if ($v_0.score == 0$)
      return TRUE
else
      return FALSE
```

```
$dec\_ante\_edges(v)$
  $v.edges \leftarrow v.edges - 1$
  $if(v.visited == true)$
    $return$
  $v.visited = true$
  $\forall u,\ direcet\ descendent\ of\ v$
    $dec\_ante\_edges(u)$
```

```
$sum\_scores(v)$
  $if(v.visited == true)$
    $return\ v.score$
  $v.visited = true$
  $\forall u,\ direcet\ descendent\ of\ v$
  $v.score \leftarrow v.score +$
              $sum\_scores(u)$
  $return\ v.score$
```

Figure 3.9: Algorithm for static analysis of independent roots. For a node v_0, a root of a subgraph G, the value of $v_0.score$ at the end of the execution is the number of edges from outside G into G, not reachable from v_0. If $v_0.score = 0$, then v_0 is an independent root.

non-important root.

For a node v, which is a non-important root of a subgraph G, v is, by its definition, an independent root. Therefore we can solve the All-SAT problem of G independently on the rest of the TRG. Moreover, the solutions returned by our All-SAT algorithm are defined as all the partial assignments to $\overline{v}$ for which there is some assignment to $\overline{I}$ such that they justify the current assignment to $\overline{v'}$. All the terminal nodes in G are in $\overline{I}$, and thus are not a part of any of the solutions. Therefore, we only have to find one assignment to the terminal nodes of G which justify the current value of v. Consequently, it is enough to solve the SAT problem, rather than All-SAT, for G, with the current value of v.

A node v, found to be a non-important root of a subgraph G by the static analysis of the TRG, is always a non-important root, regardless of the current assignment to the rest of the variables. Therefore, we can solve the SAT problem for G to justify the values 'true' and 'false' for v only once, on the first time that they are required, store the result, and reuse it when v is reached again.

The static detection of non-important roots is straightforward. For each node v in the TRG, we perform a DFS from v, counting $v.r\_count$, the number of current-state nodes *reachable* from v. If v is an independent root, and $v.r\_count = 0$, then v is a non-important root. This is an $O(|TRG|^2)$ operation, which has to be performed once, at the beginning of the checking

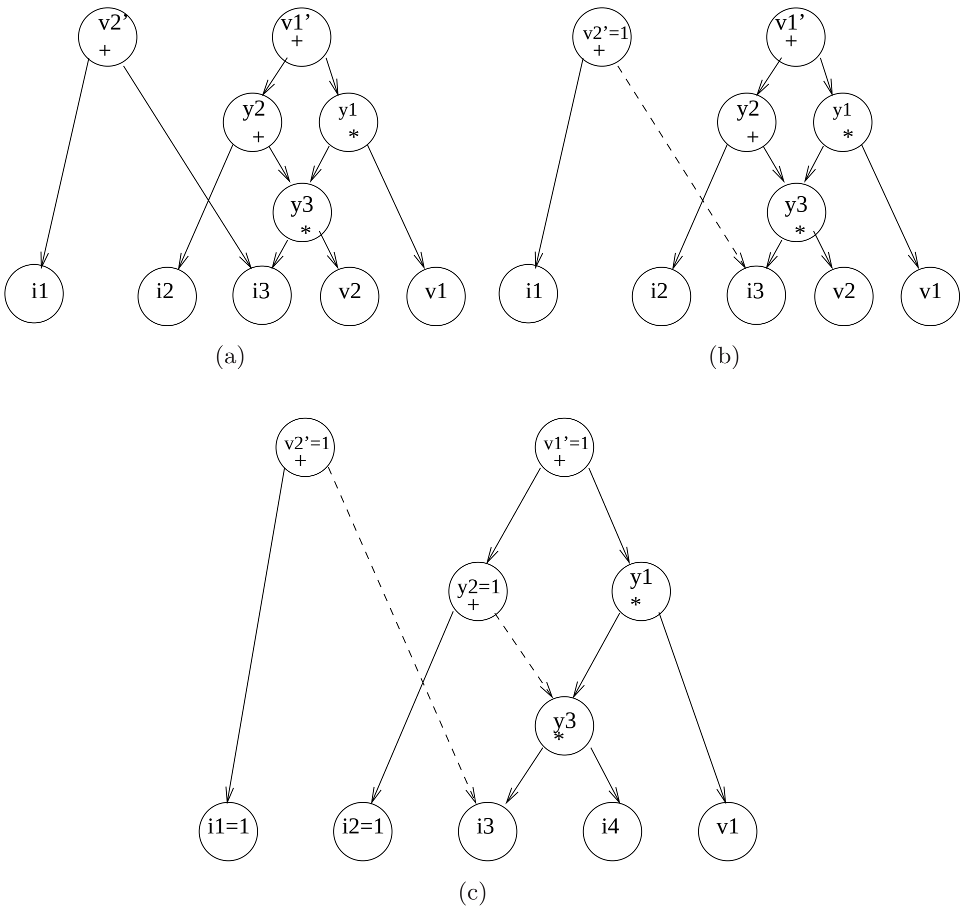


Figure 3.10: Independent and Non-Important Roots. (a) Only the terminal nodes are independent roots. The terminal nodes of input variables are the only non-important roots (b) After the assignment to v_2' and i_1, v_1' and v_3 are also independent roots. v_3 is also a non-important root. (c) The additional assignments to v_1', v_2 and i_2 make v_1 also an independent root.

process.

We dynamically modify *r_count*, which was calculated in the static analysis, for dynamic detection of non-important roots. During the solving process, if a current-state variable is given an assignment, we notify all its ancestors to decrease their *r_count*, as with the independent roots. When the All-SAT algorithm backtracks, and nullifies an assignment to a current-state variable, we notify its ancestors to increase their r_count. At any given moment, an independent root v for which $v.r\_count = 0$ is a non-important root. As with independent roots, there is a trade-off between the time spent on dynamically detecting non-important roots and the time saved on solving SAT instead of All-SAT subproblems.

From the implementation point of view, given an independent subgraph of the TRG, G, solving the SAT and the All-SAT problems for G does not require any duplication of it outside of the TRG, and can be performed in the context of the global All-SAT solver. Since G is independent of the rest of the TRG, assigning values to the variables in G does not effect, and is not effected by the rest of the variables of the problem. A solution in G is identified by our TRG based branching procedure, and therefore also does not involve the variables outside of G. This way, we avoid wasting time and memory for replicating the subproblem of G for solving it independently.

3.2 Model Checking Using Hybrid BDD and All-SAT Pre-Image Computation

In this section we show how our hybrid pre-image computation algorithm can be used to enhance the performance of the model checking algorithm.

In Section 2.4 we discuss a backward search algorithm for checking safety properties. The pseudo code of this algorithm is given in Figure 2.2. When this algorithm is implemented with BDDs, its bottleneck is the pre-image computation performed in line 7. This is because intermediate results of quantification might be one or two orders of magnitude larger than the initial and resulting BDDs.

In order to avoid the bottleneck of pre-image computation, we replace the BDD-based image computation with our hybrid pre-image computation, presented in Section 3.1. Refer to the algorithm in Figure 2.2. We represent $S_0, \neg P, S^*$ and *new* as BDDs. The operations in lines $3, 4$ and 6 of the algorithm are performed directly on their BDD representation. The pre-image operation, on the other hand, is performed with the hybrid All-SAT algorithm, which solves Equation 3.2, in which S' is replaced with *new*. Thus, pre-image computation is now all-SAT based, and does not depend on quantification in BDDs.

We describe two additional advantages to our new algorithm, apart from avoiding quantification in BDDs: using dynamic transition relation, and dynamic learning.

3.2.1 Dynamic Transition Relation

A set $S'(\overline{x'})$ is defined by a set of partial assignments to some of the state variables $\overline{y} \subseteq \overline{x'}$. In order to apply a pre-image to S', only a subset of formulae of the partitioned transition relation $T^{\overline{y}} \subseteq T$ should be involved in the computation. This subset includes those functions that define the next values for the variables in $\overline{y}$. The reduced transition relation is called a *dynamic transition relation*, since it is defined dynamically for each S'. Dynamic transition relations are used as an optimization in BDD-based pre-image computation.

We further enhance this method by using the TRG and the BDD of S' for branching in the All-SAT process. For each path π in the BDD of S', our branching procedure considers only the subset of T which is necessary for justifying the assignment induced by π. Thus, we actually use different parts of the transition relation for different subsets of states in S'. This is achieved without computational overhead.

3.2.2 Incremental Learning

For pre-image computation, we apply our All-SAT algorithm on F, S' (where $S' = new$) and S^*. Only F is given to the algorithm as a CNF formula. Therefore, the same CNF formula is given to the SAT solver in every iteration. As a result, clauses that were deduced by conflict based learning in one iteration, can be used in subsequent ones, thus contributing to the speedup of the All-SAT solver.

3.3 Experimental Results and Conclusions

We implemented our algorithm on top of the zChaff SAT solver [59], and CUDD BDD library [77]. zChaff is a state of the art SAT solver, known to be one of the fastest solvers available. CUDD is a BDD library, common in many BDD based applications. Both zChaff and CUDD are open source tools, which allowed us to interface them with the graph representation of the transition relation.

There are many optimizations common in model checking that are not incorporated in our prototype implementation. Thus, for the sake of a fair comparison, we also implemented a model checking framework that is based on BDD backward search. This framework uses partitioned transition relation, and computes the cone of influence for the properties that are checked.

For experiments we used ISCAS89 and ISCAS99 benchmarks, which are large “real-life” problems. We checked each model against a set of properties of the form AGp, using the hybrid and the BDD based tools. All experiments use dedicated computers with 1.7Ghz Intel Xeon CPU, and 1GB RAM, running Linux operating system. Timeout for each property was set to 24 hours.

The results of our experiments are presented in Table 3.1, and continues in Table 3.2. In order to mask initialization effects, we omit the results for the smaller models which take only a few seconds to solve. For each model and property, the tables show the number of iterations of the backward search which were completed by each tool, until either the checking is completed, timeout is reached, or memory out is reached. The tables also show the computation time for each completed check, the memory usage, and the percentage of the run time that was spent on quantification.

While the hybrid tool does not consistently outperform the BDD model checker, in half of the models it does better than the BDD tool for almost all the properties checked. In some cases where checking is not completed, the hybrid tool is able to perform the same number of pre-image iterations faster, or even perform more pre-image iterations. In most of the cases where none of the algorithms outperformed the other, the hybrid algorithm required less memory than the BDD algorithm.

For many of the models, the BDD tool suffers memory exhaustion while the hybrid tool continues until timeout, and even succeeds to complete additional iterations. This demonstrates the inherent space problem of BDDs that is discussed in Section 2.10: when performing quantification, intermediate results of a single pre-image computation commonly blow up to an order of magnitude larger than the eventual BDD size [39]. In contrast, the hybrid tool uses the All-SAT engine for quantification, and thus memory blowup does not become an issue, except for one model.

The table shows a strong correlation between the models for which the hybrid algorithm is faster, and the models for which it requires less memory than the BDD algorithm. This is rather than a trade-off between these characteristics. It is also shown that in these problems the BDD algorithm usually spent a higher percentage of the solving time on quantification than in other problems. We therefore conclude that there are models for which the hybrid algorithm is inherently better, and achieves the goal of reducing the effort required for quantification.

In some of the problems where the hybrid algorithm uses more memory than the BDD algorithm, we observe that the time which was spent by the hybrid algorithm on quantification is relatively low. This means that the Boolean operations on the BDDs, other than the pre-image, required a larger part of the computation time. We believe that this is the result of the BDD order that we impose on the All-SAT solver during the quantification, which may not be optimal. Further research should be conducted on adapting our method to other data structures, possibly not canonical, in order to avoid this problem.

			BDD				Hybrid										
							No Op			Op 1			Op1 + Op2				
Model	# FF	Result	# It	Time (s)	Mem	Quant	# It	Time (s)	Quant	# It	Time (s)	Quant	# It	Time (s)	Mem	Quant
S1269	37	fail	8	527	31	60	8	500	50	8	436	43	8	423	29	41
		fail	2	143	15	13	2	183	40	2	187	41	2	193	18	43
		fail	3	10	5	20	3	0	10	3	0	5	3	0	3	5
		fail	8	T.O.	60	99	9	6544	56	9	5791	50	9	5702	49	50
		pass	9	8064	64	99	9	6264	60	9	5580	55	9	5373	52	53
S1512	57	-	10	M.O.	> 1GB	46	10	T.O.	77	10	T.O.	71	10	T.O.	617	75
		fail	30	253	43	43	30	300	52	30	290	50	30	286	60	50
		pass	38	24	11	52	38	50	51	38	47	48	38	47	20	48
		pass	2	0	< 1MB	10	2	10	10	2	10	10	2	10	< 1MB	10
		-	30	M.O.	> 1GB	99	102	T.O.	80	102	T.O.	73	102	T.O.	243	63
		-	9	T.O.	422	95	3	T.O.	96	3	T.O.	95	3	T.O.	510	95
		-	11	T.O.	418	94	11	T.O.	93	11	T.O.	91	11	T.O.	541	90
S1423	74	fail	9	10106	266	48	9	13812	51	9	13790	51	9	13054	242	48
		fail	6	4300	312	70	6	4532	62	6	4504	62	6	4483	302	62
		fail	4	4150	247	50	4	8113	74	4	7949	73	4	8045	394	74
		-	11	T.O.	517	92	8	T.O.	99	8	T.O.	99	9	T.O.	412	99
		-	10	T.O.	498	93	7	T.O.	99	7	T.O.	99	7	T.O.	387	99
S9234	228	pass	258	8322	261	95	258	7741	87	258	5710	82	258	5337	194	81
		pass	3	1284	12	83	3	527	49	3	495	46	3	487	6	45
		fail	123	40	4	99	123	10	34	123	10	34	123	10	4	34
		-	7	T.O.	691	92	9	T.O.	75	9	T.O.	72	9	T.O.	483	69
		-	5	T.O.	721	94	6	T.O.	71	6	T.O.	70	6	T.O.	522	67
S15850	597	pass	4	12284	426	65	4	27331	72	4	24053	68	4	23125	510	67
		pass	3	7630	551	73	3	8121	76	3	7930	75	3	8043	643	77
		pass	3	3961	743	88	3	5403	92	3	5320	92	3	5394	861	90
		pass	2	0	3	17	2	0	24	2	0	0	2	0	5	0
		-	6	T.O.	640	87	6	T.O.	42	6	T.O.	41	6	T.O.	543	41
		-	4	T.O.	746	87	4	T.O.	99	4	T.O.	99	4	T.O.	674	99
S13207	669	-	12	T.O.	432	86	12	T.O.	82	12	T.O.	85	12	T.O.	462	86
		-	8	T.O.	843	76	8	T.O.	69	8	T.O.	65	8	T.O.	684	60
		-	6	T.O.	630	99	5	T.O.	84	5	T.O.	87	5	T.O.	690	86
		-	8	T.O.	455	82	8	T.O.	86	8	T.O.	85	8	T.O.	489	86
		-	5	T.O.	924	91	5	T.O.	90	5	T.O.	89	5	T.O.	751	87
S38584	1452	pass	1	2791	439	99	1	1023	99	1	914	99	1	849	210	99
		-	7	T.O.	792	94	7	T.O.	96	7	T.O.	96	7	T.O.	750	95
		-	8	T.O.	564	95	8	T.O.	96	8	T.O.	95	8	T.O.	498	95

T.O.: Time Out	▓	Hybrid method outperforms the BDD method
M.O: Memory Out	░	BDD method outperforms the hybrid method

Table 3.1: A Comparison of Model Checking Run Times. #FF is the number of state variables, #It is the number of pre-image steps completed, Quant is the percentage of time spent on quantification, and Mem is the size of the memory used by the tool in MB. Timeout is set to 24hr, and Memory limit is 1GB.

			BDD				Hybrid										
							No Op			Op 1			Op1+Op2				
Model	# FF	Result	# It	Time (s)	Mem	Quant	# It	Time (s)	Quant	# It	Time (s)	Quant	# It	Time (s)	Mem	Quant
B12	121	fail	214	16924	54	54	214	18302	65	214	17860	64	214	17808	49	64
		-	21	T.O.	98	64	17	T.O.	64	17	T.O.	65	18	T.O.	102	66
		-	32	T.O.	109	61	30	T.O.	72	30	T.O.	70	30	T.O.	133	71
		-	143	T.O.	45	27	131	T.O.	56	131	T.O.	50	131	T.O.	51	50
		pass	260	10943	150	60	260	12139	65	260	12130	65	260	11941	130	64
B14_1	245	fail	45	790	144	56	45	2166	66	45	2166	83	45	2166	312	83
		fail	32	22611	625	86	28	T.O.	84	28	T.O.	85	28	T.O.	483	82
		-	91	T.O.	124	66	84	T.O.	82	84	T.O.	80	84	T.O.	412	78
		-	6	T.O.	741	91	6	T.O.	88	6	T.O.	81	6	T.O.	720	77
B15_1	449	pass	13	4532	134	73	13	6234	75	13	5794	73	13	5730	120	73
		fail	9	6190	771	86	9	6906	79	9	6882	79	9	6649	674	78
		pass	17	8980	378	83	17	1037	79	17	868	75	17	532	400	59
		-	7	T.O.	683	90	6	T.O.	84	6	T.O.	84	6	T.O.	664	83
		-	19	M.O.	> 1GB	99	19	T.O.	87	19	T.O.	84	19	T.O.	798	82
B21_1	490	fail	45	3104	250	64	45	1756	48	45	1700	46	45	1940	194	53
		-	6	T.O.	774	97	8	T.O.	78	8	T.O.	69	8	T.O.	527	67
		-	6	M.O.	> 1GB	99	6	T.O.	76	6	T.O.	74	6	T.O.	618	70
		-	9	M.O.	> 1GB	99	13	T.O.	82	13	T.O.	77	13	T.O.	437	74
B20_1	490	fail	37	9437	418	72	37	11030	64	37	11002	64	37	12106	491	67
		fail	19	8204	728	98	19	6310	94	19	5629	93	19	5283	534	93
		-	5	M.O.	> 1GB	99	7	T.O.	95	7	T.O.	89	7	T.O.	559	87
		-	12	T.O.	681	96	15	T.O.	95	15	T.O.	90	15	T.O.	573	87
B22_1	735	pass	9	21045	826	97	9	13349	90	9	12174	89	9	10631	613	87
		-	6	T.O.	324	91	7	T.O.	90	7	T.O.	87	7	T.O.	271	85
		-	6	M.O.	> 1GB	98	7	T.O.	96	7	T.O.	95	7	T.O.	685	95
		-	8	T.O.	442	99	8	T.O.	99	8	T.O.	99	8	T.O.	420	99
B17_1	1415	pass	6	39702	620	91	6	43005	84	6	42930	84	6	42882	607	84
		-	2	T.O.	714	90	1	T.O.	99	1	T.O.	99	1	T.O.	681	99
		-	3	T.O.	661	95	3	T.O.	95	3	T.O.	98	3	T.O.	494	98
B18_1	3320	fail	18	5277	97	96	18	3700	94	18	2892	92	18	2431	76	91
		-	1	T.O.	862	94	1	M.O.	99	1	M.O.	99	1	M.O.	> 1GB	99
		-	-	T.O.	637	99	1	T.O.	95	1	T.O.	90	1	T.O.	729	89
B19_1	6642	fail	3	7546	340	87	3	10184	80	3	10041	80	3	11032	560	82
		-	3	M.O.	> 1GB	99	4	T.O.	92	4	T.O.	90	4	T.O.	674	88
		-	4	M.O.	> 1GB	99	4	T.O.	99	4	T.O.	99	4	T.O.	639	99

T.O.: Time Out
M.O: Memory Out

▓ Hybrid method outperforms the BDD method
░ BDD method outperforms the hybrid method

Table 3.2: A Comparison of Model Checking Run Times. #FF is the number of state variables, #It is the number of pre-image steps completed, Quant is the percentage of time spent on quantification, and Mem is the size of the memory used by the tool in MB. Timeout is set to 24hr, and Memory limit is 1GB.

