Hindawi
Wireless Communications and Mobile Computing
Volume 2020, Article ID 7346763, 15 pages
https://doi.org/10.1155/2020/7346763

Research Article

Formal Verification of Hardware Components in Critical Systems

Wilayat Khan,[1] **Muhammad Kamran**,[2] **Syed Rameez Naqvi**,[1] **Farrukh Aslam Khan**,[3] **Ahmed S. Alghamdi**,[2] **and Eesa Alsolami**[2]

[1]*Department of Electrical and Computer Engineering, COMSATS University Islamabad, Wah Campus, Pakistan*
[2]*Department of Cyber Security, College of Computer Science and Engineering, University of Jeddah, Jeddah, Saudi Arabia*
[3]*Center of Excellence in Information Assurance (CoEIA), King Saud University, Riyadh, Saudi Arabia*

Correspondence should be addressed to Wilayat Khan; wilayatk@gmail.com and Farrukh Aslam Khan; fakhan@ksu.edu.sa

Received 2 July 2019; Revised 10 January 2020; Accepted 16 January 2020; Published 20 February 2020

Guest Editor: Fadi Al-Turjman

Hardware components, such as memory and arithmetic units, are integral part of every computer-controlled system, for example, Unmanned Aerial Vehicles (UAVs). The fundamental requirement of these hardware components is that they must behave as desired; otherwise, the whole system built upon them may fail. To determine whether or not a component is behaving adequately, the desired behaviour of the component is often specified in the Boolean algebra. Boolean algebra is one of the most widely used mathematical tools to analyse hardware components represented at gate level using Boolean functions. To ensure reliable computer-controlled system design, simulation and testing methods are commonly used to detect faults; however, such methods do not ensure absence of faults. In critical systems' design, such as UAVs, the simulation-based techniques are often augmented with mathematical tools and techniques to prove stronger properties, for example, absence of faults, in the early stages of the system design. In this paper, we define a lightweight mathematical framework in computer-based theorem prover Coq for describing and reasoning about Boolean algebra and hardware components (logic circuits) modelled as Boolean functions. To demonstrate the usefulness of the framework, we (1) define and prove the correctness of *principle of duality* mechanically using a computer tool and all basic theorems of Boolean algebra, (2) formally define the algebraic manipulation (step-by-step procedure of proving functional equivalence of functions) used in Boolean function simplification, and (3) verify functional correctness and reliability properties of two hardware components. The major advantage of using mechanical theorem provers is that the correctness of all definitions and proofs can be checked mechanically using the type checker and proof checker facilities of the proof assistant Coq.

1. Introduction

Hardware, software, and communication networks altogether make systems operating in the cyberspace such as Unmanned Aerial Vehicles (UAVs). UAVs operating in an uncertain, potentially hazardous, remote, and dynamic environment are extremely important but challenging to be reliable, robust, and secure. These flying vehicles are currently being used in mission-critical [1], industry-oriented [2], and Internet of Things (IoT) [3] applications, to name a few. To meet the high standards of safety and reliability of UAV-based mission-critical [1] or IoT [2, 3] applications, the UAVs in such systems must be studied and analysed using rigorous and formal techniques. Failure or unauthentic use of software and hardware systems of air vehicles can lead to human losses [4, 5] and strategic losses [6, 7]. UAV-based human environment can be manipulated and controlled by a remote attacker using attacks such as *sensor input spoofing attack* [8]. Military UAVs have been and are currently the favourite target of attackers to gain cyber power [9], which has recently led towards a *drone war* [10].

To design fault-tolerant and secure air vehicles, conventional design and testing methods must be augmented with more robust and reliable tools and techniques called formal methods [11]. Formal methods have been successfully used to verify collision avoidance between UAVs [12], certify UAVs within civil airspace [13], and design resilient UAV systems [14]. As the security and reliability of UAVs depend upon the security and reliability of individual components, their correctness must be ensured in the design phase. In addition to

UAVs, the formal framework presented in this paper is equally applicable to any other computer system such as sensor networks [15–17] and hardware components, such as flash memory, of such systems [18]. In this paper, we address the formal specification and verification of hardware (logic) components, such as memory units and adders, designed at the logic gates level using Boolean algebra. This makes our formal model extremely important in the domain of critical system design in general and in UAVs in particular.

