A General Equivalence Checking Framework for Multivalued Logic

Chia-Chun Lin, Hsin-Ping Yen, Sheng-Hsiu Wei, Pei-Pei Chen,
Yung-Chih Chen, and Chun-Yao Wang

ABSTRACT

Logic equivalence checking is a critical task in the ASIC design flow. Due to the rapid development in nanotechnology-based devices, an efficient implementation of multivalued logic becomes practical. As a result, many synthesis algorithms for ternary logic were proposed. In this paper, we bring out an equivalence checking framework based on multivalued logic exploiting the modern SAT solvers. Furthermore, a structural conflict-driven clause learning (SCDCL) technique is also proposed to accelerate the SAT solving process. The SCDCL algorithm deploys some strategies to cut off the search space for SAT algorithms. The experimental results show that the proposed SCDCL technique saves 42% CPU time from SAT solvers on average over a set of industrial benchmarks.

KEYWORDS

Equivalence checking, multivalued logic, and SAT solvers.

ACM Reference Format:
Chia-Chun Lin, Hsin-Ping Yen, Sheng-Hsiu Wei, Pei-Pei Chen, and Yung-Chih Chen, and Chun-Yao Wang. 2021. A General Equivalence Checking Framework for Multivalued Logic. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431588

1 INTRODUCTION

In early days, for multivalued logic, the functional completeness is a fundamental and important topic to researchers. Many previous works about multivalued logic in the 1950s focused on deriving a set of practically implementable primitive functions [10][23]. Later, some heuristics for minimizing multivalued logic were proposed [1][21]. In addition to the synthesis algorithms, the research of verifying multivalued logic circuits was also presented [6][17]. For example, the work of [6] proposed a method to decompose a multivalued logic function into the corresponding canonical representation, and facilitates the test vector generation for multivalued circuits. The work of [17] investigated different ternary decision diagrams and compared their sizes using a set of benchmark functions. However, due to the lack of efficient implementations, multivalued logic had little impact on integrated circuit designs.

In the past decade, multivalued logic has attracted great attention from the researchers again due to the rapid development in nanotechnology-based devices[11][18][19]. . For example, the threshold voltage of a carbon nanotube transistor (CNTFET) is determined by its diameter, and a multi-threshold design can be achieved by the diameter adjustment. Recently, the authors of [19] proposed a design methodology that aims to minimize the size of the ternary circuits. According to that work, the selection of condition function significantly influences the gate count in the synthesized ternary circuits. Moreover, several synthesis algorithms for ternary logic were proposed in [20][24]. For example, the work of [24] proposed a methodology to minimize the transistor count in the ternary circuits by a ternary-transformed binary decision diagram.

On the other hand, logic equivalence checking is a critical task in the ASIC design flow [7]. This is because the designers have to ensure the synthesized circuit matches the specification after transformation. Since logic equivalence checking is an NP-hard problem [7], various approaches had been proposed to accelerate the checking procedure. An important branch of approaches for equivalence checking is based on the binary decision diagram (BDD) and its derivatives [3][8][16]. Generally, in BDD-based approaches, the two circuits to be compared are transformed into the corresponding canonical forms such that they can be compared based on their structures. The major concern of the BDD-based equivalence checking is scalability. Once the number of inputs increases, the construction of BDD might be failed due to explosive memory requirement.

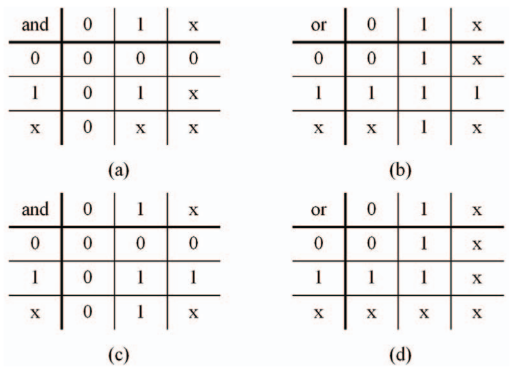
An effective solution to this scalability issue is to decompose the designs under verification into several subcircuits. The cutpoints for decomposition can be obtained by using random simulation, ATPG [4][5], or BDDs [9]. The authors in [9] adopted several methods, including circuit graph hashing, cut frontier, and false-negative elimination, to solve the memory blow-up issue. In contrast, the authors in [7][15] used SAT solvers, instead of the BDD-based approaches, to solve the equivalence checking problem. The work in [7] presented a detailed analysis on the features of SAT algorithms and claimed that SAT is a more robust and flexible engine of Boolean reasoning than BDDs for the equivalence checking application. Although the previous SAT-based works have achieved a significant success on equivalence checking, they did not deal with multivalued circuits.