3.4 Related Work

All-SAT engines that are built on top of modern SAT solvers tend to block solutions that were already found by adding their negation to the formula during the search [57, 12, 48, 63]. In [31] a specific order of the search prevents the solver from instantiating the same solution more than once, without adding clauses. In our work, as in [35], the negation of the solutions is kept in a BDD. When performing pre-image computation, the set of next-states is also given to our All-SAT algorithm as a BDD, which decreases the size of the problem substantially relative to clausal representation of the states. The size of the problem is also addressed in the following works: In [49], a ZBDD is used to store solutions found by an All-SAT solver. In [11], a method for managing ZBDDs is suggested. In [26], solutions are stored by using an or-inverter graph. Representing sets by these data structures is not necessarily more compact than using BDDs. However, if desired, we believe that it is straightforward to adapt our new hybrid algorithm, described in Section 3.1.5, to other data structures instead of BDDs.

When using All-SAT for image and pre-image computation in [12, 62, 53, 41, 45, 43], after a solution is found, it is analyzed in order to generalize it to represent a set of solutions. In our algorithm, the branching procedure instantiates maximally justifying assignments, which represent sets of solutions, without actually instantiating assignments to all the variables, and without the overhead of generalizing them.

Hybrid SAT and non-clausal procedures were presented in [3, 25, 46, 42, 78]. In these methods, the non clausal representation of the problem is used to guide the search, learning over the non-clausal representation is performed, and some pruning of empty subspaces is done. However, these branching heuristics are aimed at finding a single solution to the formula, and do not perform best when looking for all of the solutions. In addition, pruning empty subspaces using these procedures implies a significant computational overhead.

Chapter 4

3-Valued Circuit SAT for STE

Symbolic Trajectory Evaluation (STE) is a successful method for formally verifying very large models with wide data paths [73, 70, 86]. We present the theory of STE in Section 2.9. 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 [73]. 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 [85], *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 [7] and [67], the *dual rail* encoding is used to create a CNF formula for STE. This representation uses two Boolean variables. In [66], a 3-valued SAT solver was suggested, which did not perform well. Additionally in [66], an approximation for a 3-valued SAT solver is computed. This approximation, however, does not completely correspond to the semantics of STE.

Particularly interesting for hardware verification is the Circuit-SAT method, 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 Section

2.14. 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.

In the following sections we give 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, which we describe in Section 5.2.

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 3-valued justification algorithm, denoted 3VJA, 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. The DAG representation is a higher level description of the circuit than the CNF representation. It is used for branching, propagating X values, and for termination.

We exploit the fact that for each variable, a Boolean solver holds three possible values, $0, 1$ 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 3VJA on top of zChaff [59], which is a state of the art CNF SAT solver. We employed our tool for solving several STE problems, and compared it to other methods. 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-based STE engine has to compute the values of all the nodes in all the iterations prior to the contradiction.

We also 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.

4.1 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 the *transition relation graph* (TRG) of the circuit, denoted G . The second representation of the circuit is a CNF description G, denoted φ. φ is built as described in [5]. ψ^1_{and} in Figure 4.5 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 φ 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. Throughout this Section we refer to the example in Figure 4.1

4.1.1 *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 constraints 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 φ. Therefore, we do not consider the clauses to be sets of literals, as defined in Section 2.11. Instead, we consider the clauses to be *multi-sets* of literals. The definition of a conflict and constraint propagation remain as in Section 2.12. A variable with a constraint $|_{!0}$ or $|_{!1}$ is marked by the clause $(n \vee n)$ or $(\neg n \vee \neg n)$ 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 the 3-valued setting, these clauses correctly represent the fact that "n is not 0", and "n

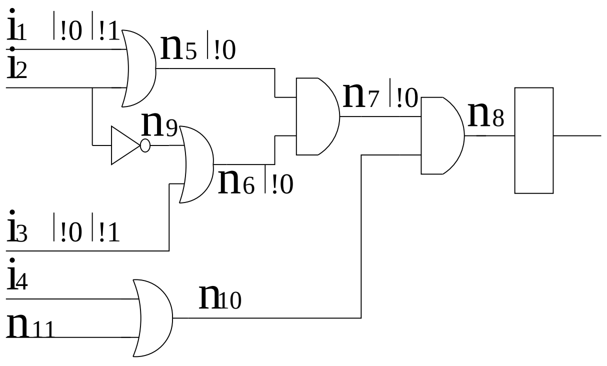


Figure 4.1: Circuit Graph

is not 1", and thus "n is X". Our algorithm does not necessarily satisfy all the clauses in φ. 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 $|_{!1}$ constraints 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 constraints directly on φ. However, when propagating them on G, we also create the appropriate clauses for the implied constraints, and add them to φ.

We demonstrate the propagation of $|_{!0}$ and $|_{!1}$ constraints in our example, where $|_{!0}$ and $|_{!1}$ constraints are given for the inputs i_1 and i_3. i_1 is $|_{!0}$ implies that n_5 is $|_{!0}$. Thus we add the clause (n_5, n_5). Additionally, i_1 is $|_{!1}$ 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 $\neg n_5$ in the clause $(i_2, \neg n_5, \neg n_5)$ prevent that. Similarly, the constraint n_6 is $|_{!0}$, and the clause $(n_9, \neg n_6, \neg n_6)$ are created, as a result of propagating the given constraints i_3 is $|_{!0}$ and i_3 is $|_{!1}$.

In section 2.11 we defined the *resolution tree* for clauses that are created by resolution. In our context, clauses can be also 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 φ. Thus, the generated clauses are considered as the result of applying resolution on the relevant clauses in φ. In the example, the clause (n_5, n_5) can be created by applying resolution on the clauses (i_1, i_1) and $(\neg i_1, n_5)$, resulting in (i_1, n_5), and applying resolution on (i_1, n_5) and $(\neg i_1, n_5)$ again. The definition of the

```
3VJA (G,$\varphi$,n,d)
1)  while true
2)   if (branch() == null)        // All nodes justified
3)     return justification       // Return assignment to leaves
4)   if (bcp on $\varphi$ $\Rightarrow$conflict){
5)     learn conflict clause
6)      if learned $X$ clause {     // $|_{!1}$ or $|_{!0}$ clauses
7)        mark $X$ on $G$
8)        propagate $X$ on $G$
9)        add clauses to $\varphi$
10)    }
11)    if possible
12)      backtrack
13)    else
14)      return unjustifiable
15)  }
```

Figure 4.2: 3VJA. Lines $2, 7$ and 8 are executed on G. Lines $4, 5$ and 9 are executed on φ.

resolution tree thus remains unchanged.

4.1.2 3-Valued Justification Algorithm

Given a *DAG* G of a circuit, a CNF description φ of it, 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 $|_{!1}$ constraints. It is also used for correct termination of the algorithm. The CNF representation φ is used for efficient constraint propagation, detection of conflicts, and for learning. 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. The value X for a node n is not explicitly assigned to it, but rather is represented by n being marked as $|_{!1}$ and $|_{!0}$. $|_{!1}$ and $|_{!0}$ constraints can be learned during the solving process. Next we describe and explain 3VJA. We refer to the pseudo-code given in Figure 4.2.

We begin by describing the branching procedure used in line (2) of 3VJA. This is a 3-valued variation of the justification procedure described in [24]. 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.

After each branch, the assigned value has to be propagated through the variables (line 4). We use φ 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.

We conclude that a justifying assignment for $[r, d]$ is found when we complete the traversal of G (line 3). 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, that is, a conflict occurs but there is no way to backtrack, then we conclude that $[r, d]$ can not be justified.

Next we discuss conflict driven learning in our algorithm. When a conflict occurs, the resolvent of the clauses that were involved in the conflict is added to the problem (line 5). 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 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). Considering the clauses as multi-sets results in clauses that do not change the set of justifying assignments to $[r, d]$ [66].

It is possible in our setting to learn conflict clauses such as (n, n) and $(\neg n, \neg n)$. As described in lines (6-9), 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, and thus maintain the correlation between G and φ.

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 cannot 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

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

Figure 4.3: Learning X clauses. $\uplus$ denotes the *resolution* operation. Refer to the circuit in Figure 4.1, and to the initial constraints that are applied to it. Clauses 1, 3 originate from propagating $|_{!1}$ for i_1 and i_3. 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. Suppose that the branching procedure assigns $n_8 = 1$. This implies $n_7 = 1, n_5 = 1, n_6 = 1, n_9 = 1, i_2 = 0$. In that case, all the literals in clause 1 are 0, and we have a conflict. Clauses $7-10$ are created by applying *resolution* on the original clauses, where clause 10 is the derived conflict clause. Note that clause 10 means that n_7 is $|_{!1}$. This constraint propagates on G, and implies that n_8 and n_{11} are $|_{!1}$. We therefore create clauses 11 and 12.

series of resolutions that is performed upon this occurrence in Figure 4.3. The learned conflict clause is $(\neg n_7, \neg n_7)$, and it is added to φ. 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 φ. The result is that n_7 is X, and n_8 and n_{11} are $|_{!1}$.

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 branching or the current partial assignment. Thus, the information about X nodes, as well as $|_{!0}$ and $|_{!1}$, is available explicitly throughout the rest of the search. Additionally, a mechanism for invalidating X assignments is not required.

4.2 STE with 3-Valued Justification

In this section we show how to employ 3VJA 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 4.1 for solving it.

4.2.1 Constructing Circuits for STE Assertions

Consider a circuit with the TRG G, and an STE assertion $A \Rightarrow C$. A and C are given in TEL, as described in Section 2.9. 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.

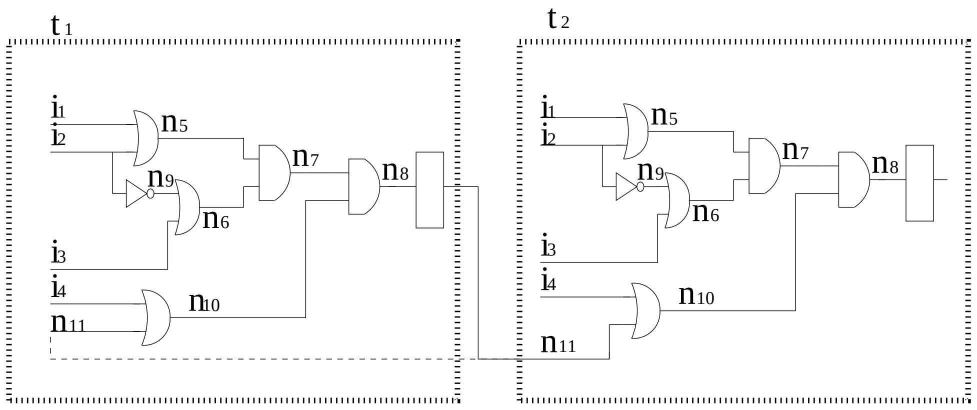


Figure 4.4: An Unrolled Circuit

We create a new graph by unrolling G k times. Each node $n \in G$ 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 at time t is an input to nodes at time t. Due to the new connectivity of the latches, and since G does not have circles, 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 4.4 we show an unrolling of the circuit given in Figure 4.1. t_1 and t_2 are two instances of the circuit. The inputs to the latch n_{11} at 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 $\mathcal{W}$ 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, consider the TEL formula $A = (n,i)\ is\ V_1$, where (n,i) is the result of applying the *next* operator i times. 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 $V_1 \in \mathcal{W}$. 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 straightforward. 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 from $\mathcal{W}$ has only one instance. Constructing a circuit for C is done is the same manner as for A. Note that the constructed circuits for A and C are DAGs. From herein we denote by A and C the corresponding circuits, respectively.

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

Figure 4.5: A CNF representation of an "AND" gate $n = in_1 \wedge in_2$. ψ^1_{and} propagates unit clause implications in both directions. ψ^2_{and} propagates implications only forwards.

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 that represents M, with the restrictions imposed on it by A. This is done by connecting the relevant nodes in M to the inputs of A. We abuse the notation of $\wedge$ and denote this circuit $M \wedge A$. The inputs to $M \wedge A$ are the k instances of the inputs to the hardware model, and the symbolic variables $\mathcal{W}$, used in A. A is in fact an assumption on the circuit. As defined in Section 2.9, 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 nodes that are given a value by A 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 when *bcp*() is applied, they allow forward propagation only. This is demonstrated in Figure 4.5. For a node $n = in_1 \wedge in_2$, we create ψ^2_{and} instead of ψ^1_{and}. For example, if n is assigned the value 1 by A, applying *bcp*() does not propagate this value to in_1 and in_2. On the other hand, forward propagation is still implied.

We construct a circuit that represents the runs M, that satisfy the restrictions of A, do not satisfy the requirements of C. This is done by connecting the relevant nodes in $M \wedge A$ to the inputs of C. We abuse the notation of $\wedge$ and denote the new circuit $\Gamma = M \wedge A \wedge \neg C$. 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 Γ. An example for such a construction is given in Figure 4.6. The node associated with "=" represents a combinational circuit that evaluates 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 is the evaluation of Γ.

4.2.2 Running STE

In order to execute STE, we first have to 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 constraints from

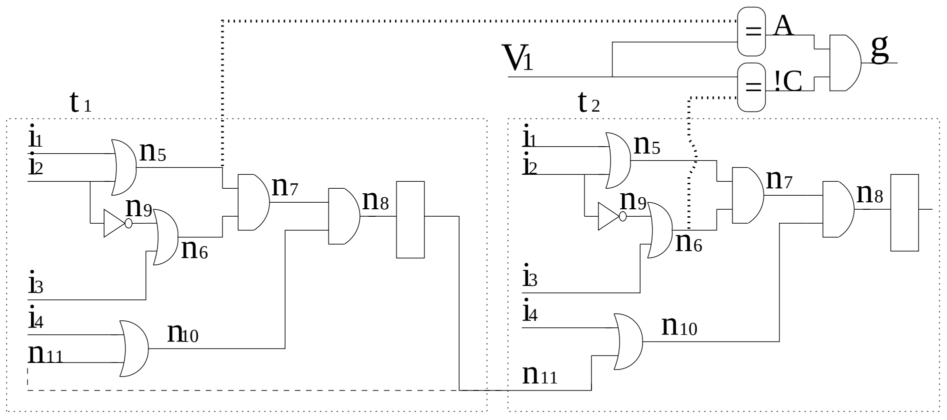


Figure 4.6: An unrolled circuit with an STE assertion

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 run STE by applying 3VJA on $[\gamma, 1]$, where γ is the root of Γ, defined in the previous section. The pseudo code of our algorithm is given in Figure 4.7, and it is explained below.

If a justifying assignment is found by 3VJA (line 3), it represents an assignment to the inputs of Γ that makes γ 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 $A \Rightarrow C$ does not hold in M, and the STE returns this assignment as a counterexample.

If $[\gamma, 1]$ is unjustifiable, then there is no counterexample for the assertion in M. However, we have to determine if the result of STE is 1 or X. Since $[\gamma, 1]$ is unjustifiable, then an empty clause was learned during the unsuccessful justification process. We extract the unSAT core from the resolution tree of the empty clause (line 5), and check if it contains clauses for $|_{!0}$ or $|_{!1}$ nodes, that originate from A (line 6). Such nodes indicate Xs that were implied by the antecedent of the STE assertion. 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 $A \Rightarrow C$ holds in M, and STE returns 1 (line 8). On the other hand, if the unSAT core includes clauses

```
1) 3VJA_STE($\Gamma$)
2)    $a \leftarrow 3VJA([\gamma, 1])$
3)    if $(a! = null)$
4)       return $a$                                      // $a$ is a counterexample
5)    $core \leftarrow\ unSAT - core([\gamma, 1])$      // $[\gamma, 1]$ is unjustifiable
6)    if $(\exists\ |_{!1}$ clause $\in core \vee \exists\ |_{!0}$ clause $\in core)$  // possibly due to $X$s
7)       return $X$
8)    return 1                                           // $[\gamma, 1]$ is unjustifiable
                                                         // regardless of $X$s
```

Figure 4.7: STE using 3VJA.

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 counterexample for $A \Rightarrow C$ due to a too coarse abstraction. Therefore, we have to refine the model in order to prove or falsify the STE assertion, and STE returns X (line 7).

Note that we return X when a proof for unjustifiability of $[\gamma, 1]$, that depends on X values from A is found. However, there might exist another proof for unjustifiability of $[\gamma, 1]$, that *does not* depend on X values. Therefore, it might be possible to prove the STE assertion without refining the model. In our current algorithm, we choose to perform this light-weight justification and refine the model if needed. We discuss automatic refinement in Section 5.2.

4.3 Experimental Results

For evaluating our justification algorithm 3VJA, presented in Section 4.1.2, we implemented it on top of zChaff [59], a state of the art SAT solver, and [83], a circuit-SAT justification framework. We used 3VJA for STE, as described in Section 4.2. For comparison, we used the dual rail encoding for solving SAT based STE [66], and *Forte*, a BDD based STE tool by Intel [73]. Additionally, we used BMC for solving the benchmarks, considering the STE assertions as LTL formulae (without abstraction). 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 4.3. We verified the *associative read* property of the CAM by using “full”, “plain” and “cam” symbolic indexing schemes, as defined in [61]. 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 because the sizes of the BDDs depends on the number of the symbolic variables, whereas the size of the SAT problem depends on the number of nodes in the unrolled circuit. 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 justification 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 [66]. On the other hand, 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

Verification					Time (s)			
	Assertion	D	# vars	#N x10^3	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 x10^3	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 4.1: 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.

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 $17 - 25$.

4.4 Related Work

SAT based methods for STE were previously suggested in [7], [85] [67], and [66].

