Practical Design Verification

EDITED BY

Dhiraj K. Pradhan and Ian G. Harris

AMBRIDGE

TN47
P895

Practical Design Verification

Edited by

DHIRAJ K. PRADHAN

University of Bristol, UK

IAN G. HARRIS

University of California, Irvine

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo, Delhi

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521859721

First published 2009

Printed in the United Kingdom at the University Press, Cambridge

A catalog record for this publication is available from the British Library

Library of Congress Cataloging in Publication data
Practical design verification / Dhiraj Pradhan, Ian G. Harris.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-521-85972-1 (hardback)
1. Integrated circuits–Verification. I. Pradhan, Diraj. II. Harris, Ian G. III. Title.
TK7874.58.P73 2009
621.3815′48–dc22
2008050653

ISBN 978-0-521-85972-1 hardback

Practical Design Verification

Improve design efficiency and reduce costs with this practical guide to formal and simulation-based functional verification. Giving you a theoretical and practical understanding of the key issues involved, expert authors explain both formal techniques (model checking and equivalence checking) and simulation-based techniques (coverage metrics and test generation). You get insights into practical issues including hardware verification languages (HVLs) and system-level debugging. The foundations of formal and simulation-based techniques are covered too, as are more recent research advances including transaction-level modeling and assertion-based verification, plus the theoretical underpinnings of verification, including the use of decision diagrams and Boolean satisfiability (SAT).

Dhiraj K. Pradhan is Chair of Computer Science at the University of Bristol, UK. He previously held the COE Endowed Chair Professorship in Computer Science at Texas A & M University, also serving as Founder of the Laboratory of Computer Systems there. He has also worked as a Staff Engineer at IBM, and served as the Founding CEO of Reliable Computer Technology, Inc. A Fellow of ACM, the IEEE, and the Japan Society of Promotion of Science, Professor Pradhan is the recipient of a Humboldt Prize, Germany, and has numerous major technical publications spanning more than 30 years.

Ian G. Harris is Associate Professor in the Department of Computer Science, University of California, Irvine. He is an Executive Committee Member of the IEEE Design Automation Technical Commitee (DATC) and Chair of the DATC Embedded Systems Subcommittee, as well as Chair of the IEEE Test Technology Technical Committee (TTTC) and Publicity Chair of the IEEE TTTC Tutorials and Education Group. His research interests involve the testing and validation of hardware and software systems.

Contributors

Samar Abdi
Center for Embedded Computer Systems, University of California, Irvine, USA

Maciej Ciesielski
University of Massachusetts, Amherst, USA

Harry Foster
Mentor Graphics Corporation, USA

Masahiro Fujita
The University of Tokyo, Japan

Daniel Gajski
Center for Embedded Computer Systems, University of California,
Irvine, USA

Ian G. Harris
University of California, Irvine, USA

Abusaleh M. Jabir
Oxford Brookes University, UK

Joao Marques-Silva
University of Southampton, UK

Dhiraj K. Pradhan
University of Bristol, UK

Matteo Sonza Reorda
Politecnico di Torino, Italy

Ernesto Sánchez
Politecnico di Torino, Italy

Giovanni Squillero
Politecnico di Torino, Italy

Shireesh Verma
Conexant Systems, Inc., USA

Wayne H. Wolf
Georgia Institute of Technology, USA

Contents

List of contributors *page* x

1 **Model checking and equivalence checking** **1**
Masahiro Fujita

1.1 Introduction 1
1.2 Techniques for Boolean reasoning 2
1.2.1 Binary decision diagrams (BDDs) 3
1.2.2 Boolean satisfiability checker 6
1.2.3 Automatic test-pattern generation (ATPG) techniques 8
1.3 Model checking techniques 11
1.3.1 Property description with temporal logic 11
1.3.2 Basic algorithms of CTL model checking 14
1.3.3 Symbolic model checking 16
1.3.4 Practical model checking 20
1.4 Equivalence-checking techniques 22
1.4.1 Definition of equivalent designs 23
1.4.2 Latch-mapping problem 23
1.4.3 Practical combinational equivalence checking 24
1.4.4 Sequential equivalence checking (SEC) 28
1.5 Techniques for higher-level design descriptions 35
1.6 References 47