Thus, in this work, we propose a general equivalence checking framework for multivalued logic. To the best of our knowledge,

This work is supported in part by the Ministry of Science and Technology of Taiwan under MOST 106-2221-E-007-111-MY3, MOST 108-2218-E-007-061, MOST 109-2221-E-007-082-MY2, MOST 109-2221-E-155-047-MY2, and MOST 109-2224-E-007-005.

C.-C. Lin, H.-P. Yen, S.-H. Wei, P.-P. Chen, and C.-Y. Wang are with the Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan 30013, R.O.C. Y.-C. Chen is with the Department of Computer Science and Engineering, Yuan Ze University, Taoyuan, Taiwan 32003, R.O.C.

ASPDAC '21, January 18–21, 2021, Tokyo, Japan
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431588



and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

(a)

or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

(b)

and	0	1	x
0	0	0	0
1	0	1	1
x	0	1	x

(c)

or	0	1	x
0	0	1	x
1	1	1	x
x	x	x	x

(d)

Figure 1: Truth table for ternary gates. (a) (b) The definition of ternary AND and OR gates in [26]. (c) (d) The definition of ternary AND and OR gates in [6][23].

$$(x_1 \vee x_2)(x_1 \vee x_3)(\overline{x_2} \vee \overline{x_3} \vee x_4)(\overline{x_4} \vee x_5 \vee x_6)(\overline{x_6} \vee \overline{x_7})(\overline{x_4} \vee x_7 \vee x_8)$$

(a)

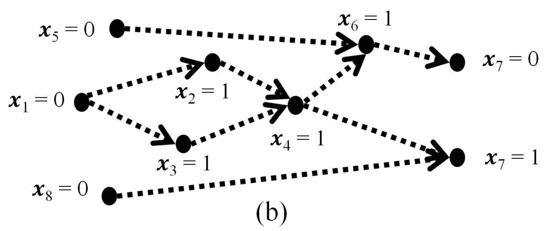


Figure 2: (a) A CNF formula to be solved. (b) A directed acyclic graph illustrates the variable assignment process in the SAT algorithm.

this is the first work that focuses on multivalued logic equivalence checking problem.

2 PRELIMINARIES

2.1 Ternary logic

In this paper, we select ternary logic $\{0, 1, x\}$ as an example to demonstrate the proposed multivalued logic equivalence checking framework. However, this framework can be applied to any multivalued logic by some minor adjustments. Fig. 1 shows the truth tables for primitive ternary gates. We observed that the same ternary gate has different functionalities in the literature [6][23][26]. For example, Fig. 1(a) and Fig. 1(b) show the definitions of ternary AND gate and OR gate described in the IEEE standard for Verilog hardware description language [26]. On the other hand, Fig. 1(c) and Fig. 1(d) show another behavior of ternary AND and OR gates [6][23]. The output of ternary AND gate under the input pattern $(1, x)$ is x in Fig. 1(a), but is 1 in Fig. 1(c). Although the multiple definitions of the ternary gates might cause some potential issues in other research, they are not troubles to the equivalence checking framework in this work. In the rest of this paper, we adopt the definition of ternary logic in the IEEE standard [26] to explain the proposed equivalence checking framework. If designers want to use another definition, they just make some changes accordingly.

2.2 Boolean satisfiability problem

Boolean satisfiability (SAT) problem is a problem of determining if there exists an assignment that satisfies the given Conjunctive Normal Form (CNF) formula. In the 1990s, Marques-Silva et al. [2][12][13][25] proposed a conflict-driven clause learning (CDCL) algorithm, which can improve the performance of modern SAT solvers [28][29]. Due to the rapid progress on the performance of SAT solvers, many EDA problems can be solved effectively and efficiently when they are modeled as an SAT problem.