In [85], *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, which is done to a limited degree of success. This method is inherently different than 3VJA.

In [7], the *dual rail* encoding is used to create a CNF formula for STE. This construction is referred to in [67] as *simulation based SAT STE.* In [67], a different construction is suggested, and is referred to as *constraint based STE.* The *constraint based* construction is equivalent to the construction presented in [5], 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 [67], it is shown that the *constraint based* construction outperforms the *simulation based* construction. As mentioned above, both constraint based and simulation based constructions use the dual rail encoding. As such, they use 2 variables to represent each circuit node. The SAT problem is exponential in the size of the input, and thus using dual rail encoding incurs a significant overhead on the SAT solver. We experimentally compared 3VJA to the *constraint based* construction in Section 4.3.

In [66], the constraint based construction is solved by a 3-valued SAT solver. In that work, Boolean variables of a SAT solver represent 3 values, considering an "unassigned" variable as X. The definition of satisfiability is changed respectively. In [66], 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 work is similar to the resolution defined in [66], the reasons for their correctness are different. As described in [66], modifying the SAT solver to fit the new definition of satisfaction and resolution did not yield good performance.

Additionally in [66], 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.

Chapter 5

Automatic Refinement for Symbolic Trajectory Evaluation

In this chapter we present automatic refinement methods for STE. In Section 5.1 we describe the abstraction - refinement flow in STE. In sections 5.2 we describe an algorithm for automatic refinement for circuit-SAT based STE. We presented this method in [32]. In Section 5.3 we describe an automatic refinement algorithm, which is based on the notion of *responsibility*. We presented this algorithm in [16].

5.1 Refinement in STE

A major strength of STE is the use of abstraction. For a model M, and an STE assertion $A \Rightarrow C$, the abstraction is determined by the assignment of the value X to nodes in M by A. However, if the abstraction is too coarse, then there is not enough information for proving or falsifying the STE assertion. That is, $[M \models A \Rightarrow C] = X$.

The common abstraction and refinement process in STE consists of the following steps: the user writes an STE assertion $A \Rightarrow C$ for M. A explicitly assigns Boolean values or free Boolean variables to nodes in M in different times. All the inputs to M that are not explicitly assigned by A are assigned X. The user then receives a result from STE. If $[M \models A \Rightarrow C] = \bot$ (an antecedent failure), then there is a contradiction between A and M, and the user has to write a new assertion. If $[M \models A \Rightarrow C] = 0$, or $[M \models A \Rightarrow C] = 1$, the process ends with the corresponding result. If $[M \models A \Rightarrow C] = X$, a refinement is required. In this case, there is some *X-possible* node (n, t) which is undecided. The user has to decide how to refine the specification such that the X truth value will be eliminated. That is, the user adds assignments of Boolean values or free Boolean variables to nodes in M that were not assigned by A, thus replacing the X value in these nodes.

The main challenge in this setting is to choose an appropriate subset of these inputs, that will help to eliminate the "*unknown*" STE result. Selecting a "right" set of inputs for refinement is crucial for the success of STE: refining too many inputs may result in memory and time explosion. On the other hand, selecting too few inputs or selecting inputs that do not affect the result of the verification will lead to many iterations with an "*unknown*" STE result.

The common approach to this problem is to manually choose the inputs for refinement. This, however, is labor-intensive and error-prone. Thus, an automatic refinement is desired.

An automatic refinement can be obtained by creating a new antecedent for the STE assertion. The refinement of A should preserve the semantics of $A \Rightarrow C$. Formally, let $A_{new} \Rightarrow C$ denote the refined assertion, and let $runs(M)$ denote the set of all concrete trajectories of M. We require that if $A_{new} \Rightarrow C$ holds in M, then so does $A \Rightarrow C$. Also, if $A_{new} \Rightarrow C$ yields a counterexample ce, then ce is also a counterexample with respect to $A \Rightarrow C$.

In [79], refinement steps add constraints to A by forcing the values of some input nodes at certain times to the value of *fresh symbolic variables* that are not already in $\mathcal{V}$. By initializing an input (in, t) with a fresh symbolic variable instead of X, the value of (in, t) is accurately represented, and knowledge about its effect on M is added. It is important to note that the new variable does not constrain input behavior that was allowed by A, nor does it allow input behavior that was forbidden by A. Thus, the semantics of A is preserved. In [79] it is proven that this refinement method satisfies the requirement discussed above. That is, if $A_{new} \Rightarrow C$ holds in M, then so does $A \Rightarrow C$. Also, if $A_{new} \Rightarrow C$ yields a counterexample ce, then ce is also a counterexample with respect to $A \Rightarrow C$.

In Section 5.2 and Section 5.3 we present automatic refinement methods for STE, which are also based on substituting X values at the inputs to the circuit with fresh symbolic variables.

5.1.1 Related Work

In [82], an automatic abstraction-refinement for symbolic simulation is suggested. However, the first automatic refinement for STE has been suggested in [79]. This work is the closest to ours, and we compare our results to this work in Section 5.3.4. In [15], an automatic refinement for GSTE is suggested. This method, like [79], traverses the circuit nodes after running STE, and performs a model and an assertion refinement. This method is also essentially different from ours, as it is aimed at solving GSTE problems, where an assertion graph describes the specification, and is used in the refinement process.

SAT based refinement was suggested in [68] , where it is used for assisting manual refinement. In [1], a method for automatic *abstraction* without refinement is suggested.

5.2 Automatic Refinement for Circuit-SAT-Based STE

In this section we present a "CEGAR" approach for refining too coarse STE assertions. This approach is applicable for the STE implementation described in Chapter 4, which is based on 3-Valued Circuit SAT.

The STE method presented in Section 4 runs 3VJA for justifying $[\gamma, 1]$, where γ is the root of $\Gamma = M \wedge A \wedge \neg C$. A justifying assignment for $[\gamma, 1]$ represents a counterexample for the assertion $A \Rightarrow C$. If $[\gamma, 1]$ is unjustifiable, than either the assertion holds, or the abstraction implied by A is too coarse.

The description of A that is given to 3VJA includes $|_{!0}$ and $|_{!1}$ clauses which correspond to the nodes that are assigned X by A. For an unjustifiable instance, 3VJA returns a resolution tree, which is a proof that the instance is unjustifiable. Note that such a proof is not unique, and depends on the the branching heuristic of 3VJA, and on its implementation. We define a *spurious* proof to be a resolution tree returned by 3VJA, where the unSAT core defined by it includes $|_{!1}$ or $|_{!0}$ clauses that originate from A. That is, a spurious proof is a proof that depends on the X assignments in A. We denote such unSAT cores *X-unSAT*. For example, the unSAT core $(n_1, n_1), (\neg n_1, n_2)$, and $(\neg n_2, \neg n_2)$ includes clauses that describe assignment of X to n_1 and n_2. Therefore, it is an *X-unSAT*, and its corresponding proof is spurious.

We refine A if 3VJA returns a spurious proof of unjustifiability of $[\gamma, 1]$. This is done by introducing fresh symbolic variables for the nodes for which there exist $|_{!1}$ or $|_{!0}$ clauses in *X-unSAT*. Thus, we invalidate the current spurious proof.

By refining only X nodes that are in the unSAT core, we refine only the variables that are needed for eliminating the current spurious proof. This means that we add fresh symbolic variables only for X nodes that took part in implying X on γ, rather than all the X nodes in the cone of influence of γ. Refer to the example in Figure 5.1. Consider an antecedent A that does not assign a Boolean value or a symbolic variable to i_1, i_3 and i_4, which implies that they are all assigned X. Assume, also, that A assigns 1 to n_{11}. This implies $n_{10} = 1$, and n_8 is $|_{!0}$. When trying to justify $n_8 = 1$, as described in Section 4.1.2, 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

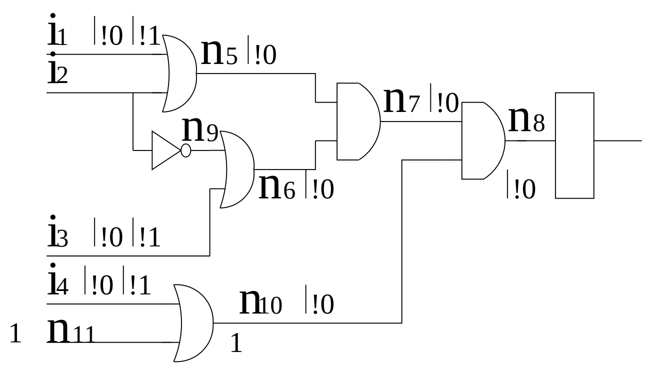


Figure 5.1: A circuit with the implications of the antecedent $A = n_{11}$ is 1.

```
3-Valued_Circuit_SAT_STE (M,A,C)
1) while (true) {
2)    $\Gamma \leftarrow M \wedge A \wedge \neg C$
3)    $(solution, resolution\_tree) \leftarrow 3VJA(\gamma, 1)$
4)    if (resolution_tree = NULL)
5)       return $(0, solution)$                    // solution is a counterexample
6)    if (no $|_{!1}$ or $|_{!0}$ clauses from $A$ in resolution_tree)
7)       return $(1, resolution\_tree)$             // resolution_tree is a proof
8)    $A \leftarrow refine\ antecedent\ by\ X\text{-}unSAT$
9) }
```

Figure 5.2: CEGAR framework for 3-Valued circuit-SAT based STE with automatic refinement. 3VJA returns a justifying assignment if one exists, or a resolution tree if the problem is unjustifiable. If the assertion holds in the model, the algorithm returns a resolution tree which is a proof of correctness. If the assertion fails, the algorithm returns a counterexample.

not add a variables for i_2 and i_4.

Our refinement eliminates the proof of unjustifiability that was returned by 3VJA. Running 3VJA again, we either find another spurious proof that has to be refined, a concrete proof of unjustifiability, or proof of justifiability (a justifying assignment). The pseudo code for the complete automatic framework is given in Figure 5.2.

Note that in line 8 of the algorithm we refine A when 3VJA returns a spurious proof for unjustifiability of $[\gamma, 1]$. However, there might exist another *non*-spurious proof for unjustifiability of $[\gamma, 1]$. Therefore, it might be possible to prove the STE assertion without refining the model. We could avoid this by changing the justification heuristic of 3VJA, and by traversing larger portions of the circuit. Thus, we have a trade-off between light-weight justification with more refinements, and heavy-weight justification with fewer refinements. In our current algorithm, we choose to perform the light-weight justification and refine the model more often.

5.3 Automatic Refinement for STE Using Responsibility

In this section we present a CEGAR approach to automatic refinement for STE, which is based on the notion of *responsibility*. When STE returns an "*unknown*" result, for each input with X value, we compute its *Degree of Responsibility* (DoR) to the "unknown" STE result. We then refine those inputs whose DoR is maximal. The algorithm we present in completely independent of the implementation of the STE engine or its data structures.

5.3.1 Causality and responsibility

In this section, we review the definitions of causality and responsibility. We start with causality. The most intuitive definition of causality is *counterfactual causality*, going back to Hume [40],which is formally defined as follows.

We say that an event J is a *counterfactual cause* of event K if the following conditions hold: (a) both J and K are true, and (b) if we assign J the value false, then K becomes false. That is, had J not happened, then K would not have happened. We refer to the dependence of K on J as a *counterfactual dependence*.

In this work, we use a simplified version of the definition of causality from [38]. In order to define causality formally, we start with the definition of causal models (again, due to [38]).

A *causal model* M is a tuple $\langle \mathcal{U}, \mathcal{D}, \mathcal{R}, \mathcal{F} \rangle$, where $\mathcal{U}$ is the set of *exogenous* variables (that is, variables whose value is determined by constraints outside of the model), $\mathcal{D}$ is the set of *endogenous* variables (that is, variables whose value is determined by the model and the current assignment to $\mathcal{D}$), $\mathcal{R}$ associates with each variable in $\mathcal{U} \cup \mathcal{D}$ a nonempty range of values, and the function $\mathcal{F}$ associates with every variable $Y \in \mathcal{D}$ a function F_Y that describes how the value of Y is determined by the values of all other variables in $\mathcal{U} \cup \mathcal{D}$.

A *context* $\vec{d}$ is an assignment for variables in $\mathcal{D}$ (the values of variables in $\mathcal{U}$ are considered constant).

In this work we restrict our attention to models in which variables do not depend on each other.

A *causal formula* φ is a formula over the set of variables $\mathcal{U} \cup \mathcal{D}$. A causal formula φ is true or false in a causal model given a context $\vec{d}$. We write $(M, \vec{d}) \models \varphi$ if φ is true in M given a context $\vec{d}$. We write $(M, \vec{d}) \models [\vec{Z} \leftarrow \vec{z}]\varphi$ if φ holds in the model M given the context $\vec{d}$ and the assignment $\vec{z}$ to the variables in the set $\vec{Z} \subset \mathcal{D}$, such that $\vec{z}$ overrides $\vec{d}$ for variables in $\vec{Z}$.

With these definitions in hand, we can give the simplified definition of cause based on the definition in [38]. The main simplification is due to the fact that in our models, variables do not depend on each other, and thus there is no need to explicitly check various cases of mutual dependence between variables.

We say that $Y = y$ is a *cause* of φ in $(M, \vec{d})$ if the following conditions hold:

AC1. $(M, \vec{d}) \models (Y = y) \wedge \varphi$.

AC2. There exists a partition $(\vec{Z}, \vec{W})$ of $\mathcal{D}$ with $Y \in \vec{Z}$ and some setting $(y', \vec{w}')$ of the variables in $Y \cup \vec{W}$ such that:

(a) $(M, \vec{d}) \models [Y \leftarrow y', \vec{W} \leftarrow \vec{w}']\neg\varphi$. That is, changing $(Y, \vec{W})$ from their original assignment $(y, \vec{w})$ (where $\vec{w} \subset \vec{d}$) to $(y', \vec{w}')$ changes φ from **true** to **false**.

(b) $(M, \vec{d}) \models [Y \leftarrow y, \vec{W} \leftarrow \vec{w}']\varphi$. That is, setting $\vec{W}$ to $\vec{w}'$ should have no effect on φ as long as Y has the value y.

Essentially, this definition of a cause says that $Y = y$ is a cause of φ if both $Y = y$ and φ hold in the current context $\vec{d}$, and there exists a change in $\vec{d}$ that creates a counterfactual dependence between $Y = y$ and φ.

The definition of responsibility introduced in [17] refines the "all-or-nothing" concept of causality by measuring the degree of responsibility of $Y = y$ for the truth value of φ in $(M, \vec{d})$. The following definition is due to [17]:

Let k be the size of the smallest $\vec{W} \subset \mathcal{D}$ such that $\vec{W}$ satisfies the condition **AC2**. Then, the *degree of responsibility (DoR)* of $Y = y$ for the value of φ in $(M, \vec{d})$, denoted $dr((M, \vec{d}), Y = y, \varphi)$, is $1/(k + 1)$.

Thus, the degree of responsibility measures the minimal number of changes that have to be made in $\vec{d}$ in order to make $Y = y$ a counterfactual cause of φ. If $Y = y$ is not a cause of φ in $(M, \vec{d})$, then the minimal set $\vec{W}$ is taken to have cardinality ∞, and thus the degree of responsibility of $Y = y$ is 0. If φ counterfactually depends on $Y = y$, its degree of responsibility is 1. In other cases the degree of responsibility is strictly between 0 and 1. Note that $Y = y$ is a cause of φ iff the degree of responsibility of $Y = y$ for the value of φ is greater than 0.

As we argue in Section 5.3.2, in our setting it is reasonable to attribute weights to the variables in order to capture the cost of changing their value. Thus, we use the *weighted* version of the definition of the degree of responsibility, also introduced in [17]:

Let $wt(Y)$ be the weight of Y and $wt(\vec{W})$ the sum of the weights of variables in the set $\vec{W}$. Then, the *weighted degree of responsibility* of $Y = y$ for φ is $wt(Y)/(k + wt(Y))$, where k is the minimal $wt(\vec{W})$ of a $\vec{W} \subset \mathcal{D}$ for which AC2 holds. This definition agrees with the non-weighted definition of degree of responsibility if the weights of all variables are 1.

We note that in general, there is no connection between the degree of responsibility of $Y = y$ for the value of φ and a probability that φ counterfactually depends on $Y = y$. Basically, responsibility is concerned with the minimal number of changes in a given context that creates a counterfactual dependence, whereas probability is measured over the space of all possible assignments to variables in $\mathcal{D}$.

5.3.2 Responsibility in STE Graphs

In section 5.3.3 we show how to refine STE assertions by using the degree of responsibility (dr) of inputs for *X-possible* nodes. Consider a circuit M, and an STE assertion $A \Rightarrow C$, such that $[M \models A \Rightarrow C] = X$ and let r be an undecided node. In this section we show how M can be viewed as a causal model, and present an algorithm for computing the degree of responsibility of an input to M for "r is *X-possible*".

STE circuits as causal models

In order to verify the assertion $A \Rightarrow C$, M has to be simulated k times, where k is the *maximal depth* of A and C. We create an unrolling of the TRG of M, as described in Section 4.2.1. We assume that the only nodes assigned by A are leaves. It is straightforward to extend the discussion to internal nodes that are assigned by A, and to nodes that get their value from propagating the assignments of A.

Regarding M as a *causal model* requires the following definitions:

1. A set of variables and their partition into $\mathcal{U}$ and $\mathcal{D}$, the exogenous and endogenous variables, respectively.

2. $\mathcal{R}$, the range of the endogenous variables.

3. Values for the exogenous variables $\mathcal{U}$.

4. A context $\vec{d}$, which is an assignment to the variables in $\mathcal{D}$.

5. $\mathcal{F}$, a function which associates each variable $Y \in D$ with a function F_Y that describes its dependence in all the other variables.

Following are the required definitions:

1. The inputs of M are defined to be the variables of the causal model. The inputs that are assigned 1 or 0 by the antecedent A are considered the exogenous variables $\mathcal{U}$. The values of these variables cannot change, and are viewed as part of the model M. The rest of the inputs to M are the endogenous variables $\mathcal{D}$.

2. The range of the variables in $\mathcal{D}$ is $\{0, 1, X\} \cup \mathcal{V}$, where $\mathcal{V}$ is the set of symbolic variables used by A.[1]
3. The values of the exogenous variables are the Boolean values given to them by A.
4. The context $\vec{d}$ is the current assignment to $\mathcal{D}$, imposed by the antecedent A. This assignment is either an association with a symbolic Boolean variable, or with the value X.
5. The function $\mathcal{F}$ associates each variable with the identity function. We use this definition since the variables are inputs to a circuit, and therefore their values do not depend on each other.