Boolean algebra [19] is the basic logic tool for the analysis and synthesis of (models of) logic circuits. This connection between Boolean algebra and logic circuits was established by Claude Shannon [20]. While different computer tools [21–25] and mathematical techniques [26, 27] are available to manipulate logic circuits modelled as Boolean functions, it must be ensured that such tools and techniques do not alter the intended interpretation (behaviour) of these circuits. To verify that the behaviour of the digital circuit (modelled as Boolean function) after the analysis is symmetric to the original behaviour, the Boolean function *before* the mathematical manipulation must be proven to be functionally symmetric to the function *after* the manipulation. This property of digital circuits is often referred to as functional equivalence [28]. Common examples of manipulation are mathematical manipulation [29], Karnaugh map [26], and tabulation method [27] used for the Boolean functions.

Boolean algebra is one of the most widely used mathematical techniques to model and analyse logic circuits. In the electronic design flow, the initial circuit design is often described at system level in high-level languages (e.g., MATLAB and C) and translated to register-transfer logic (RTL) representation in a description language (e.g., Verilog and VHDL) using high-level synthesis (hardware compilers) tools. The RTL representation of the circuit is transformed to gate-level representation (often as Boolean functions) using logic synthesis tools, which is finally fabricated to produce physical layout of the circuit. The gate-level representation described as Boolean functions is commonly used in techniques and frameworks [30–33] and in classrooms for the design and analysis of simple logic circuits [29].

1.1. Research Challenges. Boolean functions or logic circuits described as Boolean functions are normally manipulated through the error-prone pen-and-paper method using the basic theorems and postulates of Boolean algebra. To the best of our knowledge, neither the algebra nor the mathematical manipulation process has been formally defined in an interactive theorem prover and hence their correctness cannot be checked mechanically. There is no guarantee that the principle of duality indeed holds, the mathematical manipulation carried out is correct, or the logic circuits (described as Boolean functions) optimized for size, efficiency, and cost-effectiveness using K-map [26] or tabulation [24] methods are behaviourally symmetric.

1.2. Solution Overview. In order to formally reason about Boolean algebra and verify the correctness of digital components described at gate level as Boolean functions, we define a formal model of the Boolean algebra using the calculus of construction in theorem prover Coq. The formal model of Boolean algebra defined enables one to define and prove all the basic theorems as well as the principle of duality. We extend our formal model of the Boolean algebra to represent combinational circuits. To assess the efficacy of our formal model, the proof facility of Coq is used to carry out proof of correctness of gate-level combinational circuits and reason about Boolean algebra. Among other numerous advantages (Section 2) of computer-aided verification using interactive theorem prover are the following: (a) all the formal definitions and proofs can be defined in the computer, and (b) the correctness of the proofs can be automatically checked by the computer [34].

The formal approach for checking correctness of Boolean functions and digital circuits is described in Figure 1. Formal models of the digital circuit under verification and the properties of interest as theorems are fed into an ITP engine (Coq system, in our case) and a formal proof that the (model of the) circuit holds the properties is carried out interactively. A proof engineer guides the tool by providing it proof commands, which becomes part of the proof script if accepted by the tool. The correctness of the proof script is automatically checked using the tool. The formal verification in Coq using the proposed formal model is demonstrated in Section 6 by proving equivalence, reversibility, and type safety properties of multiple circuits. In this context, the major contributions of this paper are the following:

(i) A computer-based mathematical model for describing and reasoning about Boolean functions and gate-level combinational circuits is defined. Our formal model is novel as it enables, in addition to Boolean algebra and basic theorems, defining the principle of duality and logic circuits in a computer-based theorem prover. Furthermore, the model together with the basic theorems allows one to mechanize the mathematical manipulation process, which can be checked correctly using computer.

