Hindawi
Complexity
Volume 2021, Article ID 5525539, 12 pages
https://doi.org/10.1155/2021/5525539

Research Article

CoCEC: An Automatic Combinational Circuit Equivalence Checker Based on the Interactive Theorem Prover

Wilayat Khan,[1] **Farrukh Aslam Khan**,[2] **Abdelouahid Derhab**,[2] **and Adi Alhudhaif**[3]

[1]*Department of Electrical and Computer Engineering, COMSATS University Islamabad, Wah Campus, Islamabad, Pakistan*
[2]*Center of Excellence in Information Assurance (CoEIA), King Saud University, Riyadh 11653, Saudi Arabia*
[3]*Department of Computer Science, College of Computer Engineering and Sciences, Prince Sattam Bin Abdulaziz University, Al-kharj, Saudi Arabia*

Correspondence should be addressed to Farrukh Aslam Khan; fakhan@ksu.edu.sa

Received 21 January 2021; Accepted 4 May 2021; Published 26 May 2021

Academic Editor: Huihua Chen

Checking the equivalence of two Boolean functions, or combinational circuits modeled as Boolean functions, is often desired when reliable and correct hardware components are required. The most common approaches to equivalence checking are based on simulation and model checking, which are constrained due to the popular memory and state explosion problems. Furthermore, such tools are often not user-friendly, thereby making it tedious to check the equivalence of large formulas or circuits. An alternative is to use mathematical tools, called interactive theorem provers, to prove the equivalence of two circuits; however, this requires human effort and expertise to write multiple output functions and carry out interactive proof of their equivalence. In this paper, we (1) define two simple, one formal and the other informal, gate-level hardware description languages, (2) design and develop a formal automatic combinational circuit equivalence checker (CoCEC) tool, and (3) test and evaluate our tool. The tool CoCEC is based on human-assisted theorem prover Coq, yet it checks the equivalence of circuit descriptions purely automatically through a human-friendly user interface. It either returns a machine-readable proof (term) of circuits' equivalence or a counterexample of their inequality. The interface enables users to enter or load two circuit descriptions written in an easy and natural style. It automatically proves, in few seconds, the equivalence of circuits with as many as 45 variables (3.5×10^{13} states). CoCEC has a mathematical foundation, and it is reliable, quick, and easy to use. The tool is intended to be used by digital logic circuit designers, logicians, students, and faculty during the digital logic design course.

1. Introduction

In case of electronic system design or manipulation of mathematical functions, a system is often represented and transformed into different forms. During the design process of digital logic circuits, a circuit may be represented in the form of a Boolean function and is simplified using different methods, such as mathematical manipulation and Karnaugh map method. As the simplified function is the representation of the logic circuit with fewer number of logic components, it results in a compact, economical, and efficient hardware product. However, the correctness of the behaviour of the simplified circuit must be proved to ensure that the manipulation process does not alter the intended behaviour of the circuit. Such a functional verification of a system is often referred to as equivalence checking [1, 2].

With the increase in complexity and criticality of software and hardware systems, the need of sophisticated, reliable, and formal tools for the analysis of such systems is also increasing [3, 4]. A common approach to check the equivalence of two circuits or functions is model checking [5–7]. Model checker may get into a loop due to the state explosion problem [7], while checking large models with thousands of states is not user friendly [8, 9], making them tedious and error prone. A different form of the formal approach is to use interactive theorem provers (ITPs), such as Coq [10], Isabelle/HOL [11], and ACL2 [12]. This form of formal approach for checking

correctness of digital circuits [13] is described in Figure 1. Formal models of the circuit under test and the security or reliability properties of interest as theorems are given as the input into a theorem prover (such as the Coq system), and a mathematical proof that the model of the circuit satisfies the properties is carried out interactively.

Interactive theorem provers resolve issues such as state explosion using a termination checker that filters out nonterminating functions. However, proof tactics, such as destruct in Coq, can raise memory explosion issue, if used improperly. A Coq proof script in the form destruct v_1; destruct v_2; destruct $v_3 \ldots$.destruct v_n; auto., where v_i is a Boolean variable, will create 2^n subgoals at one time and would explode memory for larger n. Furthermore, interactive theorem proving is more powerful than model checking [14], but it requires human assistance to prove theorems.

There exist well-established and popular formal tools to reason about combinational circuits and Boolean functions; however, either they are not automatic (Coq, Isabelle/HOL, etc.), face state explosion issues (model checkers), and are difficult or unable to generate the counterexample (Coq, ACL2, etc.) or require expertise to encode logical formulas (Z3, BirdBrain II, etc.) (see Table 1). Boolean satisfiability solvers (SATs) can examine the conjunctive normal form (CNF) formula and check if the formula is consistent or not. SATs are useful tools, but they are not as powerful and expressive as proof assistants based on higher-order logic (such as Isabelle/HOL) and calculus of constructions (such as Coq). Another type of formal tools is automated theorem provers that support automated reasoning in inductive logical theories. One such tool is ACL2 [12], which is a logic and programming language used to model and reason about computer systems. ACL2 is fully automated and more scalable; however, it requires expertise and skills to write or encode logical formulas in its syntax. Using and-inverter graph (AIG) to represent Boolean functions, for example, the function *aigandx*$_1$ x_2, implements the conjunction, and it takes two variables as arguments.

In this paper, we propose and develop a tool called CoCEC that combines the strengths of multiple approaches (automatically proving using the ITP tool) for checking the equivalence of combinational circuits. The tool CoCEC, developed in C++, checks combinational circuits' equivalence, returns proof of their equivalence, or generates a counterexample. CoCEC accepts circuits in a user-friendly style and automatically checks their equivalence using the interactive theorem proving approach without the user guidance. It creates a Coq proof script and proof object pair for each equivalence checking test, if circuits are equivalent; otherwise, it returns a counterexample. We define two description languages: the first one is used to describe circuits in an easy and natural style, while the other is used to describe circuits in the higher-order logic of Coq to enable formal reasoning. The major contributions of the paper include the development and integration of the following components:

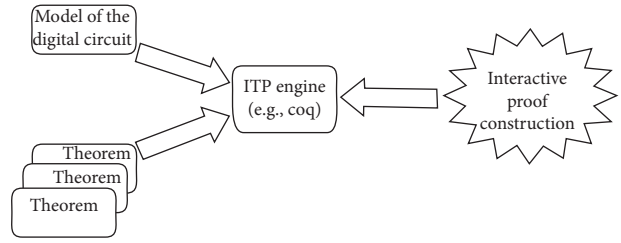

Figure 1: Formal verification of digital circuits.