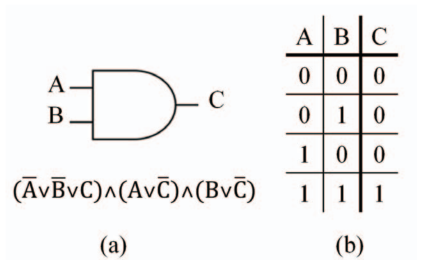
2.3 Unit propagation and the conflict-driven clause learning algorithm

Unit propagation plays an important role in the SAT problem because it prunes unnecessary search when traversing the solution space. Unit propagation means that the value of a variable can be exactly determined by logic implication. For example, Fig. 2(a) shows a CNF formula to be solved. The SAT algorithm first arbitrarily assigns an initial value to a variable and tries to find a satisfying assignment for this formula. In this example, assume that the SAT algorithm first assigns $x_1 = 0$. According to the given formula, we know that x_2 and x_3 have to be 1 simultaneously; otherwise the clauses $(x_1 \vee x_2)$ and $(x_1 \vee x_3)$ cannot be satisfied. Unit propagation can be applied on the CNF formula repeatedly if a clause only contains one variable to be assigned. The directed acyclic graph in Fig. 2(b) illustrates the unit propagation process of the CNF formula listed in Fig. 2(a). The edges in the graph stand for logic implications in the unit propagation. For example, $x_1 = 0$ implies $x_2 = 1$, hence, there exists an edge from $x_1 = 0$ to $x_2 = 1$ in the graph.

After assigning $x_1 = 0$, $x_2 = 1$, and $x_3 = 1$ to the formula, we observe that every unsatisfied clause contains more than one variable to be determined. Therefore, the SAT algorithm has to assign another variable. In this example, assume that x_5 is assigned to be 0 next. Then, we can obtain $x_6 = 1$ and $x_7 = 0$ by unit propagation again. However, we observe that x_7 has to be 1 when $x_4 = 1$ and $x_8 = 0$. Since x_7 would be assigned to be 1 and 0 simultaneously, a conflict occurs during this assignment process. Therefore, the assignments of these variables cannot satisfy the CNF formula. According to Fig. 2(b), we observe that the assignment $(x_4, x_6, x_8) = (1, 1, 0)$ will lead to the conflict without having assignments in other variables. That is, the SAT solving process cannot obtain a satisfying assignment with $(x_4, x_6, x_8) = (1, 1, 0)$. Thus, the CDCL algorithm adds an additional clause $\overline{(x_4 \wedge x_6 \wedge \overline{x_8})} = (\overline{x_4} \vee \overline{x_6} \vee x_8)$ to avoid obtaining the conflict assignment repeatedly. Another benefit of adding this additional clause is that this clause allows the SAT algorithm backtracking to the previous level in the decision diagram and thus cutting off a wide range of the search space.

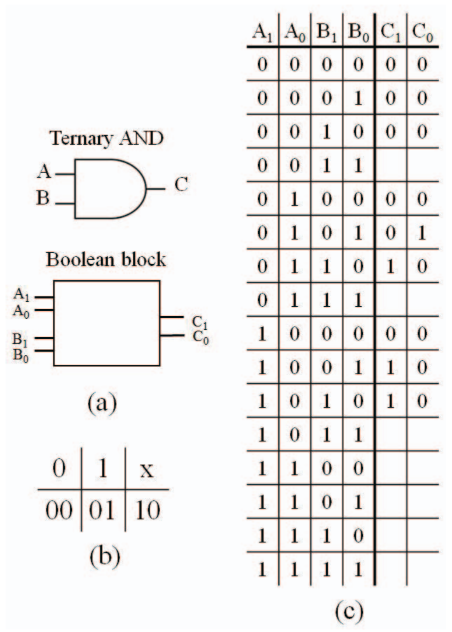
2.4 Tseitin transformation

Tseitin transformation algorithm [22] takes a combinational circuit as input and generates a corresponding CNF formula. Fig. 3(a) shows an AND gate and its corresponding CNF formula. Fig. 3(b) lists the truth table of AND function. We can see that only four assignments ABC = 000, 010, 100, and 111 satisfy the CNF formula.


A
B
C
$(\bar{A} \vee \bar{B} \vee C) \wedge (A \vee \bar{C}) \wedge (B \vee \bar{C})$
(a)
(b)

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Figure 3: (a) An AND gate and its corresponding CNF formula. (b) The truth table of AND function.



0	1	x
00	01	10

A_1	A_0	B_1	B_0	C_1	C_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1		
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1		
1	0	0	0	0	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1		
1	1	0	0		
1	1	0	1		
1	1	1	0		
1	1	1	1		

Figure 4: (a) A model of ternary AND gate and the corresponding Boolean block. (b) One encoding for ternary logic. (c) The truth table of the encoded block.

In other words, the satisfying assignments to the CNF formula are identical to the function of AND gate. In fact, all the logic gates can be expressed as their corresponding CNF formulae by Tseitin transformation.