(ii) Formal proofs of the principle of duality and basic theorems of Boolean algebra are carried out. In the literature, there is no computer-based proof of validity of the principle of duality and proof of basic theorems listed in most popular books such as [29, 36].

(iii) Formal proofs of equivalence and reversibility of multiple combinational circuits are carried out. To reduce the propagation delay in a multibit adder, a look-ahead carry generator circuit is added using a step-by-step process; however, there is no proof that it does not alter the intended behaviour of the adder. We formally prove that the look-ahead carry generator preserves the functional behaviour of the adder.

(iv) The performance of the Coq theorem prover over our model is evaluated.

The rest of the paper is organized as follows. In the next section, the significance of interactive theorem proving is highlighted. The tool Coq theorem prover and Boolean

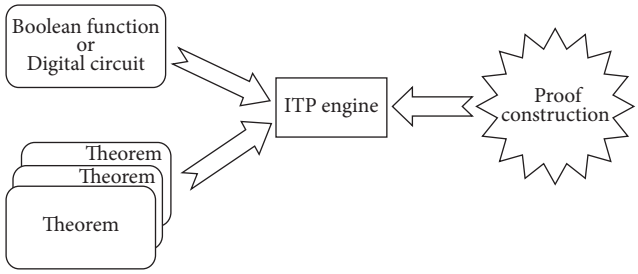

Figure 1: Formal verification of Boolean functions and digital circuits [35].

algebra are introduced in Section 3. The formal model based on Boolean algebra is defined in theorem prover in Section 4. Boolean algebra and logic circuits are reasoned about in Sections 5 and 6, respectively. Our formal model is evaluated and results are discussed in Section 7. A critical review of the related work is given in Section 8 and Section 9 concludes the paper. All the Coq source codes are available from our GitHub repository at https://www.github.com/wilstef/booleanalgebra.

2. Why Interactive Proof Assistant?

A common approach to check the two circuits (represented as Boolean functions) for functional equivalence is to extensively simulate them against many inputs and compare the results. Simulation-based testing is the most popular approach in industry because it is easy to use; however, it fails to ensure absence of faults in the system. Furthermore, simulating systems with large inputs are not computationally feasible; for instance, simulating a 256 bit memory chip would require testing it for 2^{256} possible inputs. Another approach to check correctness of the digital circuits in general, or equivalence checking in particular, is to use formal methods based on computer-based mathematical tools and techniques. The prevailing advantages of formal tools and techniques are that they are rigorous and computer-aided and can be used to formally prove properties for all the possible inputs. Moreover, formal methods based techniques can be used to ensure the absence of faults in the system.

According to a recent survey [35], most of the formal verification methods and tools are based on model checking and automated theorem provers that are restricted by well-known state and memory explosion [37, 38] problems. Formal verification using model checking is popular in industry [39, 40] and well-studied domain by the research community [40–42]; however, the focus of this paper is on interactive theorem proving (ITP) [43–45] approach. In the ITP-based formal approach, a proof engineer guides a computer-based proof assistant, such as Coq and Isabelle/HOL [45], by providing proof commands during the proof process. The ITP-based formal verification, also referred to as semiautomatic approach, combines the strengths of manual and automated proofs. This semiautomatic ITP approach requires expertise and skills; however, it has been used in the past for investigating large case studies [46]. In automated theorem proving, it is hard to get insights when a proof attempt fails, while ITP forces the designer to pay attention to even the minor details. This results in understanding the system under study more precisely.

The conventional mathematical proofs on pen and paper (informal) are error-prone and difficult to manage large and complex proofs. Computer-aided verification, on the other hand, is an effective, efficient, and rigorous way of formal specification and verification. Automated theorem provers are widely used in industry as they automatically create proofs without requiring human effort; however, they face problems such as state explosion [37, 38]. Human-assisted theorem provers, on the other hand, require human support to carry out mathematical proofs. A proof engineer guides the theorem prover by providing proof commands and interactively creating proofs. Coq is one such popular interactive proof assistant considered for defining our formal framework.