(i) Two gate-level description languages are defined. The first one is aimed to describe circuits in an easy and natural style, while the other is intended to enable formal reasoning.

(ii) A translator is designed and developed to automatically translate circuit descriptions in the first language to equivalent descriptions in the second language.

(iii) A proof checker is developed that formally checks the equivalence of two circuit descriptions. It either generates the proof object (term) of their equality or generates a counterexample of their inequality.

(iv) Extensive experiments are carried out to check the robustness and performance of the tool.

The source code of our tool (for Windows and Ubuntu platforms) is available online at https://github.com/wilstef/cocec.

The remainder of the paper is organized as follows: in the next section (Section 2), a short introduction to Boolean algebra and Coq proof assistant is given. The syntax of two description languages is discussed in Section 3. The major components of the CoCEC tool are described in Section 4. The tool is tested in Section 5, and its performance is evaluated in Section 6. A brief survey of the literature is given in Section 7. Section 8 concludes the paper with future research directions.

2. Motivation and Background

The tool CoCEC checks functional equivalence of two combinational circuits taken as the input, translates them into a formal representation in Coq, and carries proof of their equivalence in the Coq theorem prover. In this section, the motivation behind CoCEC is discussed by comparing its simplicity and ease of use with few other similar tools. The two description languages used in CoCEC are based on Boolean algebra; hence, the basics of Boolean algebra and the Coq interactive theorem prover are necessary to understand the operation of our tool.

2.1. Motivation. CoCEC accepts two gate-level descriptions in the first language and generates equivalent codes in the second language. Furthermore, CoCEC automatically checks their equality (or inequality). In addition to the inherited benefits of interactive theorem proving, CoCEC tool is simple, easy to use, and compatible with many contemporary tools, while at the same time, it solves equivalence problems in the order of few seconds. A glimpse

Table 1: Input comparison of CoCEC with other formal tools.

Tool	Input
Z3	(declare-const a Bool) ... (declare-const g Bool) (assert (= (and a (and b (and c (and d (and e (and f g)))))) (and a (and b (and c (and d (and e (and f g)))))))) (check-sat)
BirdBrain II	*hypothesis* ... a∧(b∧(c∧(d∧(e∧(f∧g))))) = a∧(b∧(c∧(d∧(e∧(f∧g))))).
ACL2	and a b c d e f g = and a b c d e f g
CoCEC	abcdefg = abcdefg

of easiness of CoCEC's usability is shown in Table 1; encoding logic functions for checking their equivalence is easy in CoCEC as compared to tools Z3 [8], BirdBrain II [9], and ACL2 [12]. Furthermore, the input syntax of CoCEC directly accepts the Boolean expressions used in popular text books on digital logic design [15]. CoCEC is built on top of the interactive theorem prover Coq, though it automatically proves the equivalence of two functions without the human assistance.

2.2. Boolean Algebra. Boolean algebra, as defined by Boole [16], is an algebra that includes a set of values, two binary operations + and · over the values in the set, and six Huntington [17] postulates with their proofs. Shannon [18] defined a two-valued version of the algebra to model and describe properties of switching circuits. In Shannon's algebra, the two values are 1 and 0, and it includes rules for the two binary operations together with a unary complement operation, as shown in Table 2.

2.3. Coq Proof Assistant. Boolean functions in the Shannon's two-valued Boolean algebra may be used to model logic circuits, as shown in Figure 2. The function $F = \overline{x}.\overline{y}$ models the circuit where the AND gate represents the product operation (Table 2) and the NOT gate represents the complement operation (Table 2). When logic circuits are represented with functions as in Figure 2, they can be manipulated, and different properties can be stated and proved using tools and techniques based on Boolean algebra.

To formally reason about computer systems, the system under test and the intended properties are formalized in the logic of a theorem prover (such as Coq [10] and Isabelle/HOL [11]). The proof language of the proof assistant is used to create a proof that the (model of the) system satisfies the property. The proof checker of the proof assistant is used to check if the proof is valid. To describe formal systems and proofs using the proof assistant, a system of numbers is defined and formally reasoned about using the proof assistant Coq. In this formal system, the numbers are inductively defined as a data type nat using the Coq keyword Inductive with two constructors to construct elements of this type (lines 1–3, Figure 3). The definition of type nat states that O (for 0) is nat, and if n is nat, then Sn is also nat. The term S (S (S O))), for example, is a number (4) in nat.

After specifying the numbers, functions can be defined to manipulate them. The recursive function add (lines 5–9) on the numbers gets two numbers n and m and returns their

Table 2: Truth tables for sum (a), product (b), and not (c) operations.

(a) Sum

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1

(b) Product

x	y	$x.y$
0	0	0
0	1	0
1	0	0
1	1	1

(c) Not

x	x'
0	1
1	0

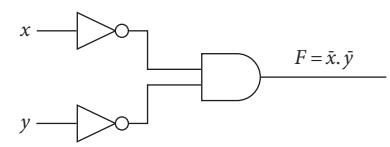

Figure 2: Implementation of Boolean function $F = \overline{x}.\overline{y}$ with gates.

```
1  Inductive nat: Type :=
2    | O : nat
3    | S : nat → nat.
4
5  Fixpoint add (n m: nat) : nat :=
6    match n with
7    | O ⇒ m
8    | Sn′ ⇒ S (add n′ m)
9    end.
10
11 Lemma add_n_o: ∀ n, add n O = n.
12 Proof.
13     induction n.
14         reflexivity.
15         simpl. rewrite IHn.auto.
16 Qed.
```

Figure 3: Interactive formal proof in Coq.

sum. It returns the second argument m if the first argument is O, and it returns S (add n'm) if the first argument is of the form S n'. A lemma add_n_o that add n O = n holds for any value of n is stated and proved in Figure 3 (lines 11–16). This lemma is proved by induction on the construction of the first argument n. During the proof process, a proof expert assists the Coq tool interactively by providing it commands called tactics (lines 12–16).

3. Description Languages

In order to describe combinational circuits in an easy and natural style, we define a simple input description language to represent circuits at the gate level. To enable formal reasoning in the interactive proof environment, the logic circuits must be described using the Coq notations. For this purpose, a simple formal description language is defined and embedded in the Coq proof assistant to represent logic circuits.

3.1. Input Description Language. The tool CoCEC is aimed to provide an easy-to-use user interface to facilitate describing logic circuits. To represent combinational circuits in a user-familiar style, a simple description language is defined. This language supports and is motivated by the descriptions in the text book [15] on digital logic and computer design and tools [19, 20]. Its syntax is defined by the grammar production listed in Figure 4.