Next we have to define a causal formula φ. For an undecided node r, we want to compute the responsibility of the leaves having X values for "r is *X-possible*". We define the *causal formula* φ to be "r is *X-possible*". Since the context $\vec{d}$ is imposed by the antecedent A, and Since "r is *X-possible*" holds under A, we have $(M, \vec{d}) \models \varphi$.

We will compute a weighted degree of responsibility, as described in Section 5.3.1. We choose $wt(n) = 1$ if $\vec{d}(n) \in \mathcal{V}$, and $wt(n) = 2$ if $\vec{d}(n) = X$. Next we explain this choice of weights. For computing the degree of responsibility, we consider changes in the context $\vec{d}$ that replace the assignments to some of the variables in $\mathcal{D}$ from X or $v_i \in \mathcal{V}$ to a Boolean value. When running STE, a symbolic variable may assume either of the Boolean values. On the other hand, a leaf that is assigned X cannot take a Boolean value without changing the antecedent of the STE assertion. Therefore, we consider changing $\vec{d}$ for a variable n such that $\vec{d}(n) \in \mathcal{V}$ to be easier than for a variable n such that $\vec{d}(n) = X$. Thus, our choice of weights takes into account the way in which STE regards X and $v_i \in \mathcal{V}$.

We have shown how an unrolled model M can be viewed as a causal model. Let $I_X(r)$ and $I_V(r)$ be the sets of leaves in $BCOI(r)$, for which A assigns X and symbolic variables, respectively. From herein, for $l \in I_X(r)$, we denote by $dr(M, l, r)$ the degree of responsibility of "l is X" for "r is *X-possible*". Next we present an algorithm that computes an approximate degree of responsibility of each leaf in $I_X(r)$ for "r is *X-possible*".

Computing Degree of Responsibility in Trees

Computing responsibility in circuits is known to be $\mathrm{FP}^{\Sigma_2^P[\log n]}$-complete [2] in general [17], and thus intractable. In order to achieve an efficiently

[1]For simplicity of presentation, we do not distinguish between a symbolic variable $v_i \in \mathcal{V}$ and its corresponding element in $\mathcal{R}$.

[2]$\mathrm{FP}^{\Sigma_2^P[\log n]}$ is the class of functions computable in polynomial time with $\log n$ queries to oracle in Σ_2.

computable approximation, our algorithm is inspired by the algorithm for read-once formulae in [18]. It involves one traversal of the circuit for each $l \in I_X(r)$, and its overall complexity is only quadratic in the size of M. We start by describing an exact algorithm for M which is a tree, and then introduce the changes for M which is a DAG. The latter approach returns approximated results.

We define the following values that are used by our algorithm.

- $c_0(n, M)$: the minimal sum of weights of leaves in $I_X(r) \cup I_V(r)$ that we have to assign 0 or 1 in order to make n evaluate to 0.
- $c_1(n, M)$: the minimal sum of weights of leaves in $I_X(r) \cup I_V(r)$ that we have to assign 0 or 1 in order to make n evaluate to 1.
- $s(n, M, l)$: the minimal sum of weights of leaves in $I_X(r) \cup I_V(r)$ that we have to assign 0 or 1 in order to make "n is *X-possible*" counterfactually depend on "l is X". If there is no such number, that is, there is no change in the context that causes this dependability, we define $s(n, M, l) = \infty$.

If clear from the context, we omit M from the notation of c_0, c_1 and s.

We would like to compute the degree of responsibility of every leaf $l \in I_X(r)$ for "r is *X-possible*". Therefore, for each $l \in I_X(r)$, our algorithm computes $s(r, l)$. We denote by $A(n)$ the assignment to node n in M, imposed by A. We discuss a model M with AND and NOT operators. Extending the discussion to OR, $NAND$ and NOR operators is straightforward. Given r and $l \in I_X(r)$, our algorithm computes $s(r, l)$ by starting at r, and executing the recursive computation described next. Note that only values that are actually needed for determining $s(r, l)$ are computed.
For a node n, $s(n, l)$ is recursively computed by:

- For n a leaf: if $(n = l)$ then $s(n, l) = 0$, because the value of l counterfactually depends on itself. Otherwise, $s(n, l) = \infty$ since a leaf does not depend on other leaves.
- For $n = n_1 \wedge \ldots \wedge n_m$: W.l.o.g. we assume that l belongs to the subtree of M rooted in n_1 (since M is a tree, l belongs to a subtree of only one input of n). In order to make the value of n counterfactually depend on the value of l, all input to n, except for n_1, should be 1, and the value of n_1 should counterfactually depend on the value of l. Thus, $s(n, l) = s(n_1, l) + \sum_{i=2}^{m} c_1(n_i)$.
- For $n = \neg n_1$: $s(n, l) = s(n_1, l)$, since "n is *X-possible*" iff "n_1 is *X-possible*".

For a node n, $c_0(n)$ and $c_1(n)$ are recursively computed by:

- For n a leaf:
  - If $A(n) = 0$, $c_0(n) = 0$, because no change in the assignments to $I_X(r) \cup I_V(r)$ is required. $c_1(n) = \infty$, because no change in the assignments to $I_X(r) \cup I_V(r)$ will change the value of n.
  - Similarly, if $A(n) = 1$, $c_1(n) = 0$ and $c_0(n) = \infty$.
  - If $A(n) = X$, $c_0(n) = c_1(n) = 2$, because only the value of n has to be changed, and the weight of n is 2.
  - If n is associated with a symbolic variable, $c_0(n) = c_1(n) = 1$, because only the value of n has to be changed, and the weight of n is 1.
- For $n = n_1, \wedge \ldots \wedge n_m$:
  - It is enough to change the value of one of its inputs to 0 in order to change the value of n to 0, thus $c_0(n) = \min_{i \in \{1,\ldots,n\}} c_0(n_i)$.
  - The values of all the inputs of n should be 1 in order for n to be 1, thus $c_1(n) = \sum_{i=1}^{n} c_1(n_i)$.
- For $n = \neg n_1$
  - $c_1(n) = c_0(n_1)$ and $c_0(n) = c_1(n_1)$, as any assignment that gives n_1 the value 0 or 1, gives n the value 1 or 0, respectively.

The computation above directly follows the definitions of c_0, c_1 and s, and thus its proof of correctness is straightforward. For a node r and leaf l, computing the values $c_0(n), c_1(n)$ and $s(n, l)$ for all $n \in BCOI(r)$ is linear in the size of M. Therefore, computing $s(r, l)$ for all $l \in I_X(r)$ is at most quadratic in the size of M. Note that when computing $s(r, l)$ for all leaves l, for a node n, each of $c_0(n)$ and $c_1(n)$ is computed once at most. This is because the values of $c_0(n)$ and $c_1(n)$ do not depend on the relationship between l and r.

We demonstrate the computations done by our algorithm on the circuit in Figure 5.3. The antecedent associates l_2, l_4 with symbolic variables, and l_1, l_3 with X. For node *out*, "*out* is *X-possible*" holds. We want to compute $s(out, l_3)$. l_3 is in the subtree of n_2. Therefore, $s(out, l_3) = c_1(n_1) + s(n_2, l_3)$. Since n_1 is an AND gate, $c_1(n_1) = c_1(l_1) + c_1(l_2)$. n_2 is also an AND gate, and therefore $s(n_2, l_3) = c_1(l_4) + s(l_3, l_3)$. The weight of the leaves is according to their assignment. Therefore, $c_1(l_2) = c_1(l_4) = 1$ and $c_1(l_1) = 2$. Additionally, $s(l_3, l_3) = 0$.

Finally, the degree of responsibility of "l is X" for "r is *X-possible*" is defined according to the weighted responsibility, $dr(M, l, r) = \frac{2}{s(r,l)+2}$. If $s(r, l) = \infty$, then $dr(M, l, r) = 0$.

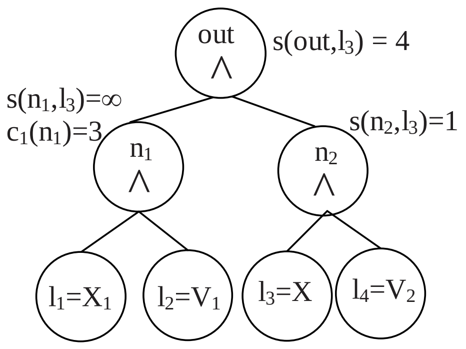


Figure 5.3: Computing Responsibility

Computing an Approximate Degree of Responsibility in DAGs

We now introduce a change to the definition of $s(n, l)$, resulting in an efficiently computable approximation of the degree of responsibility in DAGs, as required for STE.

For a DAG M, and a node $n = n_1 \wedge \ldots \wedge n_m$, we no longer assume that l belongs to a subtree of only one input of n. Let $N^S = \{n_i | s(n_i, l) \neq \infty, i \in \{1, \ldots, m\}\}$, and let $N^\infty = \{n_i | s(n_i, l) = \infty, i \in \{1, \ldots, m\}\}$. That is, N^S is the set of inputs to n whose subtrees contain l, and N^∞ is the set of inputs to n whose subtrees do not contain l.

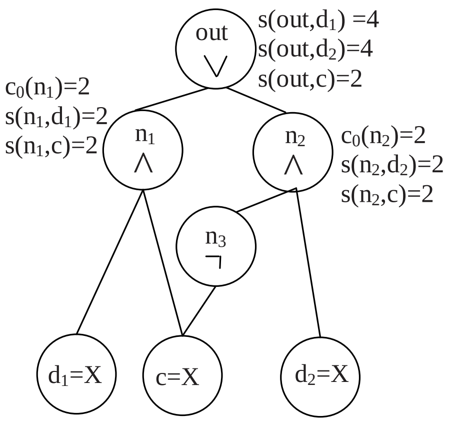
We define $s(n, l)$ to be:

$$s(n,l) = \sum_{n_i \in N^S} \frac{s(n_i, l)}{|N^S|} + \sum_{n_i \in N^\infty} c_1(n_i)$$

The definitions of $dr(M, l, n)$ remains the same, and therefore $dr(M, l, n)$ is inversely proportional to $s(n, l)$. Our new definition is aimed at giving higher degree of responsibility to leaves that belong to subtrees of multiple inputs to n. Such leaves are likely to be control signals, or otherwise more effective candidates for refinement than other variables. Our empirical experience supports this choice.

We demonstrate the effect of this definition on the multiplexer in Figure 5.4. d_1 and d_2 are the data inputs to the multiplexer, and c is its control input. If $c = 1$, then $out = d_1$, else, $out = d_2$. The value $s(out, d_1)$ is given by $s(out, d_1) = c_0(n_2) + s(n_1, d_1) = 4$. The same computation applies to d_2. On the other hand, c belongs to the subtrees of both n_1 and n_2. Therefore, $s(out, c) = \frac{s(n_1,c)+s(n_2,c)}{2} = 2$. Consequently, $dr(c, out) = \frac{1}{2}$, whereas $dr(d_1, out) = dr(d_2, out) = \frac{1}{3}$. That is, the dr of the control signal is higher than that of the data signals.

The rest of the algorithm remains as in Section 5.3.2. Note that since M is a DAG, rather than a tree, not changing the computation of c_0 and



$$out = (c \wedge d_1) \vee (\neg c \wedge d_2)$$

Figure 5.4: A multiplexer.

c_1 makes it an approximation, as it does not take into account possible dependencies between inputs of nodes.

5.3.3 Applying Responsibility to Automatic Refinement

Refinement of an STE assertion is required when the return value of an STE run is X. In that case, the set of undecided nodes is returned by STE. The goal of the refinement is to add information such that undecided nodes become decided. In this section we show how we employ the concept of responsibility for efficiently refining STE assertions.

The outline of the refinement algorithm follows the discussion in section 5.1: First, a refinement goal r is selected from within the set of undecided nodes. Then, a set of input nodes in $I_X(r)$ is chosen, to be initialized to new symbolic variables.

Choosing a Refinement goal

Our refinement algorithm chooses a single refinement goal in each refinement iteration. This way, the verification process might be stopped early if the STE requirement of a single node does not hold, without handling the other undecided nodes. Additionally, conceptual relations between the undecided nodes may make them depend on a similar set of inputs. Thus, refinement targeted at one node may be useful for the other nodes as well. For example, all bits of a data vector are typically affected by the same set of control signals.

We would like to add as little symbolic variables as possible. Thus, from within the set of undecided nodes, we choose the node with the minimal

```
$RespSTE(M, A, C)$
1) while ($[M \models A \Rightarrow C] = X$) {
2)   $r \leftarrow$ choose refinement target
3)   for all $l \in I_X(r)$
4)     compute $dr(l, r)$
5)   $max \leftarrow max\{dr(l, r))|l \in I_X(r)\}$
6)   $I_{ref} \leftarrow \{l|l \in I_X(r), dr(l, r) = max\}$
7)   for all $l_i \in I_{ref}$
8)     add a symbolic variable $v_{l_i}$ to $A$
9) }
```

Figure 5.5: RespSTE. I_{ref} is the set of inputs with the lowest degree of responsibility for "r is *X-possible*".

number of inputs in its BCOI, and among these we choose the one with the minimal number of nodes in its BCOI. This approach has also been taken in [79].

Choosing Input Nodes

Given a refinement goal r, we have to choose a subset of nodes $I_{ref} \subseteq I_X(r)$ that will be initialized to new symbolic variables, trying to prevent the occurrence of "r is *X-possible*". We choose the nodes in $I_X(r)$ with the highest degree of responsibility for "r is *X-possible*", as computed by the algorithm in Section 5.3.2. These nodes have the greatest effect on "r is *X-possible*", and are likely to be the most effective nodes for refinement. Our experimental results support this choice of nodes, as shown in Section 5.3.4.

Given the refinement algorithm described above, we construct *RespSTE*, an iterative algorithm for verifying STE assertions: for a model M and an STE assertion $A \Rightarrow C$, while STE returns $[M \models A \Rightarrow C] = X$, RespSTE iteratively chooses a refinement root $r \in M$, computes the degree of responsibility of each leaf $l \in I_X(r)$ for "r is *X-possible*" and introduces new symbolic variables to A, for all leaves with the highest degree of responsibility. A pseudo code of RespSTE is given in Figure 5.5.

5.3.4 Experimental Results

For evaluating our algorithm *RespSTE*, we implemented and used it in conjunction with *Forte*, a BDD based STE tool by Intel [73].

For our experiments we used the Content Addressable Memory (*CAM*) module from Intel's GSTE tutorial, and IBM's Calculator 2 design [81].

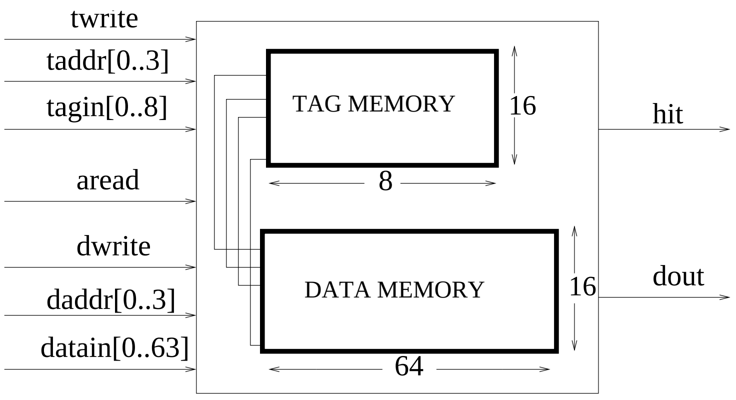


Figure 5.6: Content Addressable Memory module

These models and their specifications are interesting and challenging for model checking. We briefly described these models in Section 4.3. For analyzing our experiments with automatic refinement, it is required to further understand the structure and operation of these models, and therefore we give a more through description of them. All experiments used dedicated computers with 3.2Ghz Intel Pentium CPU, and 3GB RAM, running Linux operating system.

Verifying CAM Module

A CAM is a memory module that for each data entry holds a tag entry. Upon receiving an associative read (aread) command, the CAM samples the input "tagin". If a matching tag is found in the CAM, it gives the "hit" output signal the value 1, and outputs the corresponding data entry to "dout". Otherwise, "hit" is given the value 0. The verification of the aread operation using STE is described in [61]. The CAM that we used is shown in Figure 5.6. It contains 16 entries. Each entry has a data size of 64 bits and a tag size of 8 bits. It contains 1152 latches, 83 inputs and 5064 combinational gates.

We checked the CAM against three assertions. The refinement steps of these assertions are presented in Table 5.1. Each row in the table describes a single refinement iteration, the name of the goal node, and the name and time of the inputs for which symbolic variables were added.

Given $\overrightarrow{\mathrm{TAG}}$ and $\overrightarrow{A}$, vectors of symbolic variables, Assertion 1 is: $(\text{tagin is } \overrightarrow{\mathrm{TAG}}) \wedge (\text{taddr is } \overrightarrow{A}) \wedge (\text{twrite is } 1) \wedge \mathbf{N}\ ((\text{aread is } 1) \wedge (\text{tagin is } \overrightarrow{\mathrm{TAG}}))$ $\Longrightarrow \mathbf{N}$ (hit is 1). This is to check that if a tag value $\overrightarrow{\mathrm{TAG}}$ is written to an address $\overrightarrow{A}$ in the tag memory at time 0, and at time 1 $\overrightarrow{\mathrm{TAG}}$ is read, then it should be found in the tag memory, and hit should be 1. If at time 1 there is no write operation to the tag memory $((twrite, 1) = 0)$, then $\overrightarrow{\mathrm{TAG}}$

As.	It	Goal	Added Vars
1,2	1	hit,1	twrite,1
1,2	2	hit,1	taddr [0:3], 1
2	3	dout[0],1	dwrite,1
2	4	dout[0],1	daddr [0:3], 1
2	5	dout[0],1	din [0], 1
3	1	dout[0],2	tagmem_0 [0:7], 0

Table 5.1: Refinement steps for CAM module. Each line represents a single refinement iteration. Added Vars (n,t) represents the association of node n at time t with a symbolic variable. The first two iterations of Assertion 2 are the same as those of Assertion 1.

should be found in address $\overrightarrow{A}$. If $(twrite, 1) = 1$, $\overrightarrow{\text{TAG}}$ should still be found, since it is written again to the tag memory. Therefore, Assertion 1 should pass. However, since twrite and taddr at time 1 are X, the CAM cannot determine whether to write the value of $(tagin, 1)$ to the tag memory, and to which tag entry to write it. As a result, the entire tag memory at time 1 is X, causing $(hit, 1)$ to be X. Thus, $[M \models A \Rightarrow C] = X$. In two consecutive refinement iterations, $(twrite, 1)$ and $(tadder, 1)$ are associated with new symbolic variables, and the assertion passes.