2 **Transaction-level system modeling** **51**
Daniel Gajski and Samar Abdi

2.1 Taxonomy for TLMs 51
2.1.1 Granularity-based classification of TLMs 52
2.1.2 Objective-based classification 60
2.2 Estimation-oriented TLMs 62
2.2.1 Result-oriented modeling (ROM) 63
2.2.2 Similarity to TLM 63
2.2.3 Optimistic modeling 64
2.2.4 Measurements 64

2.3 Synthesis-oriented TLMs 65
2.3.1 Universal bus channel (UBC) 67
2.3.2 Transducer 75
2.3.3 Routing 79
2.3.4 TLMs for C-based design 80
2.3.5 Synthesizable TLMs in practice: MP3 decoder design 83
2.4 Related work on TLMs 89
2.5 Summary and conclusions 90
2.6 References 90

3 Response checkers, monitors, and assertions 92
Harry Foster

3.1 Introduction 92
3.1.1 Identifying what to check 92
3.1.2 Classifying design behavior 93
3.1.3 Observability and controllability 96
3.2 Testbench verification components 97
3.3 Assertion-based verification 99
3.3.1 Brief introduction to SystemVerilog assertion 100
3.4 Assertion-based bus monitor example 102
3.4.1 Basic write operation 104
3.4.2 Basic read operation 105
3.4.3 Unpipelined parallel bus interface requirements 106
3.4.4 Unpipelined parallel bus interface coverage 108
3.4.5 Analysis communication in the testbench 110
3.5 Summary 111
3.6 References 112

4 System debugging strategies 113
Wayne H. Wolf

4.1 Introduction 113
4.2 Debugging tools 114
4.2.1 Logic analyzers and pattern generators 115
4.2.2 Power measurement 116
4.2.3 In-circuit emulators 117
4.2.4 Emulators 117
4.2.5 Profilers 117
4.2.6 CPU simulators 118
4.3 Debugging commands 118
4.4 Functional debugging 119
4.5 Performance-oriented debugging 119
4.6 Summary 120
4.7 References 121

5 **Test generation and coverage metrics** **122**
Ernesto Sánchez, Giovanni Squillero, and Matteo Sonza Reorda

5.1 Introduction 122
5.2 Coverage metrics 128
5.3 Classification of coverage metrics 131
5.3.1 Code coverage metrics 131
5.3.2 Metrics based on circuit activity 136
5.3.3 Metrics based on finite-state machines 137
5.3.4 Functional coverage metrics 140
5.3.5 Error- (or fault-) based coverage metrics 141
5.3.6 Coverage metrics based on observability 143
5.4 Coverage metrics and abstraction levels of design 144
5.5 Stimuli generation methods 145
5.5.1 Manual generation 146
5.5.2 Automatic generation 147
5.6 Acknowledgements 151
5.7 References 151

6 **SystemVerilog and Vera in a verification flow** **154**
Shireesh Verma and Ian G. Harris

6.1 Introduction 154
6.2 Testbench components 155
6.2.1 Design under verification 156
6.2.2 Monitor 156
6.2.3 Checker 157
6.2.4 Scoreboard 158
6.2.5 Stimulus 159
6.3 Verification plan 160
6.4 Case study 160
6.4.1 DUV 160
6.4.2 Verification plan 163
6.4.3 Testbench 163
6.5 Summary 171
6.6 References 172

7 **Decision diagrams for verification** **173**
Maciej Ciesielski, Dhiraj K. Pradhan, and Abusaleh M. Jabir

7.1 Introduction 173
7.2 Decision diagrams 175
7.2.1 Binary decision diagrams (BDDs) 175
7.2.2 Beyond BDDs 181
7.3 Binary moment diagrams (BMDs) 183

7.4 Taylor expansion diagrams (TEDs) 189
7.4.1 Related work 189
7.4.2 Motivation 190
7.4.3 The Taylor series expansion 191
7.4.4 Reduction and normalization 194
7.4.5 Canonicity of Taylor expansion diagrams 196
7.4.6 Complexity of Taylor expansion diagrams 197
7.4.7 Composition of Taylor expansion diagrams 199
7.4.8 Design modeling and verification with TEDs 202
7.4.9 Implementation and experimental results 204
7.4.10 Limitations of TED representation 209
7.4.11 Conclusions and open problems 212
7.5 Representation of multiple-output functions over finite fields 212
7.5.1 Previous work 213
7.5.2 Background and notation 214
7.5.3 Graph-based representation 217
7.5.4 Reduction 220
7.5.5 Variable reordering 223
7.5.6 Operations in GF(N) 225
7.5.7 Multiple-output functions in GF(N) 227
7.5.8 Further node reduction 228
7.5.9 Representing characteristic functions in GF(N) 229
7.5.10 Evaluation of functions 230
7.5.11 Experimental results 233
7.5.12 Conclusions 240
7.6 Acknowledgements 240
7.7 References 240