The grammar production rules state that all English letters (upper and lower case) and digits (0 and 1) are logic circuits. All (non-)parenthesized circuits and their sum, product, and complement are also circuits. Finally, a sequence of circuits (separated by ":") are also (multioutput) combinational circuits. A combinational circuit, full adder, may be defined in this language as the following: $x'y'z + xx'yz' + xy'z' + xyz$: $xy + xz + yz$. This style of circuit representation is used to enter circuit descriptions directly into the user interface of CoCEC.

To describe circuits in a conventional style as modules (.txt files), a more structured syntax is given in Figure 5. A circuit definition begins with keywords begin module and ends with end module. Each output function is defined as a Boolean function that begins with keyword Output. As many other programming languages, single and multiline comments are allowed, and every statement (output function) terminates with a semicolon. The order of output definitions is important: function for the most significant bit is defined first, while the least significant bit is defined last. The module in Figure 5 describes the full-adder circuit with two outputs for sum and carry bits.

3.2. Output Description Language. The input description language does not have a formal foundation and hence is not fit for formal reasoning. To enable formal reasoning using ITP, a formal description language is defined in the logic of Coq. The Boolean algebra defined previously [21] is redefined and tailored towards combinational circuit definitions. The first part, a set of two elements, is defined in Coq as a

C	::=	circuit
	| {A-Za-z}	upper and lower case letters
	| {0,1}	digit 0 and 1
	| C + C	sum
	| CC	product
	| C′	complement
	| (C)	parenthesis
	| C : C	sequence

Figure 4: Grammar of the input description language.

```
1 //Comment: description of full-adder
2 begin module
3   Output x'y'z + x'yz' + xy'z' +xyz; //Sum
4   Output xy + xz + yz; //Carry
5 end module
```

Figure 5: Example description in the input language.

data type bool using the keyword Inductive, as given in Figure 6 (line 1). The two members of this type, true and false, represent the two values of the Boolean algebra.

Next is to define binary operations over the two values of type bool. The first operation + (sum) is defined as a Coq function sum on lines 3–7 in Figure 6. The function sum gets two arguments (two values of type bool), and it returns the value false if both input arguments are false and returns true, otherwise. Similarly, the second binary operation · (product) is defined by the function prod (lines 9–13, Figure 6). If the two input values of this function are true, it returns true; otherwise, it returns false for all other input values. Furthermore, a unary operation complement is defined by the function not (lines 15–19). It inverts the input value (returns true given false and vice versa). If the values true and false are replaced with digits 1 and 0, respectively, these functions would behave exactly as the truth tables (Table 2) in Section 2.2. Finally, a combinational circuit is described as a list of Boolean values (line 21), where the bit at the head is the most significant bit, while the last bit in the tail is the least significant bit.

The constructors and function (operations) names are cumbersome to write, in particular, when combinational circuit descriptions are long. Luckily, Coq provides shorthand notations to replace English names with notations. The three operations sum, prod, and not are represented with infix notations +, $*$, and a prefix notation ¬, respectively (Figure 7), with ¬ highest and + lowest precedence. The Coq term sum$x$(prod$x$(not$y$)), for example, is represented with $x + x*$, $\neg y$, where $\neg y$ is evaluated first, followed by the product, and then the sum operation is performed. Similarly, a combinational circuit is defined as a list of semicolon-separated Boolean functions enclosed in brackets. The formal notations just presented can be used to describe combinational circuits in Coq. The full-adder combinational circuit in Figure 5, for example, is encoded in the Coq notations, as shown in Figure 8. The sum corresponds to the

```
1  Inductive bool: type := true | false.
2
3  Definition sum (m n: bool) : bool :=
4      match m, n with
5    | false, false ⇒ false
6    | _, _⇒ true
7      end.
8
9  Definition prod (m n: bool): bool :=
10     match m, n with
11   | true, true ⇒ true
12   | _, _⇒ false
13     end.
14
15 Definition not (m : bool): bool :=
16     match m with
17   | false ⇒ true
18   | true ⇒ false
19     end.
20
21 Definition circuit := list bool.
```

Figure 6: Formal definition of the combinational circuit [22].

```
1  Notation "m + n":= (sum m n)
2                     (at level 50, left associativity) :bool_scope.
3  Notation "m ∗ n" := (prod m n)
4                     (at level 40, left associativity): bool_scope.
5  Notation" ¬m":= (not m) (at level 30, right associativity): bool_scope.
6  Notation" [m ; .. ; n]":= (cons m .. (cons n nil) ..).
```

Figure 7: Shorthand notations.

function definition at the head, while the tail represents carry.

4. CoCEC Architecture

The formal tool CoCEC accepts two combinational circuit descriptions as .txt files defined in the input description language. Alternatively, two circuit descriptions can also be entered in the two text boxes in the main window. The front end of CoCEC checks (for syntax errors) and passes the input to other components of the tool. The components of CoCEC all together translate the inputs to circuit representations in the output language, generate the Coq script, and formally verify whether or not the two circuits are equivalent. It either generates a proof object if circuits are equivalent or a counterexample, otherwise. A diagram showing the schematic view of the tool CoCEC is shown in Figure 9, and the detail of its components is given in the following sections. All the components of CoCEC have formal semantics that make it a rigorous and reliable tool.

4.1. User Interface. The CoCEC tool has a simple user interface (Figure 10). It consists of two text boxes for describing two circuits. Alternatively, the user can load existing descriptions as .txt files into the relevant text boxes by clicking on the load buttons. The button *Check Equivalence* triggers the main code of the tool, which runs in the background to manipulate the text entered (or loaded) into the boxes. The tool is closed by clicking the button *Close* or clicking the cross × in the upper right corner of the window.

When any of the circuit descriptions entered/loaded is ill-formed and the user clicks the button *Check Equivalence*, the tool displays a meaningful error message in a message box, and the cursor is moved to the erroneous text box. If both descriptions are well formed, they are given as the input to the next component.

4.2. Syntax Checker. The syntax checker (checker in Figure 11 and the library syntaxchecker.cpp in the source code) is a lexer and parser for the text entered by the user. It first preprocesses the descriptions (removes comments and keywords) and then creates circuit descriptions in the form $f_1 : f_2 : f_3 : \ldots : f_n$, where f_i represent individual Boolean functions defined following the grammar in Figure 4. Next, it ensures that the syntax of the input circuit descriptions, as defined by the grammar production in Figure 4, is correct. In other words, the syntax checker implements the rules defined in Figure 11. The checker filters out ill-formed texts and passes on the well-formed circuit descriptions from the text boxes.