Assertion 2 extends Assertion 1 by adding the constraint $(datamem[\overrightarrow{A}], 0)$ is $\overrightarrow{D}$ to the antecedent, and the requirement (dout,1) is $\overrightarrow{D}$ to the consequent. That is, in addition to $(hit, 1) = 1$, the assertion checks if the data at the output is equal to the data in the corresponding entry at time 0. This specification is erroneous. If new data is written at time 1 to the data entry associated with $\overrightarrow{\text{TAG}}$, then dout at time 1 will be equal to the new data. The first two refinements are the same as for Assertion 1. The variables that are added in the consecutive refinement iterations allow STE to generate a counterexample in which dwrite at time 1 equals 1, daddr at time 1 equals taddr at time 0, and din[0] at time 1 is different from D[0]. Thus, the assertion fails. Note that $I_X((dout[0], 1))$ includes dwrite, daddr and din[0], all at times 0 and 1, the initial values of all tag memory entries and of bit number 0 of all data memory entries. However, symbolic variables were added to the least number of inputs required for falsifying the assertion.

Assertion 3 is:
(tagin is $\overrightarrow{TAG}$)$\wedge$ (taddr is $\overrightarrow{A}$)$\wedge$(twrite is 1)$\wedge$($datamem[\overrightarrow{A}]$ is $\overrightarrow{D}$) $\wedge$
$\mathbf{N}$((twrite is 0) $\wedge$ (dwrite is 0)) $\wedge$
$\mathbf{N^2}$((aread is 1) $\wedge$ (tagin is$\overrightarrow{\text{TAG}}$) $\wedge$ (twrite is 0) $\wedge$ (dwrite is 0)) $\Rightarrow$
$\mathbf{N^2}$((hit is 1) $\wedge$ (dout is $\overrightarrow{D}$)).
This assertion checks that if at time 0 $\overrightarrow{TAG}$ is written to address $\overrightarrow{A}$, and datmem[$\overrightarrow{A}$] is $\overrightarrow{D}$, and no other write operation is performed, then reading

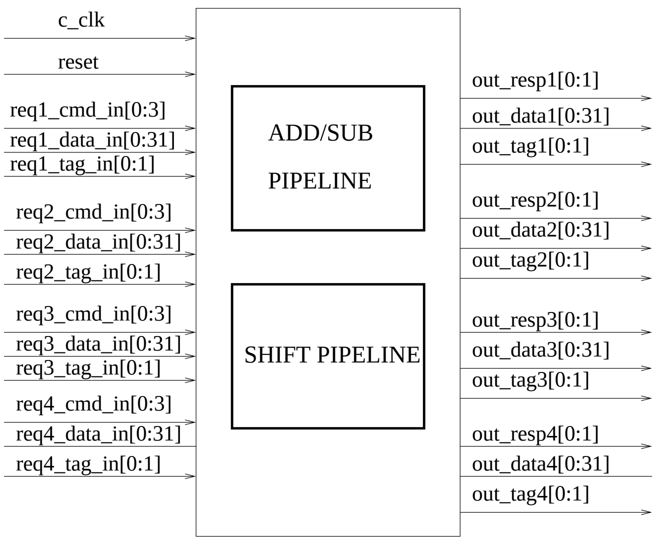


Figure 5.7: Calculator 2 modlue

$\overrightarrow{TAG}$ at time 2 results in $hit = 1$ and $dout = \overrightarrow{D}$. Assertion 3 fails after one iteration, which adds symbolic variables for tag entry 0 at time 0. This allows STE to find a counterexample in which the initial value of tag entry 0 is equal to $\overrightarrow{TAG}$, and the data entry that is associated with it is returned by the CAM, instead of $\overrightarrow{D}$. Assertion 3 was refined by adding the smallest number of symbolic variables required for falsifying it.

Verifying Calculator 2

The Calculator 2 design [81], shown in Figure 5.7, is used as a case study design in simulation based verification. It contains 2781 latches, 157 inputs and 56960 combinational gates. The calculator has two internal arithmetic pipelines: one for add/sub and one for shifts. It receives commands from 4 different ports, and outputs the results accordingly. The calculator supports 4 types of commands: add, sub, shift right and shift left. The response is 1 for good, 2 for underflow, overflow or invalid command, 3 for an internal error and 0 for no response. When running the calculator, reset has to be 1 for the first 3 cycles.

We checked the calculator against four assertions. For all but one of the assertions, RespSTE added the smallest number of symbolic variables required for proving or falsifying the assertion. The refinement steps of these assertions are presented in Table 5.2. Each row in the table describes a single refinement iteration, the name of the goal node, and the name and time of the inputs for which symbolic variables were added. In all the assertions, reset is 1 for the first 3 cycles, as required by the calculator design. Also, a vector $\overrightarrow{P}$ of symbolic variables is used to choose the port sending the

As.	It	Goal	Added Vars
1	1	out_resp1 [0],7	req1_data_in [0],4
1	2	out_resp1 [0],7	req1_data_in [31],4
1			req1_data_in [31],3
1			req1_data_in [1],4
1	3	out_resp1 [0],7	req1_data_in [0],3
2	1	out_resp2 [0],7	req1_cmd [0:2],3
2	2	out_resp2 [0],7	req1_cmd [3],3
2	3	out_resp2 [0],7	req2_cmd [0:3],3
3	1	out_resp1 [0],9	req4_tag_in [0:7],3
4	1	out_resp1 [0],7	req1_cmd [0:3],3
4	2	out_resp1 [0],7	req2_cmd [0:3],3
4	3	out_resp1 [0],7	req3_cmd [0:3],3
4	4	out_resp1 [0],7	req4_cmd [0:3],3

Table 5.2: Refinement steps for Calculator 2 module. Each line represents a single refinement iteration. Added Vars (n, t) represents the association of node n at time t with a symbolic variable.

command. This parametric encoding allows checking all four ports of the module in a single STE run.

Assertion 1 checks that if a port P_i sends an add or sub command at cycle 3 (after reset), and the other ports send any other, or no command, then P_i receives a "good" response with the appropriate tag at cycle 7. This specification is incorrect. By adding symbolic variables for the msb of req1_data_in at times 3 and 4, STE generated a counterexample with a data overflow for port 1, which triggers an invalid response at cycle 7. $I_X((out\_resp1[0], 7))$ contains all command, tag and data inputs of all ports at different times. The smallest set of inputs for producing a counterexample is {(req1_data_in[0],3),(req1_data_in[0],4)}. RespSTE added these inputs and additional 3 input bits.

Assertion 2 sets the command sent by a port P_i to add. The msb bits of the sent data are constrained to 0 to avoid an overflow. No constraints are imposed on the commands sent by other ports. The requirement is that the output data for P_i would match the expected data. Assertion 2 fails due to an erroneous specification. The calculator gives priority to the lower indexed ports. Thus, if both ports 1 and 3 send an add command, port 3 does not receive a response at the first possible cycle. Due to the implementation of the priority queue, commands of at least 3 ports have to be definite for falsifying the assertion. $I_X((out\_resp2[0], 7))$ contains cmd, data and tag inputs of all ports at cycles 3 and 4. Out of them, RespSTE added the least number of inputs required for falsifying the assertion.

In Assertion 3, a port P_i sends an add or sub command, followed by an add command with a certain tag and data arguments, while limiting the msb of the data to 0 to avoid a possible overflow. All other ports do not send an add or sub command during this time. The requirement is that P_i receives a response with the appropriate tag value and the expected output data. $I_X((resp\_out1[0], 9))$ includes all data and tag inputs of all ports, out of which RespSTE chose the tag of port 1 at cycle 3, which is the minimum for generating a counterexample. In the counterexample, the tag values of port 1 at cycles 3 and 5 are not consecutive, and out_rep1 is "invalid". This counterexample stems from a planted design bug documented in [81]: while there supposed to be no restriction on tag ordering, commands whose tags are out of order are treated as invalid.

Assertion 4 checks that if a port sends an invalid command, it receives and invalid command response at the first possible cycle. In order to prove this assertion, and due to the implementation of the priority queue, the commands of the other ports are also required to be definite. For all the ports, $I_X(out\_resp)$ includes all the cmd, data and tag inputs at cycles 3 and 4. RespSTE added symbolic variables for the cmd inputs at cycle 3. This is the least number of variables required for verifying the assertion.

Evaluation of Results

In [79], an algorithm called *autoSTE* for automatic refinement in STE, is presented. *autoSTE* exploits the results of the STE run, as computed by *Forte*, in order to identify trajectories along which all nodes have the value X. The input nodes of these trajectories are the candidates for refinement. Heuristics are used for choosing subsets of these candidates.

Note that *autoSTE* is completely dependent on the implementation of the STE engine and its data structures, whereas RespSTE is completely independent of it. Furthermore, since *autoSTE* uses the BDDs of the STE engine, it effects the performance of the STE engine as well. This is demonstrated by the number of BDD nodes that Forte holds during the process.

We compared our experimental results with those obtained by *autoSTE*. For the sake of comparison, we used in our experiments the same parametric representation of the STE assertions as in [79]. The final results of RespSTE and its comparison with autoSTE are shown in Table 5.3.

In the comparison, $RespSTE$ shows a significant speedup in all of the assertions with respect to *autoSTE*, and up to $\times 20$ speedup in the larger ones. A significant reduction in BDD nodes is also gained in most of the assertions. For some of the assertions, RespSTE added fewer symbolic variables or required fewer refinement iterations than autoSTE. The overall performance of RespSTE was better than autoSTE even when this was not the case.

Altogether, our experiments demonstrated that using the degree of responsibility as a measure for refinement is a good choice. It provides a quantitative measure of the importance of each input to an undecided node being *X-possible*. By examining these values, we conclude that this quan-

		RespSTE					AutoSTE			
		result	Iterations	Vars	BDD Nodes	Time	Iterations	Vars	BDD Nodes	Time
CAM	1	pass	2	5	3201	2	2	5	4768	3
	2	fail	5	11	30726	5	7	11	57424	20
	3	fail	1	8	14127	3	3	13	29006	17
Calc 2	1	fail	2	5	7735	32	2	2	6241	87
	2	fail	3	8	19717	25	2	8	20134	100
	3	fail	1	8	262201	43	1	8	530733	220
	4	pass	4	16	14005	27	11	16	17323	494

Table 5.3: Experimental Results. AutoSTE is the algorithm presented in [79]. "Iterations" is the number of refinement iterations that were performed, "Time" is the total runtime in seconds until verification / falsification of the property, "Vars" is the total number of symbolic variables that were added by the refinements, and "BDD Nodes" is the number of BDD nodes used by Forte.

titative measure reflects the actual importance of the inputs in the model. The results obtained by RespSTE agree with the decisions of a user who is familiar with the circuit. When using these results for automatic refinement, the quality of the results demonstrates itself in the number of refinement iterations that were required, and in the total number of symbolic variables that were added to the antecedent.

Chapter 6

Automata Theoretic Approach to 3-Valued Model Checking

In this section we present an automata approach to 3-valued model checking, following the automata theoretic approach to LTL model checking [80]. We introduce the "unknown" *constant* value X into the model checking of LTL formulae. We use a ternary $(0, 1, X)$ encoding for the states of Kripke Structures and Büchi automata, where X is *"unknown"*. We also extend the definitions of Büchi automata to be able to handle words over the ternary alphabet. Consequently, model checking may now return either $1, 0$ or X ("unknown") as a result. The new 3-valued framework can be implemented within explicit or symbolic model checking.

The abstraction we present is similar to the 3-valued abstraction used in *symbolic trajectory evaluation* (STE) [72]. As mentioned before, STE is indeed a powerful technique, and is being used in the industry to prove large data-path blocks. However, the approach presented in this chapter supports full LTL specifications, while STE only supports a small subset of LTL, limited to describing fixed latency. From the practical point of view, writing STE assertions is labor intensive, which makes it more error prone. Additionally, STE specification cannot share the modelling of the environment of the design with other validation methods.

For automatic refinement, it is straightforward to apply the *responsibility-based refinement* method presented in Section 5.3.3 for STE. We do not elaborate on this method in this chapter.

6.1 Automata Approach for 3-Valued Model Checking

6.1.1 3-Valued Kripke Structures and LTL Semantics

In this section we define 3-valued Kripke structures. Our ternary domain is $\mathcal{T}=\{0,1,X\}$, and is a subset of $\mathcal{Q}$, defined in Section 2.8. The ternary operators are as defined in Section 2.8 for the values in $\mathcal{T}$. These operators are monotonic with respect to $\sqsupseteq$. The definition of 3-valued Kripke structures follows the definition of Kripke structures in Section 2.1.

For two ternary assignments a' and a to a set of variables V, we write $a' \sqsupseteq a$, if $\forall v \in V$, $a'(v)=0 \Rightarrow a(v)=0$, and $a'(v)=1 \Rightarrow a(v)=1$.

For two ternary functions $f : \mathcal{T}^k \rightarrow \mathcal{T}$ and $g : \mathcal{T}^k \rightarrow \mathcal{T}$, where $k \in \mathbb{N}$, we say that $f \succeq g$ if for every two assignments $d_1 \in \mathcal{T}^k$ and $d_2 \in \mathcal{T}^k$, $d_1 \sqsupseteq d_2 \ \Rightarrow f(d_1) \sqsupseteq g(d_2)$. For $F=\{f_1,\dots f_k\}$ and $G=\{g_1,\dots g_k\}$, we say that $F \succeq G$ if $\forall i, 1 \leq i \leq k$, $f_i \succeq g_i$.

Let $f : \{0,1\}^{|V|} \rightarrow \{0,1\}$ be a Boolean function, defined by means of operators $\neg, \vee, \wedge$ applied to variables in V. The ternary function $\hat{f} : \mathcal{T}^{|V|} \rightarrow \mathcal{T}$ is the function obtained from replacing the Boolean variables and operators in f by ternary ones. It immediately follows that $\hat{f} \succeq f()$. For a set of Boolean functions F, we denote $\widehat{F}$ the set of ternary functions $\widehat{F}=\{\hat{f} | f \in F\}$. It immediately follows that $\widehat{F} \succeq F$.

Given a set of atomic propositions AP, we define $AP_3=\{p=0, p=1, p=X | p \in AP\}$. Let a 3-valued Kripke structure be a tuple $\widehat{M}=\langle S, I_0, R, \widehat{L}\rangle$, where the labelling function $\widehat{L} : S \rightarrow AP_3$ is defined such that for every $s \in S$, and for every $p \in AP$, exactly one of $p=0, p=1, p=X$ is in $\widehat{L}(s)$. The rest of the definitions remain as in Section 2.1.

For a path $\pi = s_0, s_1 \dots$ and a path formula ψ, $[\pi \models \psi] \in \mathcal{T}$. The 3-valued semantics of LTL formulae is defined with respect to a 3-value Kripke structure $\widehat{M}$ as follows:

$$[\pi \models p] = d, \text{ for } p \in AP, d \in \mathcal{T} \Leftrightarrow (p=d) \in L(s_0)$$
$$[\pi \models \neg\psi_1] = \neg[\pi \models \psi_1]$$
$$[\pi \models \psi_1 \wedge \psi_2] = [\pi \models \psi_1] \wedge [\pi \models \psi_2]$$
$$[\pi \models N\psi_1] = [\pi^1 \models \psi_1]$$
$$[\pi \models \psi_1 U \psi_2] = \begin{cases} 1 & \exists j([\pi^j \models \psi_2]=1 \wedge (\forall i\ 0 \leq i < j,\ [\pi^i \models \psi_1]=1)) \\ 0 & \forall j([\pi^j \models \psi_2]=0 \vee \exists i\ 0 \leq i < j([\pi^i \models \psi_1]=0)) \\ X & otherwise \end{cases}$$

where $\wedge, \vee$ and $\neg$ are ternary operators.

Let Π be the set of all paths from an initial state. For an LTL formula $P = A\psi$, $[\widehat{M} \models P] \in \mathcal{T}$, and is defined as follows:

$$[\widehat{M} \models P] = \begin{cases} 1 & \forall \pi \in \Pi,\ [\pi \models \psi] = 1 \\ 0 & \exists \pi \in \Pi,\ [\pi \models \psi] = 0 \\ X & otherwise \end{cases}$$

6.1.2 Bisimulation of 3-Valued Kripke Structures

In this section we define bisimulation relation between 3-valued Kripke structures.

Let $M = \langle S, I_0, R, L \rangle$ and $M' = \langle S', I'_0, R', L' \rangle$ be 3-valued Kripke structures, over a set of atomic propositions AP. We say that $L'(s') \succeq L(s)$ iff for every p, $(p = 0) \in L'(s') \Rightarrow (p = 0) \in L(s)$, and $(p = 1) \in L'(s') \Rightarrow (p = 1) \in L(s)$. A relation $B \subseteq S \times S'$ is an X-bisimulation between M and M' if for every s, s', if $B(s, s')$, then the following conditions hold:
1. $L'(s') \succeq L(s)$
2. $\forall s_1 \in S,\ R(s, s_1) \Rightarrow \exists s'_1 \in S'$ such that $R'(s', s'_1) \wedge B(s_1, s'_1)$
3. $\forall s'_1 \in S',\ R'(s', s'_1) \Rightarrow \exists s_1 \in S$ such that $R(s, s_1) \wedge B(s_1, s'_1)$

We say that M' is an *abstraction* of M, denoted $M' \succeq M$, if there is an X-bisimulation relation $B \subseteq S \times S'$, for which $\forall s \in I_0,\ \exists s' \in I'_0$ such that $B(s, s')$, and $\forall s' \in I'_0,\ \exists s \in I_0$ such that $B(s, s')$.

Theorem 6.1.1 *If $M' \succeq M$ then for every LTL formula P, $[M' \models P] = 0 \Rightarrow [M \models P] = 0$, and $[M' \models P] = 1 \Rightarrow [M \models P] = 1$.*

The proof of Theorem 6.1.1 is similar to the proof of Theorem 14 in [20], and relies on the following lemmas.

Lemma 6.1.2 *For two 3-valued models $M' \succeq M$, $\forall \pi = s_0, s_1 \cdots \in M$ such that $s_0 \in I_0$, $\exists \pi' = s'_0, s'_1 \cdots \in M'$ such that $s'_0 \in I'_0$ and $\forall i \geq 0,\ B(s_i, s'_i)$*

Lemma 6.1.3 *For two 3-valued models $M' \succeq M$, $\forall \pi' = s'_0, s'_1 \cdots \in M'$ such that $s'_0 \in I'_0$, $\exists \pi = s_0, s_1 \cdots \in M$ such that $s_0 \in I_0$ and $\forall i \geq 0,\ B(s_i, s'_i)$*

6.1.3 3-Valued Abstract Circuits

Given a Boolean circuit $C = \langle \mathcal{V}, I_0, PI, F \rangle$, and the sets $PI_X \subseteq PI$ and $F_X \subseteq F$, the 3-valued abstract circuit $C' = \langle \mathcal{V}, I'_0, PI', F' \rangle$ is an abstraction of C w.r.t. PI_X, F_X, where I'_0 and F' are defined as follows. For a state $s \in I_0$, let s' be a ternary state $s'(v_i) = \begin{cases} X & f_i \in F_X \\ s(v_i) & else \end{cases}$

$$I'_0 = \{s' | s \in I_0\}$$

For $f_i \in F$, let $\hat{f}_{i_{PI_X}}$ be the function obtained from $\hat{f}_i$ by replacing each input in PI_X with the value X. We define $f'_i = \begin{cases} X & f_i \in F_X \\ \hat{f}_{i_{PI_X}} & \text{else} \end{cases}$

$$F' = \{f' | f \in F\}$$

Note that a state in C' is an assignment $s : \mathcal{V} \rightarrow \mathcal{T}$, and that for each $s \in I_0$, $s' \sqsupseteq s$. Note, also, that for or each $f_i \in F$, $f'_i \succeq f_i$, and therefore $F' \succeq F$.