8 Boolean satisfiability and EDA applications 246
Joao Marques-Silva

8.1 Introduction 246
8.2 Definitions 247
8.2.1 Propositional formulas and satisfiability 247
8.2.2 Boolean circuits 249
8.2.3 Linear inequalities over Boolean variables 250
8.2.4 SAT algorithms 251
8.3 Extensions of SAT 256
8.4 Applications of SAT in EDA 257
8.4.1 Combinational equivalence checking 257
8.4.2 Automatic test-pattern generation 257

8.4.3 Design debugging 259
8.4.4 Bounded model checking 260
8.4.5 Unbounded model checking 262
8.4.6 Other applications 263
8.5 Conclusions 263
8.6 Acknowledgement 263
8.7 References 263

Index 269

1 Model checking and equivalence checking

Masahiro Fujita

1.1 Introduction

Owing to the advances in semiconductor technology, a large and complex system that has a wide variety of functionalities has been integrated on a single chip. It is called *system-on-a-chip* (SoC) or *system LSI*, since all of the components in an electronics system are built on a single chip. Designs of SoCs are highly complicated and require many manpower-consuming processes. As a result, it has become increasingly difficult to identify all the design bugs in such a large and complex system *before* the chips are fabricated. In current designs, the verification time to check whether or not a design is correct can take 80 percent or more of the overall design time. Therefore, the development of verification techniques in each level of abstraction is indispensable.

Logic simulation is a widely used technique for the verification of a design. It simulates the output values for given input patterns. However, because the quality of simulation results deeply depends on given input patterns, there is a possibility that there exist design bugs that cannot be identified during logic simulation. Because the number of required input patterns is exponentially increased when the size of a design is increased, it is clearly impossible to verify the overall design completely by logic simulation. To solve this problem, the development of formal verification techniques is essential. In formal verification, specification and design are translated into mathematical models. Formal verification techniques verify a design by proving its correctness with mathematical reasoning, and, therefore, they can verify the overall design exhaustively. Since formal verification is a mathematical reasoning process and logic circuits compute Boolean functions, it is realized on top of basic Boolean reasoning techniques, such as binary decision diagrams (BDDs), Boolean satisfiability checking methods (so-called SAT methods), and automatic test-pattern generation techniques (ATPG) for manufacturing test fields. The performance of formal verification methods relies heavily on the performance of these techniques. Figure 1.1 shows an overview of a formal verification flow. In formal verification, both specification and design descriptions are translated into mathematical models using front-end tools. Finite state machines, temporal logic, Boolean functions, and so on, are used as mathematical models. After mathematical models are obtained, they are analyzed

Practical Design Verification, eds. Dhiraj K. Pradhan and Ian G. Harris. Published by Cambridge University Press. © Cambridge University Press 2009.

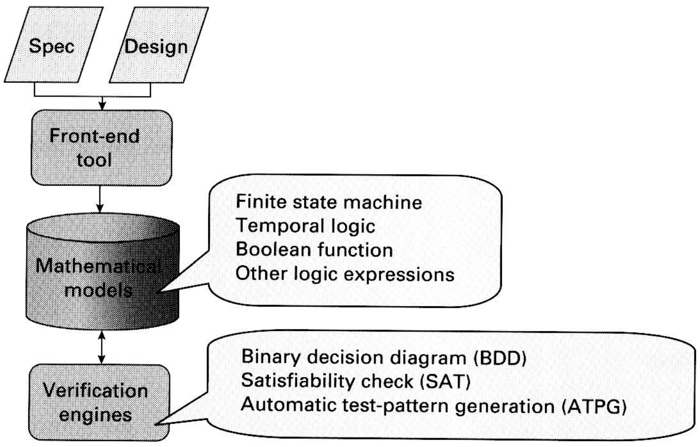

Figure 1.1 Formal verification of design descriptions

using BDD and SAT methods. Formal verification is equivalent to simulating all the cases in logic simulation. If there exists a design bug, formal verification techniques produce a counter-example to support debugging processes.