The typing rules for well-formed logic circuits, as defined in Section 2.2, are given in Figure 11. Lower and upper case English letters (for identifiers) and binary digits 1 and 0 (logic values) are circuits according to the rules T-ID and T-VALUE, respectively. Complement of a term is also a valid term (rule T-COMPL). Sum + and product (default operator) of two valid circuits are also valid circuits (rules T-SUM and T-PROD). All parenthesized circuits are also circuits (rule T-PAREN), and the sequence of circuits is also a circuit. These rules have been enforced by the function check_syntax_circuit in library syntaxchecker.cpp. The checker, for example, accepts $xx + xxy' : x, xx + xy'$ and $xx + (xy')$ but does not accept $x - (xy')$, $xx + (xy, xx + xxy' :, x.y, xx + +xy$, and so on.

4.3. Intermediate Representation Generator. The intermediate representation (IR) generator is a mini-compiler that translates the circuit description in the input language to an equivalent description in the output language. A logic circuit description following the rules given in Figure 11 is well formed (valid); however, one cannot mechanically reason about it as it does not conform to a syntax acceptable by a theorem prover. This representation is aimed to provide an easy-to-use interface to humans by allowing them to enter

```
1 Definition full_adder (x y z: bool) : circuit:=
2   [¬x ∗ ¬y ∗ z + ¬x ∗ y ∗ ¬z + x ∗ y ∗ ¬z + x ∗ y ∗ z;x ∗ y + x ∗ z + y ∗ z].
```

Figure 8: Example description in the output language.

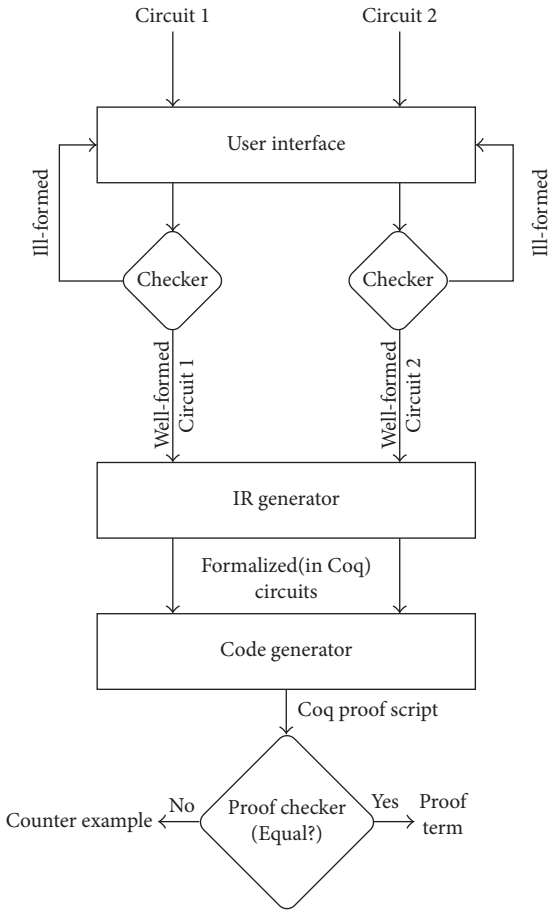

Figure 9: Schematic diagram of the CoCEC tool.

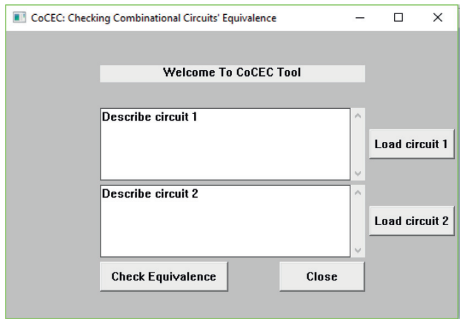

Figure 10: User interface of CoCEC.

logic circuit descriptions in a more natural way. On the contrary, circuits defined in the output description language given in Section 3 are different from the ones user is allowed to enter (in the input language). To fill this gap, a translator (irgenerator.cpp file) is written to translate the input from the user to an equivalent intermediate Coq representation of the functions in the formalized version of the input language. The translation appears direct, but it requires identifying the tokens including identifiers, operators, and separator parenthesis. In particular, translating postfix complement ' to the prefix ¬ notation and inserting the $\star$ operator between any two terms with no infix operator notation are tricky to deal with.

The inference rules in Figure 12 define the acceptable circuit description in the formalized output language. These rules describe the Coq definitions in Section 3 and make the comparison between the two circuit representations easy (note the highlighted text in Figure 12). The logic values 0 and 1 are replaced with logic values false and true, respectively. As an example to describe the above rules, the circuit $xy' + 1$ is translated to a Coq definition $x *$, $\neg y +$ true (equivalent to the term sum (prod x(not y)) true).

4.4. Code Generator. The translator creates circuit descriptions that conform to the formal output language syntax defined in Figure 6 (Section 3). To check the equivalence of any two circuit descriptions given as the output by the IR generator, a complete proof script needs to be developed. Coq theorem prover is an interactive tool, and hence, an interactive proof script can be created; however, it requires formal verification skills and is tedious to carry out the formal proof of equivalence for every couple of functions. This translation is automated by the code generator module (codegenerator.cpp file).

In the two-valued algebra, there are only two logic values, and every identifier can be assigned one of these values. In such a formal proof, the case analysis proof method is ideal: check the equivalence of functions by evaluating them for every identifier taking one of the two values. The Coq script generator module of CoCEC automates this process. The generator generates the Coq proof script and stores it in proofscript.v file (for the input to the Coq proof checker). The file proofscript.v contains code for importing a Coq library containing the formal definitions, theorem statement (or set of theorem statements for the sequence circuit), and proof(s) of the theorem(s). The theorem(s) state(s) that the two circuit descriptions (received from the IR generator) are equivalent for any value of the identifiers in the descriptions. The proof script is a proof that the two circuit descriptions are indeed functionally equivalent. An example of the equivalence checking proof of the consensus theorem is shown in Figure 13.