For s and s', 3-valued states of a circuit, we say that s' is an *abstraction* of s, denoted $s' \succeq s$, if $s' \sqsupseteq s$. It immediately follows that $s' \succeq s \Rightarrow L(s') \succeq L(s)$.

For a circuit $C = \langle \mathcal{V}, I_0, PI, F \rangle$, let $C^1 = \langle \mathcal{V}, I_0, PI^1, F^1 \rangle$ be an abstraction of C w.r.t. $PI^1_X \subseteq PI$ and $F^1_X \subseteq F$, and let $C^2 = \langle \mathcal{V}, I_0, PI^2, F^2 \rangle$ be an abstraction of C w.r.t. $PI^2_X \subseteq PI$ and $F^2_X \subseteq F$. Let M^1 and M^2 be the Kripke structures corresponding to C^1 and C^2.

Theorem 6.1.4 *If $PI^1_X \supseteq PI^2_X$ and $F^1_X \supseteq F^2_X$, then $M^1 \succeq M^2$.*
Proof: Let $B = \{(s^1, s^2) | s^1 \succeq s^2\}$. We show that B is an X-bisimulation between M^1 and M^2.

1. $\forall (s^1, s^2) \in B$, $s^1 \succeq s^2$. Therefore, $L(s^1) \succeq L(s^2)$.

2. Given that $F^1_X \supseteq F^2_X$ and $PI^1_X \supseteq PI^2_X$, from the definitions of F^1 and F^2 it holds that for or each $f_i \in F$, $f^1_i \succeq f^2_i$. Therefore, $F^1 \succeq F^2$.

   For $s^1 \succeq s^2$ and for in^2, an assignment to PI^2, let $t^2 = F^2(s^2, in^2)$. Let $t^1 = F^1(s^1, in^1)$, where in^1 is obtained from in^2 by replacing each input in PI^1_X with X. It holds that $s^1 \succeq s^2, in^1 \succeq in^2$, and $F^1 \succeq F^2$. Therefore, $t^1 \succeq t^2$.

3. For $s^1 \succeq s^2$, and for in^1, an assignment to PI^1, let $t^1 = F^1(s^1, in^1)$. Let $t^2 = F^2(s^2, in^2)$, where in^2 is obtained from in^1 by replacing each input in $PI^1_X \setminus PI^2_X$ with an arbitrary Boolean value. It holds that $s^1 \succeq s^2, in^1 \succeq in^2$, and $F^1 \succeq F^2$. Therefore, $t^1 \succeq t^2$.

From 1,2 and 3 it holds that B is an X-bisimulation between M^1 and M^2.

The definitions of I^1_0 and I^2_0 follow the definition of 3-valued circuit given above, and depend on F^1_X, F^2_X, and I_0. Since $F^1_X \supseteq F^2_X$, it immediately follows that $\forall s^1 \in I^1_0, \exists s^2 \in I^2_0$ such that $s^1 \succeq s^2$, and $\forall s^2 \in I^2_0, \exists s^1 \in I^1_0$ such that $s^1 \succeq s^2$. Therefore, $\forall s^1 \in I_0, \exists s^2 \in I_0$ such that $(s^1, s^2) \in B$, and $\forall s^2 \in I_0, \exists s^1 \in I_0$ such that $(s^1, s^2) \in B$.

We thus conclude that $M^1 \succeq M^2$. $\square$

6.1.4 3-Valued Büchi Automata

Given a Büchi automaton $\mathcal{B}$ over AP_2, we construct a 3-valued Büchi automaton $\mathcal{B}'$ over AP_3.

For a set of atomic propositions AP, let $\Sigma = \mathcal{P}(AP_2)$. For a set of state variables Y, and $D = \{1 \ldots 2^{|Y|}\}$, let $\mathcal{B} = \langle \Sigma, Q, q_{in}, \rho, \alpha \rangle$ be a Büchi automaton, where $Q = 2^Y$. The acceptance condition α can be considered as a function $\alpha : 2^Y \rightarrow \{0,1\}$, which evaluates to 1 for fair states in $\mathcal{B}$. Let F be the set of functions $f_{y_i} : Q \times \Sigma \times D \rightarrow Q$, corresponding to ρ, as described in Section 2.5.1.

Let $\Sigma' = \mathcal{P}(AP_3)$, and $Q' = \mathcal{T}^Y$. Let ρ' be the transition function corresponding to $\widehat{F}$. Note that $\widehat{F}$ is a set of functions $f'_{y_i} : Q' \times \Sigma' \times D' \rightarrow Q'$, where $D' = \{1 \dots 3^{|Y|}\}$. The 3-valued Büchi automaton for $\mathcal{B}$ is the tuple $\mathcal{B}' = \langle \Sigma', Q', q_{in}, \rho', \hat{\alpha} \rangle$.

6.1.5 Model Checking of 3-Valued Circuits

In this section we show how 3-valued Büchi automata can be used for checking 3-valued circuits.

Let $\widehat{C} = \langle \widehat{V}, \widehat{I_0}, \widehat{PI}, \widehat{F_C} \rangle$ be an abstraction of a circuit $C = \langle \mathcal{V}, I_0, PI, F_C \rangle$. Let $M = \langle S, I_0, R, L \rangle$ be the Kripke structure corresponding to C, and let $\widehat{M} = \langle \widehat{S}, \widehat{I_0}, \widehat{R}, \widehat{L} \rangle$ be the Kripke structure corresponding to $\widehat{C}$. For an LTL property $P = A\psi$, let $B_{\neg\psi} = \langle \Sigma, Q_B, q_{in}, \rho, \alpha \rangle$ be the Büchi automaton for $\neg\psi$, and let $\widehat{B}_{\neg\psi} = \langle \widehat{\Sigma}, \widehat{Q}_B, q_{in}, \hat{\rho}, \hat{\alpha} \rangle$ be the 3-valued Büchi automaton for B. Let F_B and $\widehat{F}_B$ be the sets of functions corresponding to ρ and $\hat{\rho}$. Let E, be the product of M and $B_{\neg\psi}$, as defined in Section 2.5.3. Similarly, let $\widehat{E} = \langle \widehat{S_E}, \widehat{I_0^E}, \widehat{R_E}, \widehat{L_E}, \widehat{\alpha_E} \rangle$ be the product of $\widehat{M}$ and $\widehat{\mathcal{B}}_{\neg\psi}$. Recall that a state e in E is a pair (s, q) where $s \in S$ and $q \in Q_B$. Similarly, a state $\hat{e}$ in $\widehat{E}$ is a pair $(\hat{s}, \hat{q})$.

A *strongly connected component* (SCC) $\odot$ in a graph is a set of nodes such that there is a path within $\odot$ between every two nodes in $\odot$. Note that since the number of states in E and $\widehat{E}$ is finite, every infinite path π in E or $\widehat{E}$ consists of a finite prefix π_s connected to an SCC $\odot$. We denote this by $\pi = \pi_s \cdot \odot$.

A *lasso* in $\widehat{E}$ is a path $\hat{\pi} = \hat{e}_{l_0} \dots \hat{e}_{l_k}$, where $\hat{e}_{l_0} \in \widehat{I_0^E}$, and there exists some $i, 0 \leq i < k$ such that $\hat{e}_{l_k} = \hat{e}_{l_i}$. We denote a lasso by $\hat{\pi}_i \cdot \bigcirc$, where $\hat{\pi}_i$ is the finite prefix $\hat{e}_{i_0} \dots \hat{e}_{l_i}$, and $\bigcirc$ is the cycle $\hat{e}_{l_{i+1}} \dots \hat{e}_{l_k}$. A 1-lasso is a lasso $\hat{\pi}_c \cdot \bigcirc$ where there exists $\hat{e}_{l_a} \in \bigcirc$ such that $\hat{\alpha}(\hat{e}_{l_a}) = 1$. Similarly, in an X-lasso, $\hat{\alpha}(\hat{e}_{l_a}) = X$.

For a path $\pi = e_{l_0} \dots e_{l_k}$, $first(\pi)$ is e_{l_0}, and $last(\pi)$ is e_{l_k}.

Next we reduce checking $\widehat{M}$ with respect to ψ to finding lassos in $\widehat{E}$.

Theorem 6.1.5
(1) If there exists a 1*-lasso in $\widehat{E}$, then there exists a fair path in E.*
(2)If there is no 1*-lasso and no X-lasso in $\widehat{E}$, then there is no fair path in E.*

Proof:

(1) We first prove that if there exists a 1-lasso in $\widehat{E}$, then there exists a fair path in E.

Lemma 6.1.6 *For every* $(\hat{e}, \hat{e}') \in \widehat{R}_E$*, and for every* $e \in S_E$*, such that* $\hat{e} \succeq e$*, there exists* $e' \in S_E$*, such that* $(e, e') \in R_E$*, and* $\hat{e}' \succeq e'$*.*

Proof of Lemma 6.1.6: For $\hat{e} = (\hat{s}, \hat{q})$ and $\hat{e}' = (\hat{s}', \hat{q}')$ such that $(\hat{e}, \hat{e}') \in \widehat{R}_E$, let $e = (s, q)$ such that $\hat{s} \succeq s$ and $\hat{q} \succeq q$.

Since $\hat{q}' \in \hat{\rho}(\hat{q})$, there exists a number c such that $\hat{q}' = \widehat{F}_B(\hat{q}, \hat{s}, c)$. Let $q' = F_B(q, s, c)$. $\widehat{F}_B \succeq F_B$, $\hat{q} \succeq q$, and $\hat{s} \succeq s$. Therefore $\hat{q}' \succeq q'$.

Since $(\hat{s}, \hat{s}') \in \widehat{R}_M$, and $\widehat{M}$ is defined for $\widehat{C}$, there exists some assignment $\widehat{in}$ to the inputs to $\widehat{C}$ such that $\hat{s}' = \widehat{F}_C(\hat{s}, \widehat{in})$. For some $in \in 2^{PI}$ such that $\widehat{in} \succeq in$, let $s' = F_C(s, in)$. $\widehat{F}_C \succeq F_C$, $\hat{s} \succeq s$, and $\widehat{in} \succeq in$. Therefore, $\hat{s}' \succeq s'$. Thus, $\hat{e}' \succeq e'$, and the lemma holds. □

From Lemma 6.1.6 it holds that for every finite path $\hat{\pi} = \hat{e}_{l_0} \ldots \hat{e}_{l_n}$ in $\widehat{E}$, there exits a corresponding finite path $\pi = e_{m_0} \ldots e_{m_n}$ in E, such that $\forall i, \hat{e}_{l_i} \succeq e_{m_i}$.

Next we show that for every 1-lasso in $\widehat{E}$ there exists a corresponding 1-lasso in E.

Let $\hat{\pi} = \hat{\pi}_s \cdot \bigcirc$ be a 1-lasso in $\widehat{E}$, where $\hat{s}_0 = first(\hat{\pi}_s)$. $\hat{s}_0$ was obtained from a state $s_0 \in I_0$, by replacing some of the Boolean assignments to the state variables with the value X. Therefore, $\hat{s}_0 \succeq s_0$. From Lemma 6.1.6 it holds that there is a finite path π_s in E, corresponding to $\hat{\pi}_s$, where $first(\pi_s) = s_0$, and therefore $first(\pi_s) \in I_0^E$. From Lemma 6.1.6 it holds that there is a path in E, corresponding to $\bigcirc$. However, this path may not form a cycle in E, and may not be unique. Next we show how a cycle in E that corresponds to $\bigcirc$ can be created. For that purpose, we concatenate paths that correspond to $\bigcirc$ in E. Since there is a finite number of states in E, the number of concatenations is finite, and must result in a cycle in E. Formally, let Π_c be the set of all finite paths π_c^i that correspond to $\bigcirc$ in E. Let the path π_c be a concatenation of paths π_c^i from Π_c, such that $first(\pi_c^0) = last(\pi_s)$, and for every i, $first(\pi_c^{i+1}) = last(\pi_c^i)$. Since there is a finite number of states in E, there is a finite number of states e_{n_k} such that $\hat{e}_{l_k} \succeq e_{n_k}$ in E. Therefore, there exists a finite number j such that the path $\pi_c = \pi_c^0 \ldots \pi_c^j$ is a cycle in E.

Let $\hat{e}_{l_a} \in \bigcirc$ be a state in $\widehat{E}$ such that $\widehat{\alpha_E}(\hat{e}_{l_a}) = 1$. For every π_c^i, let e_{n_a} be the corresponding state to $\hat{e}_{l_a}$. $\widehat{\alpha_E} \succeq \alpha_E$, and $\hat{e}_{l_a} \succeq e_{n_a}$. Therefore, $\alpha_E(e_{n_a}) = 1$.

We conclude that π_c is a cycle in E which includes a fair state. Let the infinite path π in E, be the concatenation of π_s and π_c. π is a fair path in E.

(2) Next we show that if there is no 1-lasso and no X-lasso in $\widehat{E}$, then there is no fair path in E.

Lemma 6.1.7 *If there is no* 1*-lasso and no* X*-lasso in* $\widehat{E}$*, then for every path* $\hat{\pi} = \hat{\pi}_s \cdot \odot$ *in* $\widehat{E}$*,* $\forall \hat{e} \in \odot[\widehat{\alpha_E}(\hat{e}) = 0]$
Proof of Lemma 6.1.7: Assume to the contrary that there is no 1-lasso and no X-lasso in $\widehat{E}$, but there is some path $\hat{\pi} = \hat{\pi}_s \cdot \odot$ in $\widehat{E}$, and a state $\hat{e}_a \in \odot$ such that $\widehat{\alpha_E}(\hat{e}_a) \neq 0$. Let $\hat{\pi}_1$ be a header of $\hat{\pi}$ that ends in $\hat{e}_a$. Since $\hat{e}_a \in \odot$, then there is a cycle $\hat{\pi}_2 = \hat{e}_a \ldots \hat{e}_a$. The result of the concatenation of $\hat{\pi}_1$ and $\hat{\pi}_2$ is a 1-lasso or an X-lasso in $\widehat{E}$, depending on the value of $\widehat{\alpha_E}(\hat{e}_a)$. Thus, the assumption is contradicted, and the Lemma holds □

Lemma 6.1.8 *For a path* $\pi = e_{m_0}, e_{m_1} \ldots$ *in* E*, where* $e_{m_0} \in I_0^E$*, there exits a path* $\hat{\pi} = \hat{e}_{l_0}, \hat{e}_{l_1} \ldots$ *in* $\widehat{E}$*, where* $\hat{e}_{l_0} \in \widehat{I_0^E}$*, such that* $\forall i(\hat{e}_{l_i} \succeq e_{m_i})$*.*

Proof of Lemma 6.1.8: We prove Lemma 6.1.8 by induction on the length of π.

- For paths of length 0, $e_{m_0} \in I_0^E$. Therefore, e_{m_o} is of the form $e_{m_o} = (s_{m_0}, q_{in})$, where $s_{m_0} \in I_0$. Let $\hat{s}_0 \in \widehat{I_0}$ be the corresponding state to s_0, with respect the abstraction between M and $\widehat{M}$. The state $\hat{e}_{m_0} = (\hat{s}_0, q_{in})$ is in I_0^E. Since $\hat{s}_0 \succeq s_0$, it holds that $\hat{e}_{m_0} \succeq e_{m_0}$.
- Assuming that the proposition is correct for paths of length n, we show that it is correct for paths of length $n+1$.

  Let $e_{m_n} = (s, q)$, $e_{m_{n+1}} = (s', q')$ and $\hat{e}_{l_n} = (\hat{s}, \hat{q})$. There exists a number c such that $q' = F_B(q, s, c)$. Let $\hat{q}' = \widehat{F}_B(\hat{q}, \hat{s}, c)$. $\widehat{F}_B \succeq F_B$, $\hat{s} \succeq s$, and $\hat{q} \succeq q$. Therefore, $\hat{q}' \succeq q'$.

  There exists some assignment *in* to the inputs to C such that $s' = F_C(s, in)$. Let $\hat{s}' = \widehat{F}_C(\hat{s}, in)$. $\widehat{F}_C \succeq F_C$, and $\hat{s} \succeq s$. Therefore, $\hat{s}' \succeq s'$.

  $\hat{q}' = \widehat{F}_B(\hat{q}, \hat{s}, c)$, and $\hat{s}' = \widehat{F}_C(\hat{s}, in)$. Therefore, for the state $\hat{e}_{l_{n+1}} = (\hat{s}', \hat{q}')$ it holds that $(\hat{e}_{l_n}, \hat{e}_{l_{n+1}}) \in \widehat{R_E}$.

  $\hat{s}' \succeq s'$ and $\hat{q}' \succeq q'$. We thus conclude that $\hat{e}_{l_{n+1}} \succeq e_{m_{n+1}}$.

From the above we conclude that the Lemma holds. □

Lemma 6.1.9 *If for every path* $\hat{\pi} = \hat{\pi}_s \cdot \odot$ *in* $\widehat{E}$*,* $\forall \hat{q} \in \odot[\hat{\alpha}(\hat{q}) = 0]$*, then there is no fair path in* E*.*

Proof of Lemma 6.1.9: Assume to the contrary that for every path $\hat{\pi} = \hat{\pi}_s \cdot \odot$ in $\widehat{E}$, $\forall \hat{q} \in \odot[\hat{\alpha}(\hat{q}) = 0]$, but there is a fair path $\pi = q_{m_0}, q_{m_1} \ldots$ in E.