3 PROPOSED EQUIVALENCE CHECKING FRAMEWORK

3.1 Encoding and optimization

Since solving the equivalence checking problem for Boolean circuits with SAT solvers has achieved a great success, we model a ternary logic circuit as a Boolean circuit for exploiting the strength of advanced SAT solvers. Intuitively, we can use two bits to represent the three values in the ternary logic. Therefore, the first step in the framework is to transform a ternary gate to a corresponding Boolean block, as shown in Fig. 4(a). Inputs A, B, and output C in a ternary AND gate are represented by wire pairs (A_1, A_0), (B_1, B_0), and (C_1, C_0) in the Boolean block.

Next, the original ternary value $\{0, 1, x\}$ are encoded to $\{00, 01, 10\}$ as shown in Fig. 4(b). To synthesize the transformed Boolean block, we construct its truth table as shown in Fig. 4(c) based on the definition of ternary AND gate in Fig. 1(a) and the encoding rules in Fig. 4(b). Since the truth table of the ternary AND gate contains $3^2 = 9$ entries only, when they are mapped to the truth table of the Boolean block, which contains $2^4 = 16$ entries, 7 entries in Fig. 4(c) remain unmapped. These unmapped entries can be viewed as don’t cares when synthesizing the circuit because they will never occur in the Boolean block. Since the input size of the Boolean block is small, we can use K-map or Quine-McCluskey minimization algorithm [14] to obtain the optimal result, as shown in Fig. 5(a). Fig. 5(b) shows the synthesized Boolean block, which contains four Boolean AND gates and one Boolean OR gate.

Obviously, different encoding rules influence the gate count of the synthesized Boolean block. For example, we can obtain another truth table when we encode the ternary value $\{0, 1, x\}$ as $\{01, 10, 00\}$ as shown in Fig. 6(a). Fig. 6(b) and Fig. 6(c) show the corresponding optimized result and the synthesized Boolean block, which contains only two gates, respectively. Since all the primitive ternary gates can be transformed into the corresponding Boolean blocks in the similar way, a ternary circuit can be transformed into a Boolean circuit while keeping its functionality. In this work, we have evaluated all the encoding rules and adopt the encoding rule in Fig. 6(a) for the Boolean block transformation such that the gate count in the transformed Boolean circuit is minimized.

3.2 Compatible equivalence checking

In the Boolean logic equivalence checking problem, the outputs of two circuits have to be exactly identical under every input pattern if and only if we claim that they are equivalent. The ternary logic, however, might have different definitions. The Problem A in the CAD Contest@ICCAD 2020 [30] gives the definition of compatible equivalence.

Definition 1: Given two variables $a, b \in \{0, 1, x\}$. Variable a is compatibly equivalent to b if and only if $(a, b) \in \{(0, 0), (1, 1), (x, 0), (x, 1), (x, x)\}$. On the contrary, variable a is not compatibly equivalent to b if and only if $(a, b) \in \{(0, 1), (1, 0), (0, x), (1, x)\}$.

That is, we can consider the variable a is from the specification while the variable b is from the implementation. Hence, it is allowed that when the specification requires the variable a to be x, the implementation returns the variable b to be 0, 1, or x. However, it is not allowed that when the specification requires the variable a to be either 0 or 1, the implementation returns the variable b to be x.

Given two circuits under equivalence checking G and R that are generated based on the encoding rule presented in Section 3.1. According to Definition 1, G is compatibly equivalent to R if and only if the output values of G are compatibly equivalent to the output values of R under every input pattern. We construct a subcircuit, as shown in the right part of Fig. 7, and connect it with the circuits G and R. Then, we use SAT solvers to check whether G is compatibly equivalent to R. Since G and R are transformed from ternary logic circuits, the outputs become two encoded Boolean pairs (G_1, G_0) and (R_1, R_0). The four gates AND_1, AND_2, AND_3, and AND_4 are designed to detect the non-compatible combinations $(G, R) = (0, 1)$, $(1, 0)$, $(0, x)$, and $(1, x)$, respectively. Note that the