Mathematical proofs carried out in a human-assisted theorem prover are more organized because of the following reasons: (i) in mechanized theorem prover (such as Coq), the proofs can be divided into modules (to make proof handling easy); (ii) lemmas/theorems already proven can be easily invoked and applied; (iii) the Coq Language of Tactics (Ltac) can be used to combine complex set of proof commands (tactics) in a single tactic; (iv) the proofs can be read and checked by the computer using a proof checker; (v) a new proof can be opened inside an existing unfinished proof; (vi) large proof scripts can be checked for unproved lemmas just by using a single Coq command; and so on. In summary, the proofs in proof assistants such as Coq are more organized as compared to pen-and-paper proof methods and the correctness of the proof scripts can be checked by the computer.

3. Background

To mathematically prove the properties of Boolean algebra and verify correctness of digital circuits using computer, the algebra must be defined in the logic of a mechanical (computer-readable) proof assistant such as Coq. Such a formal definition of Boolean algebra is given in Section 4. To understand the formal definition, the Boolean algebra and the formal tool Coq used are introduced in this section. For an in-depth understanding of circuit modelling using Boolean algebra and Coq theorem prover, the readers are recommended to refer to books [29, 47], respectively.

3.1. Coq Proof Assistant. Coq [44] is an interactive theorem prover based on the calculus of inductive construction. It is available with a language of specification called Gallina, a type checker and a proof checker. The language, type checker and proof checker are used for creating formal specifications of the systems under test, checking the specifications for type errors, creating and checking formal proofs, respectively. To understand formalizing and proving systems using interactive theorem prover, we define a simple system of numbers using the proof assistant Coq and then give reasoning about the system.

We begin with formally defining numbers inductively as data type `Nat` using the Coq keyword `Inductive` with two constructors for generating elements of the type `Nat` (lines 1–3, Listing 1). The definition `Nat` in this listing states that `O` (for 0) is `Nat` and if `n` is `Nat`, `Succ n` is also `Nat`. The first constructor has no argument and simply states that `O` is a member of type `Nat`. The second constructor `Succ` has one argument of type `Nat` which states that `S` followed by a `Nat` (the `Nat` on left of the arrow) is also a `Nat` (the `Nat` on right of the arrow). The term `Succ (Succ (Succ O))`, for example, is a `Nat` that corresponds to number 3.

To operate on numbers, we define a recursive function `Add` (lines 5–9) to add values of type `Nat`. The function returns the second argument `m` if the first argument is `O`; otherwise, it returns `Succ (Add n′ m)`. A lemma `Add_N_O`, where `Add n O = n` holds for any value of `n`, is stated and proven in Listing 1 (lines 11–18). The proof of this lemma has been carried out by applying induction on the construction of the first argument `n`. While carrying out the proof, the Coq tool was interactively guided by giving proof commands called *tactics* (lines 13–17). Many other properties of the `Add` function, such as commutative and associative properties, may also be stated and proven mechanically using the Coq proof assistant.

3.2. Boolean Algebra. According to George Boole [19], Boolean algebra is an algebraic structure with a set of values, two binary operations "+" and "." over the values in the set and proof of Huntington [48] postulates. Later on, Shannon [20] introduced a two-valued version of the algebra to represent properties of the switching circuits. In a two-valued algebra, the set of values include elements 1 and 0 and the two binary operations are conjunction (logical AND) and disjunction (logical OR). In addition, there is a unary operation called negation or complement (NOT).

Using Boolean functions in the two-valued Boolean algebra, logic circuits can be modelled. The function $F = \overline{x} \cdot \overline{y}$ models a logic circuit where the AND gate represents the product operation and the NOT gate represents the complement operation. When logic circuits are represented with functions, they can be manipulated and reasoned about using tools and techniques developed for Boolean algebra. For example, the algebraic manipulation steps using postulates can be written down in and checked using Coq tool (see Section 5).