Let $\pi = q_{m_0}, q_{m_1} \ldots$ be a fair path in E, and let $\hat{\pi} = \hat{q}_{l_0}, \hat{q}_{l_1} \ldots$ be its corresponding path in $\widehat{E}$. There are infinitely many states q_{m_i} in π such that $\alpha_E(q_{m_i}) = 1$. From the monotonicity of $\widehat{\alpha_E}$ it holds that either

$\widehat{\alpha_E}(\hat{q}_{l_i}) = 1$ or $\widehat{\alpha_E}(\hat{q}_{m_i}) = X$. This contradicts the assumption that for every path $\hat{\pi} = \hat{\pi}_s \cdot \odot$ in $\widehat{E}$, $\forall \hat{q} \in \odot[\hat{\alpha}(\hat{q}) = 0]$, and the Lemma holds $\square$

From Lemma 6.1.7 and Lemma 6.1.9 it holds that if there is no 1-lasso and no X-lasso in $\widehat{E}$, then there is no fair path in E. This concludes the proof of Theorem 6.1.5. $\square$

As a result of theorem 6.1.5, checking $\widehat{C}$ is now reduced to finding 1-lasso and X-lasso in $\widehat{E}$. Note that this result is independent of the model checking method that is used. In the next section we show how checking $\widehat{C}$ is done by 3-valued BMC

6.2 3-Valued Bounded Model Checking

As discussed in Section 2, *Bounded model checking* (BMC) is one of the most efficient methods for formal verification, but even state of the art BMC tools cannot cope with today's large CPU blocks. Thus, verification engineers are required to extract small enough designs under test (DUT) for formal verification. This involves the application of various model reductions and abstractions. On large industrial CPU designs, these tasks usually cannot be carried out entirely by automated tools. In most cases it requires a lot of manual work, and close familiarity of the verification engineer with the design. A significant effort is also spent on modelling an environment for the DUT, and on identifying and debugging false negatives, which are mainly due to over-approximating abstractions. Since the DUT is low level, it usually includes internal signals and interfaces which may change often during the development and verification cycle of an industrial design. Thus, keeping up with even minor changes in the design may require costly maintenance of the formal verification environment.

In order to keep up with the growing design complexity, users must be able to apply formal verification on large designs. When working on the natural block boundaries, as opposed to working on a small DUT, the interfaces are well defined, and modelling the environment is simpler. Working at this level enables proving higher level properties, the cost of maintenance drops significantly, and proofs become more portable. Additionally, the environment for the block boundaries can be shared with other validation techniques (e.g. simulation).

In this section we show how 3-valued model checking enables applying formal verification to very large industrial designs, using their natural interfaces instead of extracting small DUTs. Given a large design and an LTL property, the verification engineer chooses a set of inputs to the circuit that are in the cone of influence of the property being checked, but are irrelevant to it. These inputs are assigned the value X. The Xs propagate through the design and eliminate irrelevant parts of it, and thereby simplify the model checking problem. This is done without defining a DUT explicitly or accu-

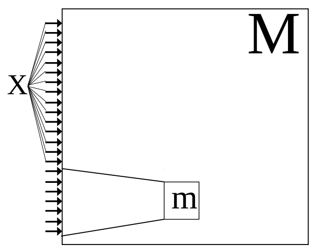


Figure 6.1: High Level Checking

rately. Choosing irrelevant inputs can be done automatically (e.g. by static analysis) or manually. However, even if done manually, the user is only required to be familiar with the interface of the large design, rather than with its internal details. This interface is likely to be well defined, and it does not change during the development and verification cycle. We demonstrate this in Figure 6.1. Instead of extracting the small unit m out of the design M, we assign some of the inputs to M with X. This cuts out irrelevant parts of M without extracting m explicitly. The user does not have to be familiar with the internal details of M, and does not have to model the environment of m, which is already implemented in M. Our method returns $true$ or $false$ if the property holds or fails on the design, or X if it is impossible to determine whether the property holds or not, due to too many X's assigned to inputs.

We developed our technique within Intel's BMC framework, implemented on top of a state-of-the-art CNF SAT solver [60]. We refer to this implementation as X-BMC. We used X-BMC for checking real life assertions on next-generation CPU designs currently being developed, as well as on a set of smaller benchmarks. In our experiments we obtained outstanding results. We were able to run verification on huge designs, up to an order of magnitude larger than the models that current industrial-strength methods can handle.

In addition to the advantages mentioned above, X-BMC can enhance the performance of standard formal verification methods that do involve extracting DUTs. Extracting a DUT from a large design is typically done by defining a set of cut-points, and removing the logic that drives them. This task is based on various manual and automatic abstraction methods such as *localization reduction* [47]. The user then has to build an environment to model the behavior of the internal cut-points, or leave them as free inputs to the DUT. X-BMC, on the other hand, does not leave cut-points as free inputs, but rather assigns them the constant X. This way, the DUT is further simplified, as the Xs propagate through the model. This leads to a significant speedup in the checking procedure, both in terms of faster iterations, and fewer iterations required for finding a bug. X-BMC also gives immediate indication of too coarse abstractions, and enables tracing the reason for an X result returned by model checking.

6.2.1 Related Work

The common approach to abstraction and refinement for BMC of circuits is *localization reduction*. Various methods based on this approach were suggested in works such as [34, 51, 36] and [50]. In these methods, abstract models are created by choosing cut-points in the circuit, and removing the logic that drives them, leaving them as free inputs to the new abstract circuit. This way, a reduction in the size of the model is achieved. These methods only allow internal nodes to be cut-points. Therefore, they do not allow defining the abstraction on the boundaries of the design that is being checked, as with X-BMC. Additionally, X-BMC enhances these abstraction methods by assigning X to cut-points. As explained above, this results in a further simplified model, which leads to faster and fewer checking iterations. Leaving the cut-points as free inputs to the abstract models also results in an over-approximating abstraction, which may yield false negative results. X-BMC, on the other hand, never returns a false negative result, but rather returns X as an indication of a too coarse abstraction.

Works such as [88, 64] gain reduction in the size of circuits by looking for *observability don't cares.* These are nodes in the cone of influence of the property being checked, that have no effect on whether it is satisfied by the circuit or not. However, the method for finding such nodes imposes a significant computational overhead. It is only roughly under-approximated, and is not feasible for very large designs.

6.2.2 X-BMC

X-BMC is an iterative algorithm, similar to BMC, for checking bounded paths in an abstract model. In each iteration of X-BMC, all the paths of length k in an abstract model $\widehat{M}$ are checked against an LTL property P. If a bug is found, X-BMC returns 0. Otherwise, X-BMC checks if there is any run of $\widehat{M}$ for which it is impossible to decide if P holds. If no such path exists, X-BMC concludes that there is no erroneous path of length k in $\widehat{M}$, and the bound is increased. If such a path exists, the model is refined. Alternatively, it is possible to increase the length k, and refine the abstract model only if a bug is not found in longer paths.

The abstraction in X-BMC is derived by choosing a set of cut-points in the circuit. These nodes may be chosen by any abstraction scheme, either automatic, manual, or a combination of the two, allowing the user to incorporate his knowledge about the model into the abstraction. X-BMC replaces the functions corresponding to the cut-points with the constant value

X, thus removing the logic that drives them. Inputs to the circuit may also be chosen as cut-points. Assigning these inputs with the constant X and propagating their value forward abstracts out parts of the circuit that depend on them. This is obtained without defining the abstracted parts explicitly or accurately.

6.2.3 X-BMC: Implementation

For a circuit C, an abstraction $\widehat{C}$ for C, and an LTL formula $P = A\psi$, let M be the Kripke structure corresponding to C, let $\widehat{M}$ be the Kripke structure corresponding to $\widehat{C}$, and let $\widehat{B}_{\neg\psi}$ be the Büchi automaton corresponding to $\neg\psi$. Let $\widehat{E}$ be the product of $\widehat{M}$ and $\widehat{B}_{\neg\psi}$, as defined in Section 6.1.5.

Let Y be a set of state variables, representing the states of $\widehat{E}$. For $\hat{e}, \hat{e}' \in \widehat{S_E}$, we say that $\hat{e}' =_b \hat{e}$ iff $\forall y_i \in Y[\hat{e}'(y_i) =_b \hat{e}(y_i)]$. By using the operator $=_b$ we can conclude that two ternary assignments to Y represent the same state in $\widehat{E}$.

We construct the ternary formulae $\widehat{fair_i}, \widehat{\varphi_{\pi i}}$, and $\widehat{\varphi_{ei}}$ following the construction of $fair_i, \varphi_{ei}$, and $\varphi_{\pi i}$ given in Equations 2.2, 2.3 and 2.4. Note that since we are comparing abstract states in $\widehat{E}$, we replace $(e_l = e_i)$ in $fair_i$, by $(\hat{e}_l =_b \hat{e}_i)$ in $\widehat{fair_i}$.

$$\widehat{fair_i}(e_0 \dots e_i) = \bigvee_{0 \le l < i} \left((\hat{e}_l = \hat{e}_i) \wedge \bigvee_{l \le j < i} \widehat{\alpha_E}(\hat{e}_j) \right) \tag{6.1}$$

$$\widehat{\varphi_{ei}}(\hat{e}_0 \dots \hat{e}_i) = \widehat{I_0^{BE}}(\hat{e}_0) \wedge \bigwedge_{0 \le j < i} \widehat{R_B^E}(\hat{e}_j, \hat{e}_{j+1}) \wedge \widehat{fair_i}(\hat{e}_0 \dots \hat{e}_i) \tag{6.2}$$

$$\widehat{\varphi_{\pi i}}(\hat{e}_0 \dots \hat{e}_i) = \widehat{I_0^{ME}}(\hat{e}_0) \wedge \bigwedge_{0 \le j < i} \widehat{R_M^E}(\hat{e}_j, \hat{e}_{j+1}) \tag{6.3}$$

We define

$$\hat{\varphi}_{(1,i)} = (\widehat{\varphi_{\pi i}} =_b 1) \wedge (\widehat{\varphi_{ei}} =_b 1)$$
$$\hat{\varphi}_{(X,i)} = (\widehat{\varphi_{\pi i}} =_b 1) \wedge (\widehat{\varphi_{ei}} =_b X)$$

Note that $\hat{\varphi}_{(1,i)}$ and $\hat{\varphi}_{(X,i)}$ are Boolean, and their satisfiability can be checked by a SAT solver. An assignment satisfying $\hat{\varphi}_{(1,i)}$ represents a 1-lasso in $\widehat{E}$. An assignment satisfying $\hat{\varphi}_{(X,i)}$ represents an X-lasso in $\widehat{E}$.

The pseudo code for X-BMC is given in Figure 6.2, and complies with Theorem 6.1.5. The SAT calls in lines 3 and 5 search for 1-lasso and X- lasso in $\widehat{E}$. A satisfying assignment found by the SAT call in line 3 represents a 1-lasso in $\widehat{E}$, and thus represent a counterexample in M. If neither of the SAT calls in line 3 and 5 return a satisfying assignment, then there are no 1-lasso and no X-lasso in $\widehat{E}$, and thus it is definite that there is no counterexample of the current length in M. If there is no 1-lasso, but

```
0. $X - BMC(\widehat{M}, P)$ {
1.   $k \leftarrow 0$
2.   while($true$) {
3.     if $SAT(\hat{\varphi}_{(1,k)})$
4.       return false
5.     if $SAT(\hat{\varphi}_{(X,k)})$
6.       refine $\widehat{M}$ or $inc(k)$
7.     $inc(k)$
8.   }
9. }
```

Figure 6.2: X-BMC

there is an X-lasso, then the result of X-BMC is X. This is due to $\widehat{M}$ not containing enough information for reaching a definite conclusion. At this point (line 6), $\widehat{M}$ should be refined, in order to try and get a definite result, or k should be increased, trying to find a deeper counterexample. Note that the satisfying assignment for $\varphi'_{(X,i)}$ is in fact a trace that leads to the unknown result due to the X's in the cut-points.

6.2.4 Experimental Results

For evaluating $X-BMC$ we incorporated it into Intel's framework for BMC, implemented on top of the state of the art CNF SAT solver [60].

We conducted our first experiments on IBM's Calculator 2 design [81], a few arithmetic modules, and the Content Addressable Memory (CAM) module from Intel's GSTE tutorial. These designs have 1150 to 2850 latches. All of these experiments use dedicated computers with a 3.2Ghz Intel Xeon CPU and 8GB RAM.

The results of these experiments are given in the tables in Figure 6.3(a,b). The assertions in (a) hold up to the computed bound, whereas the assertions in Figure (b) fail. The columns "BMC Depth" and "X-BMC Depth" present the bound that was reached by each algorithm with a timeout of 1 hour. For a given bound reached by BMC, the column "Speedup to BMC Depth" in (a) shows the speedup of X-BMC over BMC for reaching this bound. In cases where BMC reaches deeper than X-BMC, *speedup* < 1. The column "Speedup" in (b) presents the speedup of X-BMC over BMC for falsifying the properties.

For most of the properties that we checked, X-BMC significantly outperformed BMC both in terms of depth reached and in run times. These experiments also show that the overhead incurred by the dual rail encoding was negligible relative to the advantages of X-BMC.

Next we discuss running X-BMC on a very large industrial design. For

	Pass Model	BMC Depth	X-BMC Depth	Speedup to BMC depth
1	CAM	432	20	0.022
2		27	21	0.029
3		51	13	0.046
4	Calc2	16	21	23.2
5		11	14	8.6
6	ar1_32_b	112	142	17.1
7		46	63	3.9
8		88	88	2.3
9	ar1_64_b	37	50	19.1
10		20	24	11.7
11		5	149	96.8
12	bar_32_b	45	48	12.3
13		15	23	32.3

(a) Properties verified up to the reached bound

	Fail		BMC	X-BMC	
	Model	Depth	Time	Time	Speedup
14	CAM	12	25	107	0.23
15		36	226	431	0.52
16	Calc2	19	35	30	1.17
17		14	87	95	0.92
18	ar1_32_b	9	7	12	0.58
19		9	10	13	0.77
20	ar1_64_b	17	431	323	1.33
21		16	> 1hr	1778	----
22		5	21	26	0.81
23		15	108	79	1.37
24	bar_32_b	14	2241	209	10.72
25		15	> 1hr	2610	----
26		25	> 1hr	3103	----
27		50	> 1hr	> 1hr	----

(b) Failed Properties

		Model	ALU	Abs 1	Abs 2	Abs 3	Abs 4	Abs 5
		# Latches	133K	132K	115K	108K	74K	71K
		# Gates	6.1M	6.0M	5.9M	5.8M	0.6M	0.5M
	P	Result	Run Time (s)					
XBMC	P1	fail	266	281	270	254	103	105
	P2	pass	262	271	265	244	212	205
	P3	fail	264	280	249	282	285	103
	P4	pass	412	365	342	323	X	X
	P5	fail	278	267	252	264	110	108
	P6	pass	654	640	631	615	587	552
BMC	P1	fail	M/O	M/O	M/O	12280	525	168
	P2	pass	M/O	M/O	M/O	479	411	235
	P3	fail	M/O	M/O	M/O	M/O	M/O	408
	P4	F/N	M/O	M/O	M/O	M/O	F/N	F/N
	P5	fail	M/O	M/O	M/O	M/O	908	632
	P6	pass	M/O	M/O	M/O	M/O	2241	199

(c) Large ALU Verification

Figure 6.3: X-BMC run time. (a),(b) The number of latches in CAM, Calc2, ar1_21_b, ar_64_b, and bar_32_b is $1152, 2781, 572, 1914$, and 2813 respectively. (c) Checking a large ALU on a 32GB RAM machine. P stands for property, M/O stands for memory out, and F/N stands for false negative. Abs 1 to Abs 5 are abstractions of the ALU, where Abs 5 is the smallest abstract model. #FF and #Gates are the number of flip flops (in thousands) and the number of combinational gates (in millions) in each model.

these experiments we used a large ALU from a core of a next-generation CPU, which is currently being developed at Intel. All experiments used a dedicated machine with an Intel Xeon 3Ghz CPU and 32GB RAM.

We checked the design against 6 different properties, all to depth 32. Since there is no automatic BMC-based tool that can handle such a large design, we followed the common methodology in the industry, and cut out parts of the design until a solvable instance was reached. This was done manually, based on our familiarity with the model. Each column in the table describes a smaller abstract model, where ALU is the complete design, and Abs 5 is the smallest model required for getting concrete results. The sizes of the models that are shown in the table include only latches and gates that are in the cone of influence of the checked property.

Figure 6.3(c) shows the results of using BMC and X-BMC on the different models. As seen in the table, BMC is capable of handling only the smallest abstract models that were extracted from the original design, and suffers from a memory blowout on the larger ones. That is, using BMC requires a significant manual work for achieving solvable DUTs. In contrast, X-BMC could tackle the whole ALU, simply by assigning some of its *inputs* to be X. This is a relatively simple task at the level of the whole ALU, where the interface is simple and well defined, and it does not require familiarity with the design itself. Note that the ALU is an order of magnitude larger than models that are handled by the methods discussed in Section 4.4.

Apart from the methodological difference, X-BMC also shows a significant speedup in run-times over the abstract models that BMC could solve. This shows that assigning cut-points with the constant X simplifies the work of the SAT solver, as opposed to assigning them with free variables. Note that for each counterexample returned by BMC for the abstract models, we have to verify that it is not spurious. We did not add the time required for this check to the run times of BMC. These checks are not required in X-BMC, which returns X if the model is too abstract. This indeed happens in $P4$, where BMC returns a spurious counterexample, while X-BMC returns X. In that case, Abs3 is used for obtaining a concrete result in X-BMC, whereas BMC can not obtain a concrete result.

In order to avoid masking of marginal cases, we conducted the same experiments on a machine with 64GB RAM, and a timeout of 10 hours. The results of these experiments are presented in Figure 6.4. These experiments show that the run times of BMC grows exponentially with the size of the model, while the additional memory allows only a few additional abstract models to be handled. In contrast, X-BMC is almost insensitive to the addition of irrelevant parts of the model.

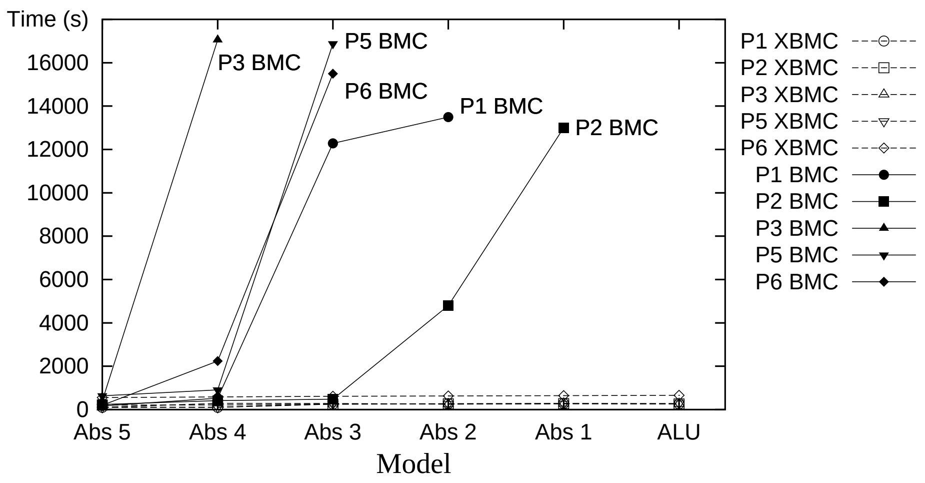


Figure 6.4: BMC and X-BMC run times on a 64GB RAM machine. Abs 1… Abs 5 are abstractions of ALU.