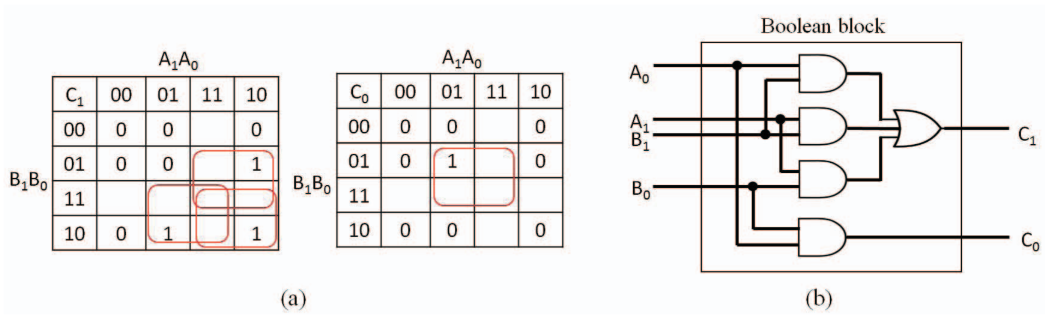


Figure 5: (a) The optimal result of the truth table in Fig. 4(c). (b) The synthesized Boolean block.

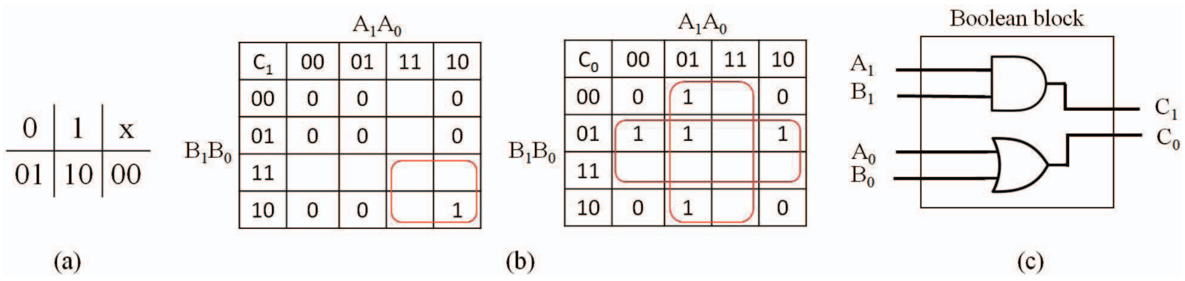


Figure 6: (a) Another encoding for ternary logic. (b) The truth table of the encoded block under the encoding rule in Fig. 6(a). (c) The synthesized Boolean block.

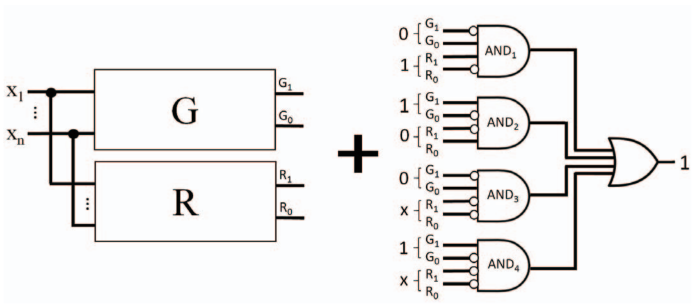


Figure 7: The proposed subcircuit for compatible equivalence checking.

output of circuit in Fig. 7 is set to be 1. This setting implies that at least one output of AND_1, AND_2, AND_3, and AND_4 has to be 1. When the SAT solver returns a *satisfiable* assignment, which is called a counterexample, for the corresponding CNF formula of the circuit, it means that there exists an input assignment such that G is not compatibly equivalent to R. If the SAT solver returns *unsatisfiable* for the corresponding CNF formula, G is compatibly equivalent to R. Note that the equivalence checking problem for the circuits with multiple outputs can be solved by duplicating the proposed subcircuit for each output.

3.3 Structural conflict-driven clause learning

Since the equivalence checking problem has been modeled as an SAT problem, it is important to deploy some strategies that can accelerate the SAT solving process. The CDCL algorithm [2][12][13] has been adopted by most modern SAT solvers because of its ability to learn new clauses from the conflicts to improve the performance in the search process.

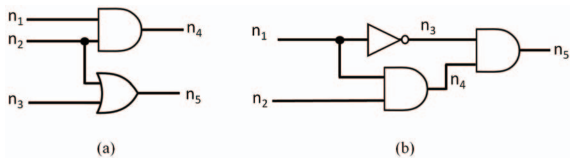


Figure 8: (a) An example for SCDCL. (b) An inconstant example for SCDCL.