The proof script generated for every equivalence check may be different in two places, as highlighted in Figure 13. The list of identifiers (line 12) and body of the theorem statement (line 13) change for every test. The left and right parts of = in the body are description 1 and description 2, respectively, received from the upper module (IR generator). The list of identifiers on lines 12 and 15 is gathered from the bodies of both descriptions separated by comma ",". The proof commands on lines 15–19 close the proof by repeating

$$\frac{id \in \{\text{A-Za-z}\}}{\vdash id : C}\ \text{T-ID} \qquad \frac{x \in \{1, 0\}}{\vdash x : C}\ \text{T-VALUE} \qquad \frac{\vdash x : C}{\vdash x' : C}\ \text{T-COMPL}$$

$$\frac{\vdash x_1 : C \quad \vdash x_2 : C}{\vdash x_1 + x_2 : C}\ \text{T-SUM} \qquad \frac{\vdash x_1 : C \quad \vdash x_2 : C}{\vdash x_1 x_2 : C}\ \text{T-PROD}$$

$$\frac{\vdash x : C}{\vdash (x) : C}\ \text{T-PAREN} \qquad \frac{\vdash c_1 : C \quad \vdash c_2 : C}{\vdash c_1 : c_2}\ \text{T-CIRCUIT}$$

Figure 11: Rules for checking logic circuits in the input description language.

$$\frac{id \in \{\text{A-Za-z}\}}{\vdash id : C}\ \text{TF-ID} \qquad \frac{x \in \{\text{true, false}\}}{\vdash x : C}\ \text{TF-VALUE} \qquad \frac{\vdash x : C}{\vdash \neg x : C}\ \text{TF-COMPL}$$

$$\frac{\vdash x_1 : C \quad \vdash x_2 : C}{\vdash x_1 + x_2 : C}\ \text{TF-SUM} \qquad \frac{\vdash x_1 : C \quad \vdash x_2 : C}{\vdash x_1 * x_2 : C}\ \text{TF-PROD}$$

$$\frac{\vdash x : C}{\vdash (x) : C}\ \text{TF-PAREN} \qquad \frac{\vdash c_1 : C \quad \vdash c_2 : C}{\vdash [c_1; c_2]}\ \text{TF-CIRCUIT}$$

Figure 12: Rules for checking logic circuits in the output description language.

```
1  Input by the user (interface)
2     circuit 1: xy + x'z + yz
3     circuit 2: xy + x'z
4  Output of the checker (well formed):
5     circuit 1: xy + x'z + yz
6     circuit 2: xy + x'z
7  Output of the IR generator:
8     circuit 1 : x * y + ¬x * z + y * z
9     circuit 2 : x * y + ¬x * z
10 Output of the code generator:
11    Require Import bal gebra.
12    Theorem boolean_equivalence_0 : forall x y z : bool,
13     x * y + ¬x * z + y * z = x * y + ¬x * z.
14    Proof.
15     intros; un fold sum; un fold prod.
16     repeat (
17        try reflexivity;
18        try Destruct Match Arg
19     ); auto.
20     Show.
21     Qed.
22     Print boolean_equivalence_0.
23     Print Assumptions boolean_equivalence_0.
24 Output of the proof checker:
25    Circuits are equivalent!
```

Figure 13: CoCEC equivalence checking example.

a Coq tactic DestructMatchArg. The tactic DestructMatchArg recursively uses pattern matching over Boolean operators prod and sum and then solves goals by case analysis over the variables. It consequently simplifies the subgoals and closes them by applying Coq's reflexivity tactic. The Show, Qed, and Print tactics are used for debugging and error reporting and are discussed in more detail in Section 5. The Show and Print tactics are used for debugging only and are removed from the final proof script.

4.5. Proof Checker. Once the proof script, following the Coq syntax, is generated, its correctness can be checked mechanically by the Coq proof checker. The proof checker module in Figure 9 (file proofchecker.cpp) automates this process by using the *system* command in C++. The checker runs the Coq tool and gives the script received from the previous component as the input to the Coq proof checker. The last command in the proof script (line 21, Figure 13) saves the proof in the Coq repository if closed; otherwise, it prints the error message *Error: Attempt to save an incomplete proof*. Based on the result of the Coq proof checker, the CoCEC proof checker displays to the user whether the circuits are

(i) Equivalent: creates and adds the proof object to file proofobject.txt, or

(ii) Not equivalent: creates and adds a counterexample in the counterexample.txt file, or

(iii) Returns a syntax/type error caused by incorrect translation (this state is reached only when no or incompatible (other than Coq 8.8) version is installed)

The modules in Figure 9 discussed so far translate a valid circuit description (module) in one format (input language) to a valid circuit description in another format (Coq script). Such a tool is more commonly referred to as compiler or interpreter [23]. The translator in the CoCEC tool is designed to translate from one mathematical representation to another and is not part of a standalone programming language and is directly (manually) implemented without using standard tools and techniques. Regardless of the approach used to develop such a translator, it is immensely important to ensure the correctness of the tool; it must be verified that CoCEC generates the correct code (see Section 5).

4.6. Counterexample Generation. After the correct Coq proof script is created, the CoCEC tool creates a proof object (term) using the tactic Show if descriptions are equal; otherwise, it generates a counterexample. The proof object is generated automatically from the Coq proof script using the tactic Show; however, a translator is designed and written to

create a counterexample. The tactic Show adds the first pending goal to proofobject.txt if the earlier commands (lines 15–19, Figure 13) are unable to close it. The first goal encountered that cannot be closed is taken, and a counterexample is generated from it. The variables opened up with logic values (true or false), which resulted in unequal left-right sides, are replaced with corresponding values in the original body of functions. As an example, a counterexample for the test $x + x \star y! = y$ is true + true⋆false! = false, which should be interpreted as $\exists x = \text{true}$, $\exists y = \text{false}$, $x + x \star y! = y$.

5. Testing

To check the robustness of the tool CoCEC, the correctness of the results must be verified. As the target language is Coq with a type checker, all the translated formalized descriptions are type-checked by the Coq compiler. The Coq compiler accepts only well-formed codes, thereby ensuring that only correct translations are processed by the CoCEC tool. If the tool suggests that the descriptions are equal or unequal, then it must be guaranteed that they indeed are equal or unequal. In the former case, the CoCEC tool gives the Coq proof object and proof script, which the user can confirm using the Coq checker, while in the latter case, a counterexample is returned. In case of error, CoCEC not only returns a detailed error message but also the erroneous Coq script, which again can be checked using the Coq checker.

The Coq tactic repeat on lines 16–19 (in Figure 13) repeats tactics on lines 17 and 18 until the proof is closed. The Coq commands Show, Qed, and Print are used to update the content of four files, which together are used to determine one of the three possibilities, as shown in Table 3. The three possibilities (right-most three columns in Table 3) are (1) the code generated is incorrect (e.g., syntax or type errors have been introduced), (2) functions are equivalent and the proof closes, or (3) functions are not equivalent, and repeat returns without closing the proof. The contents of these files (left-most column, Table 3) are used for the final decision to be displayed to the user and creating a counterexample. The command Show adds to file proofobject.txt the current pending goal (or does nothing if the proof is already closed by preceding tactics). Tactic Qed adds the proof script to the Coq repository for later referral if everything is fine; otherwise, it adds an error message to file coqerror.txt. The command Print boolean_equivalence_0 adds the Coq proof term to file proofobject.txt (this is executed only if the Coq proof is closed by the earlier tactics). The tactic Print Assumptions boolean_equivalence_0 prints pending assumptions/lemmas, if any, not yet proved.