6.3 3-Valued Unbounded Model Checking

Let C be a circuit, and let $P = AGexp$ be a safety property, where exp is a Boolean expression over atomic formulae. Let $\widehat{C}$ be an abstraction of C, and let $\widehat{M} = \langle \widehat{S}, \widehat{I_0}, \widehat{R}, \widehat{L} \rangle$ be the Kripke structure corresponding to $\widehat{C}$. We follow the definitions in equations 2.6, 2.7 and 2.8.

$$\widehat{\varphi_b^k}(\hat{e}_0 \dots \hat{e}_k) = \widehat{I_0^E}(\hat{e}_0) \wedge \bigwedge_{i=0}^{k-1} \widehat{R}(\hat{e}_i, \hat{e}_{i+1}) \wedge \bigvee_{i=0}^{k} \neg\widehat{exp}(\hat{e}_i) \tag{6.4}$$

$$\widehat{loop\_free}^k(\hat{e}_0 \dots \hat{e}_k) = \bigwedge_{\substack{0 \le i,j \le k \\ i \ne j}} \neg(\hat{e}_i = \hat{e}_j) \tag{6.5}$$

$$\widehat{\varphi_{ind}^k}(\hat{e}_0 \dots \hat{e}_k) = \bigwedge_{i=0}^{k} \Big(\widehat{R_E}(\hat{e}_i, \hat{e}_{i+1}) \wedge \widehat{exp}(\hat{e}_i)\Big) \wedge \widehat{loop\_free}^k(\hat{e}_0 \dots \hat{e}_k) \wedge \neg\widehat{exp}(\hat{e}_{k+1}) \tag{6.6}$$

The pseudo code of our 3-valued UBMC algorithm is given in Figure 6.5. At iteration k, satisfiability of $\widehat{\varphi_b^k} =_b 1$ implies that there is a path of length k from an initial state in $\widehat{M}$, where $\neg\widehat{exp}$ is 1, and therefore $[\widehat{M} \models P] = 0$. In lines $8, 9$ the algorithm proceeds to the next iteration. This is the case where there is no erroneous path of length k from an initial state in $\widehat{M}$, but k consequent states where $\widehat{exp}$ is 1 do not yet imply that $\widehat{exp}$ is 1 in the next state as well. If $\widehat{\varphi_b^k} =_b X$ or $\widehat{\varphi_{ind}^k} =_b X$ are satisfiable, then there is not

```
0. $X - UBMC(\widehat{M}, P)$
1.     $k \leftarrow 0$
2.     while(1) {
3.       if $SAT(\widehat{\varphi_b^k} =_b 1)$
4.         return false
5.       if $SAT(\widehat{\varphi_b^k} =_b X)$
6.         refine $\widehat{M}$
7.       if $SAT(\widehat{\varphi_{ind}^k} =_b 1)$
8.         $inc(k)$
9.         do next iteration
10.      if $SAT(\widehat{\varphi_{ind}^k} =_b X)$
11.        refine $\widehat{M}$
12.      return true
13.    }
```

Figure 6.5: M is a model, $M' \succeq M$, and P is a safety property.

enough information on paths of length k, and $\widehat{M}$ has to be refined. If none of the formulae is satisfiable, then $[\widehat{M} \models P] = 1$.

Chapter 7

Conclusion

We presented new approaches to symbolic model checking of hardware models.

We presented SAT and all-SAT algorithms for model checking. Our algorithms use propositional representation, as well as a graph representation, of the models. We also adapt our SAT algorithm for solving 3-valued problems. Our SAT and all-SAT methods allow us to increase the performance of various model checking algorithms, and solve problems more efficiently than was done before. We also combine SAT based algorithms with BDD representation, and exploit the benefits of both approaches.

We presented different methods for automatic refinement in STE, instead of the common manual refinement. Our SAT based automatic refinement complements our SAT based STE algorithm. Our responsibility based STE refinement is independent of the STE method, and gives high quality results. Automatic refinement requires far less user effort and expertise, which are the current bottleneck in practical usage of STE.

Last, we presented an automata theoretic approach to 3-Valued model checking. We defined 3-valued models, and gave 3-valued semantics to the LTL specification language. Our approach can be used for explicit or symbolic model checking.

We implemented this approach in 3-valued *Bounded Model Checking*. This implementation allowed us to propose a new methodology for BMC of very large circuits. In our methodology, a lot of the manual work that is performed in the common approach to BMC is avoided, by allowing the user to generate efficient abstract models without explicitly defining them. Our 3-valued implementation also speeds up the SAT solving process in BMC. Therefore, it can also be used to speedup the SAT solving process when other, either manual or automatic, abstraction methodologies for BMC are used.

Bibliography

[1] Sara Adams, Magnus Bjork, Tom Melham, and Carl Seger. Automatic Abstraction in Symbolic Trajectory Evaluation . In *FMCAD '07*, 2007.

[2] Roy Armoni, Sergey Egorov, Ranan Fraer, Dmitry Korchemny, and Moshe Y. Vardi. Efficient LTL compilation for sat-based model checking. In *International Conference on Computer-Aided Design (ICCAD'05)*, pages 877–884, 2005.

[3] Clark Barrett and Jacob Donham. Combining SAT methods with non-clausal decision heuristics. In *Proceedings of the CADE-20 Workshop: Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'04)*, July 2004. Cork, Ireland.

[4] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electr. Notes Theor. Comput. Sci.*, 66(2), 2002.

[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC*, pages 317–320, 1999.

[6] Per Bjesse and Koen Claessen. Sat-based verification without state space traversal. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Proceedings*, pages 372–389, 2000.

[7] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *Computer Aided Verification, 13th International Conference, CAV 2001, Proceeding*, pages 454–464, 2001.

[8] Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Computer Aided Verification, 11th International Conference, CAV '99, Proceedings*, pages 274–287, 1999.

[9] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[10] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[11] Kameshwar Chandrasekar and Michael S. Hsiao. State set management for sat-based unbounded model checking. In *ICCD*, pages 585–590. IEEE Computer Society, 2005.

[12] Pankaj Chauhan, Edmund M. Clarke, and Daniel Kroening. Using SAT based image computation for reachability analysis. Technical Report CMU-CS-03-151, Carnegie Mellon University, School of Computer Science, 2003.

[13] Pankaj Chauhan, Edmund M. Clarke, James H. Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002, Proceedings*, pages 33–51, 2002.

[14] Marsha Chechik and Wei Ding. Lightweight reasoning about program correctness. In *Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative Research, CASCON'01*, page 1, 2001.

[15] Yan Chen, Yujing He, Fei Xie, and Jin Yang. Automatic abstraction refinement for generalized symbolic trajectory evaluation. In *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Proceedings*, pages 111–118, 2007.

[16] Hana Chockler, Orna Grumberg, and Avi Yadgar. Efficient automatic ste refinement using responsibility. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008. Proceedings*, pages 233–248, 2008.

[17] Hana Chockler and Joseph Y. Halpern. Responsibility and blame: A structural-model approach. *J. Artif. Intell. Res. (JAIR)*, 22:93–115, 2004.

[18] Hana Chockler, Joseph Y. Halpern, and Orna Kupferman. What causes a system to satisfy a specification&quest;. *ACM Trans. Comput. Log.*, 9(3), 2008.

[19] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[20] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT press, December 1999.

[21] Edmund M. Clarke, Anubhav Gupta, James H. Kukula, and Ofer Strichman. Sat based abstraction-refinement using ilp and machine learning techniques. In *Computer Aided Verification, 14th International Conference, CAV 2002, Proceedings*, pages 265–279, 2002.

[22] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[23] Martin. Davis and Hilary. Putnam. A computing procedure for quantification theory. *JACM*, 7(3):201–215, July 1960.

[24] Hideo Fujiwara and Takeshi Shimono. On the acceleration of test generation algorithms. *IEEE Trans. Computers*, 32(12):1137–1144, 1983.

[25] Malay K. Ganai, Pranav Ashar, Aarti Gupta, Lintao Zhang, and Sharad Malik. Combining Strengths of Circuit-Based and CNF-Based Algorithms for a High-Performance SAT Solver. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 747–750, New York, NY, USA, 2002. ACM Press.

[26] Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient sat-based unbounded symbolic model checking using circuit cofactoring. In *ICCAD*, pages 510–517. IEEE Computer Society / ACM, 2004.

[27] Marcelo Glusman, Gila Kamhi, Sela Mador-Haim, Ranan Fraer, and Moshe Vardi. Multiplecounterexample guided iterative abstraction refinement: An industrial evaluation. In *TACAS'03*, 2003.

[28] Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. In *DATE '02*, page 142, Washington, DC, USA, 2002. IEEE Computer Society.

[29] Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. *Don't know* in the μ-calculus. In *VMCAI'05*, volume 3385 of *Lecture Notes in Computer Science*. Springer, 2005.

[30] Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. When not losing is better than winning: Abstraction and refinement for the full mu-calculus. *Inf. Comput.*, 205(8):1130–1148, 2007.

[31] Orna Grumberg, Assaf Schuster, and Avi Yadgar. Memory efficient all-solutions sat solver and its application for reachability analysis. In *5th Conference on Formal Methods in Computer Aided Design (FMCAD'04)*, pages 275–289, 2004.

[32] Orna Grumberg, Assaf Schuster, and Avi Yadgar. 3-valued circuit sat for ste with automatic refinement. In *ATVA'07*, pages 457–473, 2007.

[33] Aarti Gupta and Pranav Ashar. Integrating a boolean satisfiability checker and bdds for combinational equivalence checking. In *VLSID '98: Proceedings of the Eleventh International Conference on VLSI Design: VLSI for Signal Processing*, page 222, Washington, DC, USA, 1998. IEEE Computer Society.

[34] Aarti Gupta, Malay Ganai, Zijiang Yang, and Pranav Ashar. Iterative abstraction using sat-based bmc with proof analysis. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 416, Washington, DC, USA, 2003. IEEE Computer Society.

[35] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. Sat-based image computation with application in reachability analysis. In *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, 2000.

[36] Anubhav Gupta and Ofer Strichman. Abstraction refinement for bounded model checking. In *Computer Aided Verification, 17th International Conference, CAV 2005, Proceedings*, pages 112–124, 2005.

[37] Arie Gurfinkel and Marsha Chechik. Why waste a perfectly good abstraction? In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 2006.

[38] Josef Y. Halpern and Juda Pearl. Causes and explanations: A structural-model approach — part 1: Causes. In *Uncertainty in Artificial Intelligence: Proceedings of the Seventeenth Conference (UAI-2001)*, pages 194–202, San Francisco, CA, 2001. Morgan Kaufmann Publishers.

[39] Tamir Heyman, Daniel Geist, Orna Grumberg, and Aassaf Schuster. A scalable parallel algorithm for reachability analysis of very large circuits. *Formal Methods in System Design*, 21(3):317 – 338, November 2002.

[40] David Hume. *A treatise of human nature.* John Noon, London, 1739.

[41] Madhu K. Iyer, Ganapathy Parthasarathy, and Kwang-Ting Cheng. SATORI - A Fast Sequential SAT Engine for Circuits. In *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, page 320, Washington, DC, USA, 2003. IEEE Computer Society.

[42] HoonSang Jin, Mohammad Awedh, and Fabio Somenzi. CirCUs: A Satisfiability Solver Geared towards Bounded Model Checking. In *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 519–522. Springer, 2004.

[43] HoonSang Jin and Fabio Somenzi. Prime clauses for fast enumeration of satisfying assignments to boolean circuits. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 750–753, New York, NY, USA, 2005. ACM Press.

[44] Hyeong-Ju Kang and In-Cheol Park. Sat-based unbounded symbolic model checking. In *DAC*, 2003.

[45] Andreas Kuehlmann. Dynamic Transition Relation Simplification for Bounded Property Checking. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD'04)*, San Jose, California, November 2004.

[46] Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi. Circuit-based boolean reasoning. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 232–237, New York, NY, USA, 2001. ACM Press.

[47] Robert P. Kurshan. *Computer-Aided Verification of coordinating processes - the automata theoretic approach.* Princeton Univ. Press, 1994.

[48] Shuvendu K. Lahiri, Randal E. Bryant, and Byron Cook. A symbolic approach to predicate abstraction. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 141–153. Springer, 2003.

[49] Bin Li, Michael S. Hsiao, and Shuo Sheng. A novel sat all-solutions solver for efficient preimage computation. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10272, Washington, DC, USA, 2004. IEEE Computer Society.

[50] Bing Li and Fabio Somenzi. Efficient abstraction refinement in interpolation-based unbounded model checking. In *TACAS*, pages 227–241, 2006.

[51] Bing Li, Chao Wang, and Fabio Somenzi. Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. *STTT*, 7(2):143–155, 2005.

[52] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI (1)*, pages 366–371, 1997.

[53] Feng Lu, Madhu K. Iyer, Ganapathy Parthasarathy, Li-C. Wang, Kwang-Ting Cheng, and Kuang-Chien Chen. An efficient sequential sat solver with improved search strategies. In *DATE*, pages 1102–1107. IEEE Computer Society, 2005.

[54] Feng Lu, Li-C. Wang, Kwang-Ting Cheng, and Ric C-Y Huang. A Circuit SAT solver with signal correlation guided learning. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10892, Washington, DC, USA, 2003. IEEE Computer Society.

[55] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer, New York, 1995.

[56] Joao P. Marques-Silva and Kerem A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *IEEE International Conference on Tools with Artificial Intelligence*, 1996.

[57] Kenneth L. McMillan. Applying sat methods in unbounded symbolic model checking. In *Computer Aided Verification, 14th International Conference, CAV 2002, Proceedings*, pages 250–264, 2002.

[58] Kenneth L. McMillan. Interpolation and sat-based model checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Proceedings*, pages 1–13, 2003.

[59] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 530–535, New York, NY, USA, 2001. ACM.

[60] Alexander Nadel, Moran Gordon, Amit Palti, and Ziyad Hanna. Eureka-2006 sat solver. in SAT-Race 2006, Seattle, WA, USA, affiliated with SAT'06.

[61] Manish Pandey, Richard Raimi, Randal E. Bryant, and Magdy S. Abadir. Formal verification of content addressable memories using symbolic trajectory evaluation. In *DAC '97: Proceedings of the 34th annual Design Automation Conference*, pages 167–172, New York, NY, USA, 1997. ACM.

[62] Ganapathy Parthasarathy, Madhu K. Iyer, Kwang-Ting (Tim) Cheng, and Li-C. Wang. Safety Property Verification Using Sequential SAT and Bounded Model Checking. *IEEE Des. Test*, 21(2):132–143, 2004.

[63] David A. Plaisted. Method for design verification of hardware and non-hardware systems. United States Patents, 6,131, 078, October 2000.

[64] Stephen M. Plaza, Kai-hui Chang, Igor L. Markov, and Valeria Bertacco. Node mergers in the presence of don't cares. In *ASP-DAC '07: Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pages 414–419, Washington, DC, USA, 2007. IEEE Computer Society.

[65] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, FOCS*, pages 46–57, 1977.

[66] Jan-Willem Roorda. Symbolic trajectory evaluation using a satisfiability solver. Licentiate Thesis, 2005.

[67] Jan-Willem Roorda and Koen Claessen. A new sat-based algorithm for symbolic trajectory evaluation. In *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Proceedings*, pages 238–253, 2005.

[68] Jan-Willem Roorda and Koen Claessen. SAT-based assistance in abstraction refinement for Symbolic Trajectory Evaluation. In *Proc. of Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer Verlag, August 2006.

[69] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[70] Tom Schubert. High level formal verification of next-generation microprocessors. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 1–6, New York, NY, USA, 2003. ACM.

[71] Tobias Schuele and Klaus Schneider. Three-valued logic in bounded model checking. In *MEMOCODE '05: Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pages 177–186, Washington, DC, USA, 2005. IEEE Computer Society.

[72] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Form. Methods Syst. Des.*, 6(2):147–189, 1995.

[73] Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Thomas F. Melham, Mark Aagaard, Clark W. Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 24(9), 2005.

[74] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, London, UK, 2000. Springer-Verlag.

[75] Sharon Shoham and Orna Grumberg. A game-based framework for CTL counterexamples and 3-valued abstraction-refinemnet. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 275–287, Boulder, CO, USA, July 2003. Springer.

[76] Ofer Shtrichman. Tuning SAT checkers for bounded model checking. In *Computer Aided Verification*, pages 480–494, 2000.

[77] Fabio. Somenzi. CUDD: CU decision diagram package release 2.3.0, 1998.

[78] Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving non-clausal formulas with dpll search. In *Principles and Practice of Constraint Programming , 10th International Conference, CP 2004, Proceedings*, pages 663–678, 2004.

[79] Rachel Tzoref and Orna Grumberg. Automatic refinement and vacuity detection for symbolic trajectory evaluation. In *Computer Aided Verification, 18th International Conference, CAV 06, Proceedings*, pages 190–204, 2006.

[80] Moshe Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 322–331, Cambridge, 1986.

[81] Bruce Wile, Wolfgang Roesner, and John Goss. *Comprehensive Functional Verification: The Complete Industry Cycle.* Morgan-Kaufmann, 2005.

[82] James Christofer Wilson. *Symbolic Simulation Using Automatic Abstraction of Internal Node Values.* PhD thesis, Stanford University, Dept. of Electrical Engineering, 2001.

[83] Avi Yadgar, Orna Grumberg, and Assaf Schuster. Hybrid bdd and all-sat method for model checking. In *Languages: From Formal to Natural*, pages 228–244, 2009.

[84] Eran Yahav, Thomas Reps, and Mooly Sagiv. LTL model checking for systems with unbounded number of dynamically created threads and objects. Technical report, TR-1424, Computer Sciences Department, University of Wisconsin, Madison, WI, March 2001.

[85] Jin Yang, Rami Gil, and Eli Singerman. satGSTE: Combining the abstraction of GSTE with the capacity of a SAT solver. In *DCC*, 2004.

[86] Jin Yang and Amit Goel. Gste through a case study. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 534–541, New York, NY, USA, 2002. ACM.

[87] Hantao Zhang. SATO: An efficient propositional prover. In *Proc. of the 14th International Conference on Automated Deduction (CADE'97)*, 1997.

[88] Qi Zhu, Nathan Kitchen, Andreas Kuehlmann, and Alberto L. Sangiovanni-Vincentelli. Sat sweeping with local observability don't-cares. In *Proceedings of the 43rd Design Automation Conference, DAC 2006,*, pages 229–234, 2006.