Here, we propose a structural conflict-driven clause learning (SCDCL) algorithm to find the conflicts from the viewpoint of the circuit structure. That is, we find the wire assignment combinations that will never occur, satisfiability don't cares, in the circuit by logic implication. Fig. 8(a) shows an example to demonstrate SCDCL algorithm. First, we set $n_4 = 1$, and this setting implies $n1 = 1$ and $n2 = 1$. Then, we know that $n_5 = 1$ because $n2 = 1$ is the controlling value for the OR gate. Since $n_4 = 1$ implies $n_5 = 1$, this assignment (n_4, n_5)=$(1, 0)$ is not possible in the circuit. Thus, we can add an additional clause $\overline{(n_4 \wedge \overline{n_5})} = (\overline{n_4} \vee n_5)$ into the CNF formula for pruning the search space for SAT solver. To maximize the ability of SCDCL algorithm, we iteratively assign the output of AND gate to be 1 and the output of OR gate to be 0. Next, we perform logic implications on the circuit as much as possible. Finally, we add the learned clauses from the structural conflicts into the CNF formula.

In the logic implication process, however, we might encounter the situation that we have to assign inconsistent values to a wire. For example, assume that Fig. 8(b) is a subcircuit embedded in a large circuit, $n_5 = 1$ implies $n_3 = 1$ and $n_4 = 1$; $n_3 = 1$ implies

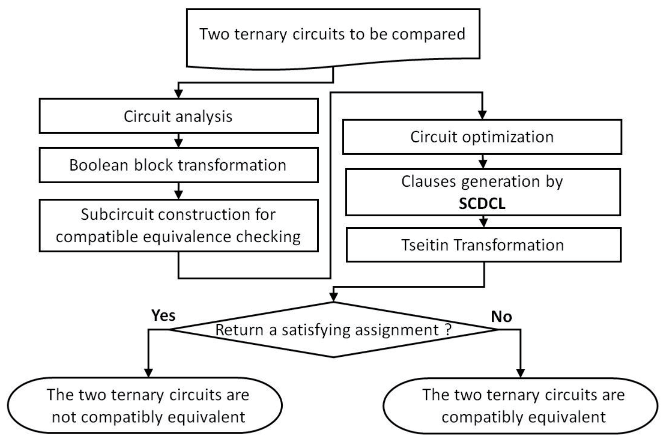

Two ternary circuits to be compared
Circuit analysis
Boolean block transformation
Subcircuit construction for compatible equivalence checking
Circuit optimization
Clauses generation by SCDCL
Tseitin Transformation
Yes
Return a satisfying assignment ?
No
The two ternary circuits are not compatibly equivalent
The two ternary circuits are compatibly equivalent

Figure 9: The overall flowchart of the proposed equivalence checking framework.

$n_1 = 0$; and $n_4 = 1$ implies $n_1 = 1$ and $n_2 = 1$. Thus, both $n_1 = 0$ and $n_1 = 1$ are required in the implication process. This inconsistent situation means that n_5 cannot be 1 in the given circuit. Therefore, n_5 can be replaced with a constant zero and the transitive fanin cone of n_5 can be totally removed for circuit minimization when it drives n_5 only.

Adding these additional clauses can accelerate the SAT solving process because it prunes the solution space. In addition to the conflicts obtained by logic implication, we can also add other clauses from the encoding rule to improve the performance. According to Fig. 6(a), the ternary values $\{0, 1, x\}$ are encoded as $\{01, 10, 00\}$ in this work, respectively. In other words, the value of 11 will not occur in the encoded wire pair. Therefore, we can also exploit this information to add the corresponding clauses into the CNF formula. For example, we will add three clauses $(\overline{A_1 \wedge A_0}) = (\overline{A_1} \vee \overline{A_0})$, $(\overline{B_1 \wedge B_0}) = (\overline{B_1} \vee \overline{B_0})$, and $(\overline{C_1 \wedge C_0}) = (\overline{C_1} \vee \overline{C_0})$ into the CNF formula for the transformed Boolean block in Fig. 4(a).

4 OVERALL FLOW

The proposed general equivalence checking framework is shown in Fig. 9. The inputs are two ternary logic circuits under verification. First, we conduct circuit analysis on the benchmarks. The benchmarks used in the experiments are provided by the contributor of Problem A in the CAD Contest@ICCAD 2020 [30], which contains large industrial designs. According to the problem description, the ternary values are generated from some ternary sources. In other words, the ternary values might not pass through the whole circuit. Therefore, the framework first recognizes the part of circuit that has to be transformed into the corresponding Boolean block. Second, the Boolean blocks are generated by the selected encoding rule. Third, the subcircuit for compatible equivalence checking is constructed. Since the ternary circuits are transformed into the corresponding Boolean circuits, we can further optimize circuits with existing optimization algorithms. Next, we perform the proposed SCDCL algorithm to add additional clauses for reducing the SAT solving time. After that, the CNF formulae of the two circuits under verification are generated by Tseitin transformation. Finally, the two ternary circuits are reported as not compatibly equivalent if the SAT solver returns a satisfying assignment; otherwise, the two ternary circuits are compatibly equivalent.