The tool is extensively tested in four sets of experiments (see the next section) where each set includes experiments checking equivalence of 45 pairs of functions with binary variables 1 to 45. Furthermore, it is tested against numerous examples taken from the digital logic design text book [15] and used by our undergraduate students during their digital logic design course.

Table 3: Content of files after CoCEC execution.

File	Incorrect translation	Equal	Unequal
coqerror.txt	Error detail	Empty	Empty
proofscript.v	Incorrect script	Proof script	Empty
proofobject.txt	Empty	Proof object	Empty
counterexample.txt	Empty	Empty	Counterexample

6. Performance Evaluation

The main strength of the CoCEC tool is the ability to automatically prove the equivalence of combinational circuits using an interactive proof assistant. The CoCEC tool, for example, can be used by the students studying digital logic design course to formally verify the equivalence of a given Boolean function against its simplified function. Regardless of the simplification algorithm used, whether it is the Karnaugh map, the tabular method, or the logical manipulation, the functional equivalence of the original and the simplified version can be checked. Automatically proving equivalence using the interactive proof assistant inherits the benefits of interactive proof assistants, while at the same time, it has the automatic feature of automated provers. Automated provers, such as model-checking tools, which deal with Boolean logic, have drawbacks: they have to run through all the state spaces to determine whether the assertion is true for every case. Theorem proving, with the human support, on the contrary, can prove systems using induction, which otherwise, would be intractable [14]. CoCEC tool is designed to automatically solve equivalence checking problems with the proof script generated in interactive proof assistant Coq. As it proves theorems without human assistance, a performance degradation is expected if Coq tactics are used inefficiently.

Using the language of tactic in Coq [24], a special tactic is defined that applies tactic destruct when a particular pattern in the goal is found. The tactic DestructMatchArg (DestructMatchArg tactic was suggested by Michael Soegtrop on coq-club on May 08, 2018), slightly edited from the original version, pattern matches over the goal and applies tactic destruct over terms with binary operator + or ⋆, followed by reflexivity and auto. A simple application of the destruct tactic, without proper treatment as mentioned above, over every variable creates too many goals to keep in memory at the same time and often results in memory explosion; destructing 45 variables in functions would create 2^{45} subgoals at one time. In an experiment, the system took more than 14 seconds just for 11 variables, while it did not return for more than 20 variables (memory explosion). A careful and controlled repetitive application of the DestructMatchArg inside repeat closes the goal one by one efficiently, without opening and checking all the cases at one time.

Using the divide-and-conquer rule, DestructMatchArg replaces one variable with its two logic values and simplifies and checks if the equivalence can be solved; otherwise, it repeats the case analysis for other variables until the goal is

closed or all cases are seen one by one. This improves the performance, and it can prove the equivalence of circuits with more than 45 variables (more than 3.5×10^{13} possible cases) in just one and a half second. The performance of CoCEC was checked against circuits with up to 45 variables on our Windows machine with Intel Core i5, 2.60 GHz microprocessor for four different sets of equation pairs (Figure 14). Each time, the experiments were performed for 45 variable descriptions. CoCEC checked the equality of syntactically similar descriptions in just a fraction of a second (black line), while it took less than two seconds to check the commutative property (red line). In the third set of experiments, the tests were performed on random (the circuit description pairs were manually created, and randomness was introduced by writing one expression in each pair in the reverse order and adding a syntactic sugar to make the descriptions syntactically different) descriptions, and it took less than 14 seconds to check the equality of circuits up to 45 input variables. Finally, similar experiments were performed for unequal circuit description pairs. The inequalities were proved in about 7 seconds for as many as 45-variable unequal pairs of descriptions. These tests suggest that the CoCEC tool can be used effectively to check the equality of complicated descriptions up to 45 variables.

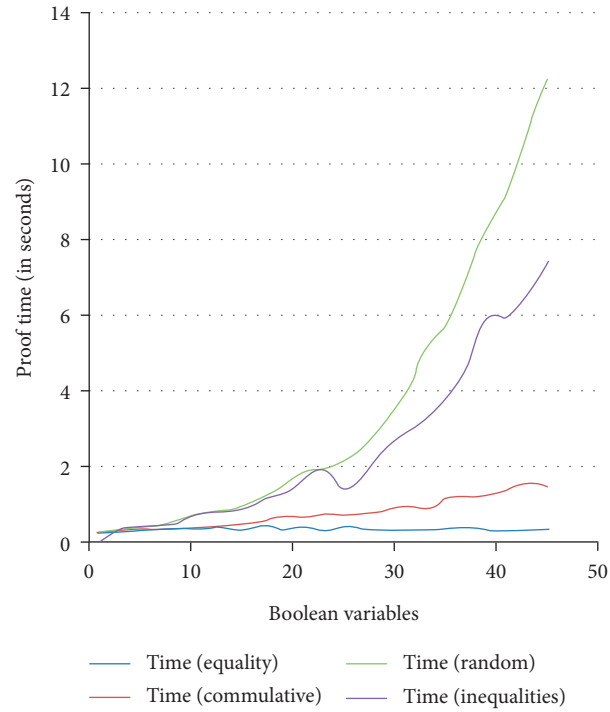

Figure 14: Performance evaluation of CoCEC.

7. Related Work

Equivalence checking of two functions, models, or circuits has been extensively studied in the literature. Recently, Khan et al. [22] built a formal model of Boolean algebra in the Coq proof assistant. Their model can be used to interactively prove the equivalence of Boolean functions or logic circuits modeled as Boolean functions. The most common approach to equivalence checking of hardware descriptions is to simulate the functions/descriptions in simulators such as VCS [25] and Icarus [26] by giving all possible inputs. Such methods are semiautomatic and are not reliable due to the lack of mathematical foundations. On the contrary, formal verification approaches [1], such as model checking [5, 6, 27] and theorem proving [28], are more popular in the literature. Formal tools such as ACL2, Coq, and Isabelle/HOL are well established, powerful, and more popular in the industry; however, the focus of this paper is to present a specialized tool for checking the equivalence of logical formulas that is simple and easy to use, automatic, and powerful (in terms of the underlying logic) and can generate the counterexample. There are other formal equivalence checking tools such as Conformal and FormalPro [29, 30], but they are used for equivalence checking at the RTL level, while CoCEC is aimed to check equivalence at the gate level. Both of these tools are proprietary and operate at different levels of circuit representation than CoCEC; hence, a comparison with CoCEC is not possible. Furthermore, none of these two tools is based on interactive proof assistants, which are more powerful than automated solvers.