There are basically two problems in the verification of designs: model checking and equivalence checking. Model checking (or property checking) verifies whether a design satisfies the properties given as its specification. The performance of model checking has drastically improved in recent years, mainly owing to the significant progress of SAT-based efficient implementations. Equivalence checking verifies whether two given designs are equivalent or not. Equivalence checking can be applied to two designs in the same design level or in two different design levels. Depending on the types of equivalence definitions, equivalence checking can be made only on combinational parts of the circuits or on both combinational and sequential parts of the designs. In particular, the former type of equivalence checking has become very practical, and very large designs, such as those with more than 10 million gates, can be formally verified in a couple of hours.

In the actual design flow from highly abstracted design stages down to implementation levels, model checking is applied to each design level to ensure correct functionality, and equivalence checking is applied to any two different design levels so that correctness of the designs can be established. In this chapter, I first briefly review the Boolean reasoning techniques, BDD, SAT, and ATPG methods, in Section 1.2. Property checking and equivalence checking techniques are presented in Sections 1.3 and 1.4 respectively. In Section 1.5, formal verification techniques used in design levels higher than RTL are discussed.

1.2 Techniques for Boolean reasoning

In this section, I introduce three Boolean reasoning techniques, BDD, SAT, and ATPG techniques, which are the bases of formal verification methods. The performance of

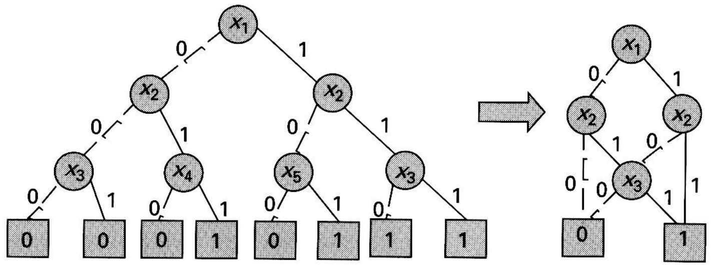

Figure 1.2 A binary decision tree representation of a Boolean function and its corresponding binary decision diagram (BDD)

formal verification methods fully relies on the performance of these techniques. In recent years SAT and ATPG methods, especially their program implementations, have been drastically improved, which make it feasible to verify real-life designs formally within reasonable time.

1.2.1 Binary decision diagrams (BDDs)

Reduced ordered binary decision diagrams (ROBDDs), simply called BDDs, are a canonical representation for Boolean functions. For many Boolean functions of practical interest in VLSI designs, BDDs provide a substantially more compact representation than other traditional alternatives, such as truth tables, sum-of-products (SOP) forms, or conjunctive normal form representations. Further, there exist efficient algorithms to manipulate BDDs. Thus, BDDs and their variants have become widely used in various areas of digital system design, including logic synthesis and formal verification of systems that can be represented in finite state machines. Binary decision diagrams represent the Boolean function as a directed acyclic graph. Let us first consider binary decision trees, an example of which appears on the left-hand side of Fig. 1.2, for the majority function, $f(x_1,x_2,x_3) = (x_1{}^\wedge x_2)^\vee(x_2{}^\wedge x_3)^\vee(x_1{}^\wedge x_3)$. The binary decision tree is a rooted directed tree with two kinds of node, terminal nodes and non-terminal nodes. Each non-terminal node v is labeled with a variable *var(v)* and has two successors, *hi(v)* and *lo(v)*, corresponding to the cases when *var(v)* is set to 1 and 0, respectively. The edge connecting v and *hi(v)*, shown as a solid line (*lo(v)* is shown as a dashed line), is labeled with 1 (0). Each terminal node (leaf node of the tree) is labeled by the Boolean value 0 or 1. Each truth assignment to the variables of the function has a one-to-one correspondence to a path in the tree from the root to a terminal node. This path can be traversed by starting with the root node and taking the edge corresponding to the truth value of the variable labeling the current node. The value labeling the terminal node is the value of the function under this truth assignment. This representation is, however, fairly redundant. For example, the sub-trees corresponding to the assignment $(x_1 = 0, x_2 = 1)$ and $(x_1 = 1, x_2 = 0)$ are isomorphic, and the vertex that corresponds to $(x_1 = 0, x_2 = 0)$ is redundant, since both assignments to x_3 at this point have the same consequence.