5 EXPERIMENTAL RESULTS

We implemented the proposed equivalence checking framework in C++ language. The experiments were conducted on a 2.6 GHz Linux platform (CentOS 6.7). The benchmarks used in the experiments are provided by the contributor of Problem A in the CAD Contest@ICCAD 2020 and are available online [30]. Note that we will apply the existing logic optimization tool, ABC [27] and a rewriting script *resyn2*, in the circuit optimization stages for exploiting logic sharing among the two circuits under verification.

Table 1 shows the experimental results. Column 1~3 list the benchmarks information. Columns 4 and 5 show the gate counts of the original ternary circuit and the corresponding optimized Boolean circuit, respectively. In general, the gate count in Boolean circuits is larger than that in the ternary circuits. However, it is not the case for some benchmarks due to logic optimization. Columns 6~7 show the required CPU time for identifying whether the given two circuits are compatibly equivalent without or with applying the proposed SCDCL algorithm. Column 8 shows the reduction percentage of required CPU time. The last column is the result obtained by the equivalence checking framework. For example, consider the benchmark case 5, the gate count of the original ternary circuit is 11183 and that of the transformed Boolean circuit is 16562. The required CPU time without or with applying the SCDCL algorithm is 6788.05 or 1649.38 seconds, respectively. The reduction percentage of the required CPU time is 76%, and result is equivalent (EQ). According to Table 1, the proposed SCDCL algorithm saves the SAT solving time by 42% on average for all the solved benchmarks. Note that the framework cannot finish cases 3, 8, 10, 15, and 17 in 12 hours for both approaches.

6 CONCLUSION

In this work, we propose a general multivalued logic equivalence checking framework and demonstrate it using ternary logic. This framework can be applied to other multivalued logic by selecting a proper encoding rule. Furthermore, an SCDCL algorithm for reducing the SAT solving time is also presented. The experimental results show that a 42% CPU time reduction can be obtained by applying the SCDCL algorithm.

REFERENCES

[1] C. M. Allen and D. D. Givone, "A minimization technique for multiple-valued logic systems," *IEEE Trans. on Computers*, vol. C-17, no. 2, pp.182-184, 1968.
[2] R. J. Bayardo Jr and R. C. Schrag, "Using CSP look-back techniques to solve real world SAT instances," in *Proc. AAAI*, pp. 203-208, 1997.
[3] C. L. Berman and L. H. Trevillyan, "Functional comparison of logic designs for VLSI circuits," *Proc. ICCAD*, pp. 456-459, 1989.
[4] A. Biere and W. Kunz, "SAT and ATPG: Boolean engines for formal hardware verification," *Proc. ICCAD*, pp. 782-785, 2002.
[5] M.K. Ganai, A. Aziz, and A. Kuehlmann, "Enhancing simulation with BDDs and ATPG," in *Proc. DAC*, pp. 385-390, 1999.

Table 1: CPU time comparison for ternary equivalence checking between w/o and w/ SCDCL.

	|PI|	|PO|	|T. gate|	|B. gate|	w/o SCDCL (s)	w/ SCDCL (s)	Reduction (%)	Result
case 1	48	20	1653	3077	1096.65	963.32	12	EQ
case 2	48	20	1679	3097	1.94	0.29	85	NEQ
case 3	64	32	11649	15354	>12hrs	>12hrs	-	-
case 4	64	32	11944	15649	5.22	3.70	29	NEQ
case 5	160	32	11183	16562	6788.05	1649.38	76	EQ
case 6	160	32	11183	16562	7352.98	1810.77	75	EQ
case 7	96	59	14843	42692	6.87	3.70	46	NEQ
case 8	8214	93	137051	169131	>12hrs	>12hrs	-	-
case 9	96	58	31415	96248	422.10	5.24	99	NEQ
case 10	256	119	90473	88957	>12hrs	>12hrs	-	-
case 11	256	85	88711	86975	34.41	29.93	13	NEQ
case 12	614	54	40537	50447	25789.68	20004.78	22	EQ
case 13	614	68	38645	67887	10896.54	6620.4	39	EQ
case 14	128	1	46958	45446	34.11	33.01	3	NEQ
case 15	193	120	97727	98471	>12hrs	>12hrs	-	-
case 16	1903	1382	102739	90587	159.18	101.66	36	EQ
case 17	128	115	56887	56274	>12hrs	>12 hrs	-	-
case 18	217	2	6110	3489	0.4	0.34	15	EQ
Avg.	-	-	-	-	-	-	42	-

[6] T. A. Giuma, M.A. Tapia, “Canonical representation of multi-valued logic functions,” in *Proc. IEEE Southeastcon*, 1992

[7] E. I. Goldberg, M. R. Prasad and R. K. Brayton, “Using SAT for combinational equivalence checking," in *Proc. DATE*, 2001.