7.1. Formal Hardware Verification. Formal verification techniques and tools can be used to verify the absence of faults in hardware system components. We refer to [31] for a detailed comparison of the formal method and simulation-based approaches to hardware designs' equivalence checking and verification. Kabat and Wojcik [32] suggested to use automated provers for the synthesis and verification of combinational logic. The authors used the rewriting logic demodulation for the simplification of canonical circuit structures. The most closely related tools are Son of BirdBrain II [9] and Solver Z3 [8]. BirdBrain is an automatic equivalence checker based on prover Prover9 for proving equivalence and Mace4 for finding a counterexample [33]. The proof checker Ivy behind BirdBrain II is based on the first-order logic, while Coq is based on the calculus of construction, which is more powerful and expressive than the first-order logic. Checking equivalence using BirdBrain is cumbersome: it requires disambiguating input using parenthesis, which is tedious, in particular, for large descriptions. Furthermore, every test requires proper selection of hypothesis from the given set of hypotheses. Z3 is a low-level software analysis and verification tool developed at Microsoft for checking satisfiability of logical formulas. Z3, a multipurpose tool, requires programming skills to encode formulas in the Z3 language syntax. None of these tools are based on interactive theorem proving and compatible with existing tools. These tools cannot be directly applied to check the equivalence of examples in text books [15] or formulas generated by other tools [20]. CoCEC, on the contrary, is based on interactive theorem proving with additional advantages [14] and accepts the input in a natural style without requiring any programming skills.

In the industry, automated theorem provers are more widely used; however, they are far behind proof assistants in terms of power and expressiveness. Model checking-based

tools can be used to check functions' equivalence; however, they are limited by the popular state explosion [7, 34] problem. Proof assistants include the advantages of both automated and interactive theorem provers. Proof assistants are currently investigated and encouraged for hardware verification [35, 36]. In this direction, Meredith et al. [37] defined executable semantics for HDL Verilog and embedded it in the theorem prover Maude [38] using rewriting logic as the underlying logic. Inspired from [37], Khan et al. [28] introduced a hardware description language, dubbed as VeriFormal. VeriFormal is a formal version of the hardware description language Verilog and is deeply embedded in the Isabelle/HOL theorem prover. Braibant and Chlipala [39] defined a language Fe-Si in Coq, which is a deeply embedded version of functional hardware description language blackspec. To produce the proof-carrying code, Love et al. [40] implemented a framework and formalized a synthesizable subset of Verilog in proof assistant Coq. Hasan et al. [36] introduced a formal framework in the higher-order logic of the theorem prover HOL. Their framework was intended for proving the reliability property of combinational circuits. These research contributions are mainly focused towards checking the reliability property. CoCEC enables functional equivalence in a stronger and expressive logic of Coq called the calculus of inductive constructions [35]. Furthermore, most of these research contributions address formal verification at a high level of abstraction and are based on semiautomatic tools. Our tool CoCEC, on the contrary, analyzes logic circuits at the gate level and is purely automatic. Interactive theorem proving approach is a viable alternative but at the cost of effort and expertise required to assist the theorem prover tools. At higher-level, such as at the RTL level, formal verification is equally important, but we consider it orthogonal to our work, which targets circuits at the gate level.

7.2. Tools Checking Equivalence of Boolean Expressions. To manipulate Boolean functions, a number of tools have been developed in the literature. Sondre et al. [41] created a database of Boolean functions. Their tool can prove properties of Boolean functions and can translate between different forms of functions. Another tool, the WolframAlpha computational engine [42], is used to translate the logical function given as input to a truth table and other minimal forms. Furthermore, WolframAlpha generates a Venn diagram and logic circuit for the corresponding input function. This engine has been included by a company TutorVista as a Boolean algebra calculator. To simplify Boolean functions, another tool [43] was developed. This tool uses Karnaugh maps [44] to reduce the number of literals and terms in Boolean functions. It gets the function as a sequence of notations or as a truth table (up to six variables).

One of the most recent tools to manipulate Boolean functions is 32×8 [19]. The tool 32×8 accepts a function as a truth table and returns as the output a Karnaugh map, Boolean function, and logic circuit for it. It supports up to eight-variable Boolean functions and can generate functions as the sum of product or product of sums form. Further tools were developed to simplify Boolean functions. Lean and Marxel developed a solver QMSolver [20] to simplify Boolean functions. The solver QMSolver, based on Quine–McCluskey algorithm, obtains the number of minterm indices (separated by spaces) and returns a simplified function. The output of these last tools can directly be given to the CoCEC tool. None of these Boolean function tools checks the equivalence of two circuit descriptions and does not have formal semantic.

8. Conclusions

Logic gate notations are used to describe logic circuits at the gate level during the electronic design automation process. It is common to manipulate circuit descriptions to produce power-efficient, small, and cheap logic circuit components. To ensure that the desired functionality is preserved, functional equivalence of two descriptions is often desired. In this paper, a formal tool CoCEC was developed to automatically prove the equivalence of two combinational circuit descriptions. The CoCEC tool is built on top of the formal framework for Boolean algebra developed in the Coq theorem prover. It accepts circuit descriptions in a natural style and translates them into equivalent descriptions in the Coq notations. Our tool automatically creates the proof of equivalence that can be checked mechanically. If circuit descriptions are functionally not equivalent, a counterexample is generated. CoCEC solves the equivalence problem with over a billion states in just few seconds. The robustness of the tool was tested by experimenting on a number of function pairs up to 45 variables.

The tool currently supports uniliteral variables, and it should be extended to accept variables consisting of multiple letters. Furthermore, CoCEC generates one counterexample; however, it can easily be extended to generate multiple or all (if any) counterexamples.

Abbreviations

CoCEC: Combinational circuit equivalence checker
ITP: Interactive theorem prover
ACL2: A Computational Logic for Applicative Common Lisp
HOL: Higher-order logic
CNF: Conjunctive normal form.

Data Availability

The source code of our tool (for Windows and Ubuntu platforms) is available online at https://github.com/wilstef/cocec.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Authors' Contributions

W. K. designed and developed the tool and defined the formal foundation in Coq. F. A. K. and A. D. contributed to

the design and development of the tool, carried out the experiments, and arranged the funding for this research. A. A., A. H., and W. K. created the paper draft and analysed the results.