A BDD could be obtained for a given Boolean function by essentially placing two restrictions on its binary decision tree representation. The first restriction imposed is a total order $<$ on the variables labeling the vertices, such that for any vertex u in the diagram, if u has a non-terminal successor v, then $var(u) < var(v)$. The second set of restrictions involves merging isomorphic sub-trees and removing redundant vertices by repeatedly applying the following three reduction rules until no further application is possible.

1. *Remove duplicate terminals* Eliminate all but one terminal vertex with a given label and redirect all arcs going to the eliminated vertices into the remaining vertex.
2. *Remove duplicate non-terminals* If two non-terminal vertices u and v have $var(u) = var(v)$, $lo(u) = lo(v)$, and $hi(u) = hi(v)$, then eliminate one of u or v and redirect all incoming arcs to the eliminated vertex to the one that remains.
3. *Remove redundant tests* If a non-terminal vertex v has $hi(v) = lo(v)$, then eliminate v and redirect all its incoming arcs to $hi(v)$.

The resulting representation is a BDD. Figure 1.2 shows an example. The graph on the right-hand side is a BDD corresponding to the binary decision tree of the majority function, shown on the left-hand side in the figure.

Binary decision diagram representations are canonical – that is, two BDDs for a given Boolean function under a given variable ordering are isomorphic. [1] Because of this the equivalence of two Boolean functions can be simply checked by a graph isomorphism check on their respective BDD representations. A function is a tautology if and only if it is isomorphic to the trivial BDD corresponding to a single terminal 1 vertex and satisfiable if and only if it is not isomorphic to the trivial 0 BDD represented by a single 0 terminal vertex. A function is independent of a variable x if and only if there is no vertex labeled with x in its BDD.

The size of a BDD representation is critically dependent on its variable order. Figure 1.3 shows two different BDD representations for the comparator function. The one on the left side uses the ordering $a_1 < a_2 < b_1 < b_2$, while the one on the right uses the order $a_1 < b_1 < a_2 < b_2$. More generally, for an n-bit comparator, the ordering $a_1 <...<a_n < b_1 <...< b_n$ yields a BDD with $3 \cdot 2^n - 1$ vertices, while the ordering $a_1 < b_1 <...< a_n < b_n$ gives a BDD of size $3n + 2$. Thus, the size characteristics of the

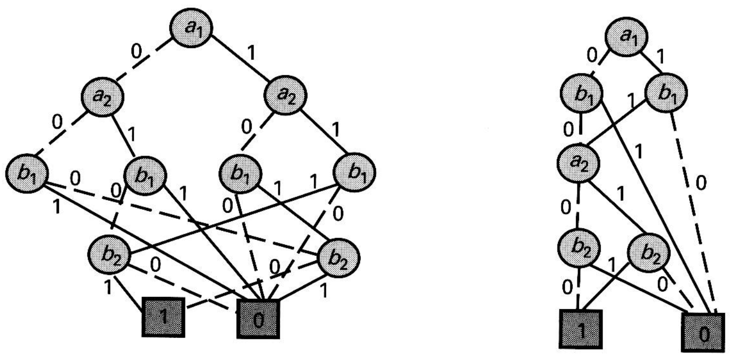

Figure 1.3 An example of how variable ordering can affect the size of an ROBDD

BDD can change from linear asymptotic growth to exponential asymptotic growth by altering the variable ordering strategy. In general, finding the optimal BDD variable order for a given function is a hard problem. Specifically, checking that a given variable order is optimal for a given function is an NP-complete problem. [2] Some classes of Boolean function are particularly difficult cases for BDDs, since any variable order results in a BDD with exponential complexity. The Boolean functions for the middle two outputs of an n-bit integer multiplier are one such example. [3]

The optimal variable order is, however, typically not necessary in order to effectively use BDDs. In practice, we need a variable order that keeps the BDD representations within reasonable limits so that suitable algorithms can manipulate them using the available computer power. In fact, many functions encountered in practical applications do have reasonably compact BDD representations. Moreover, efficient heuristics for BDD variable ordering have been developed that keep BDD sizes in check. One class of variable-ordering heuristics uses domain-specific knowledge to effect a good ordering. For example, if the Boolean function represents a logic gate network, then a depth-first traversal on the network graph can provide a good ordering. [4,5] Another technique, called *dynamic reordering* or *sifting*, [6] is an orthogonal approach, which is used when a domain-specific or constructive ordering algorithm is not available for the functions being manipulated. The technique simply performs a sequence of local reordering moves with the aim of reducing BDD size. It does this on a periodic basis to keep BDD sizes smaller and has often proved to be quite effective in practice.