[8] J. Jain, R. Mukherjee, and M. Fujita, “Advanced verification technique based on learning,” in *Proc. DAC*, pp. 420-426, 1995.

[9] A. Kuehlmann and F. Krohm, “Equivalence checking using cuts and heaps," in *Proc. DAC*, pp. 263-268, 1997.

[10] C. Y. Lee and W. H. Chen, “Several-valued combinational switching circuits," *Trans. of the American Institute of Electrical Engineers*, vol. 75, no. 3, pp. 278-283, 1956.

[11] S. Lin, Y.-B. Kim, and F. Lombardi, “A novel CNTFET-based ternary logic gate design," in *Proc. International Midwest Symposium on Circuits and Systems*, 2009

[12] J. P. Marques-Silva and K. A. Sakallah, “GRASP-A new search algorithm for satisfiability," in *Proc. ICCAD*, pp. 220-227, 1996.

[13] J. P. Marques-Silva and K. A. Sakallah, “GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. on Computers*, pp. 506-521, 1999.

[14] E. J. McCluskey, “Minimization of Boolean functions," *Bell Syst. tech J.*, vol. 35, no. 5, pp. 1417-1444, Nov. 1956.

[15] A. Mishchenko, R. Brayton, J.-H. Jiang, and S. Jang, “Scalable don't-care-based logic optimization and resynthesis,” in *International Symposium on FPGA*, pp. 151-160, 2009

[16] S. M. Reddy, W. Kunz, and D. K. Pradhan, “Novel verification framework combining structural and OBDD methods in a synthesis environment,” in *Proc. DAC*, pp. 414-419, 1995.

[17] T. Sasao, “Ternary decision diagrams,” in *Proc. International Symposium on Multiple-Valued Logic*, 1997.

[18] N. Soliman, M. E. Fouda, and A. G. Radwan, “Memristor-CNTFET based ternary logic gates,” *Microelectronics Journal*, vol. 72, pp. 74–85, 2018.

[19] N. Soliman, M. E. Fouda, A. G. Alhurbi, L. A. Said, A. H. Madian, and A. G. Radwan, “Ternary functions design using memristive threshold logic,” *IEEE Access*, vol. 7, 2019.

[20] B. Srinivasu and K. Sridharan, “A synthesis methodology for ternary logic circuits in emerging device technologies,” *IEEE Trans. on Circuits and Systems*, vol. 64, no. 8, pp. 2146-2159, 2017.

[21] S. Y. H. Su, and P. T. Cheung, “Computer minimization of multivalued switching functions,” *IEEE Trans. on Computers*, vol. C-21, no. 9, pp. 995-1003, 1972.

[22] G. Tseitin, “On the complexity of derivation in propositional calculus," Studies in constructive mathematics and mathematical logic, vol. 2, no. 115-125, pp. 10–13, 1968.

[23] Z. G. Vranesic, E. S. Lee, and K. C. Smith, “A many-valued algebra for switching systems,” *IEEE Trans. on Computers*, vol. C-19, no. 10, pp. 964-971, 1970.

[24] C. Vudadha, A. Surya, S. Agrawal and M. B. Srinivas, “Synthesis of ternary logic circuits using 2:1 multiplexers”, *IEEE Trans. on Circuits and Systems*, vol. 65, no. 12, pp. 4313-4325, 2018.

[25] L. Zhang, C. F. Madigan, M. H. Moskewicz and S. Malik, “Efficient Conflict Driven Learning in a Boolean Satisfiability Solver,” *Proc. ICCAD*, 2001.

[26] IEEE Standard Verilog Hardware Description Language, in *IEEE Standard 1364-2005*.

[27] Berkeley Logic Synthesis and Verification Group, “ABC: a system for sequential synthesis and verification,” Available: https://people.eecs.berkeley.edu/~alanmi/abc/.

[28] https://www.labri.fr/perso/lsimon/glucose/.

[29] https://github.com/arminbiere/cadical/.

[30] http://iccad-contest.org/2020/problems.html