Acknowledgments

The authors extend their sincere appreciation to the Deanship of Scientific Research at King Saud University, Saudi Arabia, for funding this work through the research group no. RGP-214. The work of Adi Alhudhaif was supported by the Deanship of Scientific Research at Prince Sattam Bin Abdulaziz University, Al-Kharj, Saudi Arabia.

References

[1] S. Y. Huang and K. T. T. Cheng, *Formal Equivalence Checking and Design Debugging*, Springer Science & Business Media, Berlin, Germany, 2012.

[2] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[3] C. Baier and C. Tinelli, *Some Advances in Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Berlin, Germany, 2017.

[4] W. Khan, M. Kamran, A. Ahmad, F. A. Khan, and A. Derhab, "Formal analysis of language-based android security using theorem proving approach," *IEEE Access*, vol. 7, pp. 16550–16560, 2019.

[5] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, Cambridge,, MA, USA, 1999.

[6] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[7] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," *Lecture Notes in Computer Science in Tools for Practical Software Verification*, vol. 7682, pp. 1–30, 2012.

[8] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963, pp. 337–340, 2008.

[9] M. William, "Son of birdbrain II," 2020, https://www.cs.unm.edu/%7Emccune/sobb/.

[10] B. Barras, S. Boutin, and C. Cornes, *The Coq proof assistant reference manual: version 6.1*, Ph.D. thesis, Inria, Rocquencourt, France, 1997.

[11] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Springer Science & Business Media, Berlin, Germany, 2002.

[12] M. Kaufmann and J. S. Moore, "Acl2 automated prover," 2020, https://www.cs.utexas.edu/users/moore/acl2/.

[13] T. Grimm, D. Lettnin, and M. Hübner, "A survey on formal verification techniques for safety-critical systems-on-chip," *Electronics*, vol. 7, no. 6, p. 81, 2018.

[14] R. C. Armstrong, R. J. Punnoose, M. H. Wong, and J. R. Mayo, *Survey of Existing Tools for Formal Verification*, Sandia National Laboratories, Albuquerque, New Mexico, 2014.

[15] M. M. Mano, *Digital Logic and Computer Design*, Pearson Education India, Noida, Uttar Pradesh, India, 2017.

[16] G. Boole, *Investigation of the Laws of Thought*, Dover, Illinois, IL, USA, 1854.

[17] E. V. Huntington, "New sets of independent postulates for the algebra of logic, with special reference to whitehead and russell's principia mathematica," *Transactions of the American Mathematical Society*, vol. 35, no. 1, pp. 274–304, 1933.

[18] C. E. Shannon, "A symbolic analysis of relay and switching circuits," *Electrical Engineering*, vol. 57, no. 12, pp. 713–723, 1938.

[19] 2018 Logic circuit simplification (SOP and POS). http://www.32x8.com/.

[20] R. Lean and M. Kryle, "QMSolver," 2018, http://agila.upm.edu.ph/%7Ekmmolina/qms/index.html.

[21] B. C. Pierce, C. Casinghino, and M. Gaboardi, "Software Foundations. Webpage," https://www.cis.upenn.edu/bcpierce/sf/current/index.html.

[22] W. Khan, M. Kamran, S. R. Naqvi, F. A. Khan, A. S. Alghamdi, and E. Alsolami, "Formal verification of hardware components in critical systems," *Wireless Communications and Mobile Computing*, vol. 2020, Article ID 7346763, 15 pages, 2020.

[23] A. V. Aho and J. D. Ullman, *Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[24] D. Delahaye, "A tactic language for the system Coq," in *Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 85–95, Springer, Alicante, Spain, May 2000.

[25] VCS Online Simulator, http://www.edaplayground.com/x/FfR, 2015.

[26] Icarus Verilog, http://www.icarus.com/eda/verilog/, 2016.

[27] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 1020 states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.

[28] W. Khan, A. Tiu, and D. V.F. Sanán, *An Executable Formal Model of a Hardware Description Language*, SG-CRC, Singapore, Singapore, 2017.

[29] Cadence, "Conformal Equivalence Checker," 2021, https://www.cadence.com/ko_KR/home/tools/digital-design-and-signoff/logic-equivalence-checking.html.

[30] Siemens, "FormalPro," 2021, https://eda.sw.siemens.com/en-US/ic/formalpro-equivalence-checking/.

[31] W. K. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series)*, Prentice Hall PTR, Hoboken, NJ, USA, 2005.

[32] W. C. Kabat and A. S. Wojcik, "Automated synthesis of combinational logic using theorem-proving techniques," *IEEE Transactions on Computers*, vol. C-34, no. 7, pp. 610–632, 1985.

[33] W. McCune, "Prover9 and Mace4: version 2009-11A," 2013.

[34] A. Valmari, *The State Explosion Problem. Lectures on Petri Nets I: Basic Models 1998*, Springer-Verlag, Berlin, Germany, 1998.

[35] S. Coupet-Grimal and L. Jakubiec, "Coq and hardware verification: a case study," *Lecture Notes in Computer Science*, Springer, in *Proceedings of the International Conference on Theorem Proving in Higher Order Logics*, pp. 125–139, August 1996.

[36] O. Hasan, J. Patel, and S. Tahar, "Formal reliability analysis of combinational circuits using theorem proving," *Journal of Applied Logic*, vol. 9, no. 1, pp. 41–60, 2011.

[37] P. Meredith, M. Katelman, J. Meseguer, and G. Roşu, "A formal executable semantics of verilog. Formal methods and

models for codesign (MEMOCODE)," in *Proceedings of the 8th IEEE/ACM International Conference on IEEE*, pp. 179–188, Massachusetts, MA, USA, May 2010.

[38] M. Clavel, F. Durán, J. Hendrix, S. Lucas, J. Meseguer, and P. Ölveczky, "The Maude formal tool environment," in *Proceedings of the International Conference on Algebra and Coalgebra in Computer Science*, pp. 173–178, Springer, Warsaw, Poland, August 2007.

[39] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis, computer aided verification," in *Proceedings of the International Conference on Computer Aided Verification*, pp. 213–228, Springer, Los Angeles, CA, USA, July 2013.

[40] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: a pathway to trusted module acquisition," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 25–40, 2012.

[41] R. Sondre, A. Mohamed, and D. Lars, "Online database of boolean functions," 2018, http://www.ii.uib.no/%7Emohamedaa/odbf/index.html.

[42] WolframAlpha computational knowledge engine, https://www.wolframalpha.com/, 2018.

[43] Online minimization of boolean functions: https://www.tma.main.jp/logic/index_en.html/,.

[44] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, vol. 72, no. 5, pp. 593–599, 1953.