One operation that is central to the construction, representation, and manipulation of BDDs is the *restriction* or *co-factoring* operation. A restriction or co-factor of f is the function that results when some variable x of f is set to a constant value k (0 or 1), denoted as $f_{x=k}$ or alternatively as f_x for $x = 1$ and $f_{\overline{x}}$ for $x = 0$. Given the two co-factors of a function, it can be expressed using the following identity, known as *Shannon's expansion*: $f = x \cdot f_x + \overline{x} \cdot f_{\overline{x}}$.

The manipulation of BDDs – that is, performing logical operations on functions represented as BDDs – is done using a single universal operation called the *ite (if-then-else)* operator (which internally makes use of the restriction operation). [7] The *ite* operator is a ternary operator, akin in functionality to a multiplexor (mux) in hardware or the *if-then-else* construct available in programming languages. It realizes the function expressed as $ite(f, g, h) = f \cdot g + \overline{f} \cdot h$, where f, g, and h are Boolean functions (possibly non-unique) represented as BDDs. In particular, *ite* can be used to implement any two-variable logic function, such as $f \oplus g = ite(f, \overline{g}, g)$ and $f \geq g = ite(f, 1, \overline{g})$.

Figure 1.4 shows the algorithm used to implement the *ite* operator for BDDs. It is a recursive algorithm where the leaves (terminal cases) of the recursion are degenerate cases of the *ite* operator for which precomputed and stored solutions are substituted, such as $ite(1,f,g) = ite(0,g,f)$ and f $ite(f,g,g) = g$. During the course of the algorithm, the BDD being generated may not remain fully reduced and canonical owing to the addition of new nodes, R. The *reduce*() function in the figure refers to the application of the reduction rules discussed earlier. In practical BDD packages, the need for this

```
ite(f,g,h) {
  if (terminal case) {
    return computed-result;
  } else { // general case
    let $\nu$ be the top variable of (f,g,h);
    $\tilde{f} \leftarrow$ ite($f_\nu, g_\nu, h_\nu$)
    $\tilde{f} \leftarrow$ ite($f_\nu, g_\nu, h_\nu$)
    R = new node labeled by $\nu$
    R.hi $\leftarrow \tilde{f}$
    R.low $\leftarrow \tilde{g}$
    reduce(R)
    return R;
```

Figure 1.4 Algorithm to implement the *ite* operator

reduce() operation is obviated by maintaining hash tables of both unique BDD nodes and previous *ite* calls. New *ite* calls, as well as new BDD nodes (R) created through them, are looked up against these hash tables before initiating new ones, thereby dynamically maintaining and growing a reduced-ordered BDD.

1.2.2 Boolean satisfiability checker

The Boolean satisfiability (SAT) problem is a well-known constraint satisfaction problem, with many applications in the fields of VLSI computer-aided designs and artificial intelligence fields. Given a propositional formula φ, the Boolean satisfiability problem posed on φ is to determine whether there exists a variable assignment under which φ evaluates to *true*. Such an assignment, if one exists, is called a *satisfying assignment* for φ, and φ is called *satisfiable*. Otherwise, φ is said to be *unsatisfiable*. The SAT problem is known to be NP-complete. [8] However, in recent years, there have been tremendous advancements in SAT technology, making SAT solvers a viable option for solving many real-world problems.

Most SAT solvers use a *conjunctive normal form (CNF)* representation of the propositional formula. A CNF formula consists of a conjunction of clauses, each of which is a disjunction of literals, and a literal is a variable or its negation. For example $(a + b + \bar{c})(\bar{a} + c)(a + \bar{b} + c)$ is a propositional formula in CNF over the variables a, b, and c. It is composed of a conjunction of three clauses. The clause $(a + \bar{b} + c)$ is one of the clauses, a disjunction of literals a, $\bar{b}$, and c. Note that for a CNF formula to be satisfied, each of its clauses must be satisfied – that is, evaluate to *true*. There exist polynomial algorithms to transform an arbitrary propositional formula into a satisfiability equivalent CNF formula, which is satisfiable if and only if the original formula is satisfiable.

Most modern SAT solvers are based on the *Davis–Putnam–Logemann–Loveland (DPLL) procedure*. [9,10] The DPLL algorithm essentially performs a