Shannon's two-valued version of the Boolean algebra is defined in Coq theorem prover as described in Listing 2 [47]. The first part of Boolean algebra, a set of two elements, is represented with Coq type `bool` defined using the Coq keyword `Inductive` as shown in Listing 2 (line 1). The two values of the Boolean algebra defined are `true` and `false`.

The second part of Boolean algebra is to define two binary operations over the values of the Boolean type `bool`. The rules for the first operation + (sum) are defined as function `sum` on lines 3–7 in Listing 2. Using pattern matching, the function `sum` gets two values of type `bool` and returns value `false` if both input values are `false`; otherwise, it returns `true`. The rules for the second binary operation . (product) are defined in function `prod` (lines 9–13, Listing 2). This function returns `true` only if the two input values are `true`; otherwise, it returns `false`. A unary operation. complement on elements of Boolean set is defined in the function `not` on lines 15–19 in the listing. Given one value as the input, `not` returns the other value. Among these operations, the operation ¬ has the highest and + has the lowest precedence.

4. Formalizing Logic Circuits and Boolean Algebra

The Boolean algebra defined in Section 3 is extended and tailored towards combinational circuit definitions. To begin with, the set of notations in Listing 3 is extended with symbol ⊕ (Listing 4) for `xor` operation with same precedence as. (operation `prod`). The formal definitions of logic operations `sum`, `prod`, `xor`, and `not` model the basic logic components OR, AND, XOR, and NOT gates, respectively. After formally modelling these basic logic components, they are combined together to form combinational circuits. A combinational circuit is defined as a list of Boolean (functions) values (Listing 5).

The list "$\texttt{F}^1$ `x y::`$\texttt{F}^2$ `x y::nil`" represents a combinational circuit with two inputs `x` and `y` and two outputs defined as two Boolean functions. To evaluate each Boolean function in the list to a Boolean value, an evaluation function `eval_cir` is defined in Listing 6. The function takes a combinational circuit as list of Boolean functions and returns a list of Boolean values (Listing 7).

The final requirement of Boolean algebra is to formalize the six Huntington postulates [29, 48] and prove the set of all basic theorems. Formal definitions of Huntington postulates are included in the Coq script available from our repository. Furthermore, the formal definitions of operation `sum`, `prod`, and `not` are sufficient to reason about Huntington postulates and all the basic theorems except principle of duality (discussed in Section 5). As the derivation in principle of duality requires operating on the Boolean expressions, the principle cannot be defined in the current setting. The existing definitions of expressions do not differentiate between variables and values (identity elements), which is the basic requirement of operation in the duality. To do this, a type `exp` for Boolean expressions is defined as shown in Listing 8. The first three constructors (line 2–4) correspond to the three logical operations `sum`, `prod`, and `not`, respectively. The last constructor converts a Boolean term, variable, or value to an expression.

All the postulates and basic theorems, except the *closure*, *involution*, and *consensus*, have two parts: one part is for the operation + (`sum`) and the other is for . (`prod`). According to the *principle of duality*, one part of the postulate/theorem can be derived from the other if the operators and the identity elements are interchanged. The duality principle is significant in proving the theorems using postulates and in algebraic manipulation. In particular, when the proof of one part of a theorem is given, the other can easily be carried out following the principle of

```
(1) Inductive Nat : Type :=
(2)   |O : Nat
(3)   |Succ : Nat ⟶ Nat.
(4) Fixpoint Add (n m: Nat) : Nat :=
(5) match n with
(6)   |O ⟹ m
(7)   |Succ n′ ⟹ S (Add n′ m)
(8) end.
(9) Lemma Add_N_O: ∀ n:Nat, Add n O = n.
(10) Proof.
(11)   induction n.
(12)   (∗CASE 1: n is O∗)
(13)     reflexivity.
(14)   (∗CASE 2: n is (S n)∗)
(15)     simpl. rewrite IHn. auto.
(16) Qed.
```

Listing 1: Example of interactive formal proof in Coq.

```
(1) Inductive bool: Type := true | false.
(2) Definition sum (x y: bool) : bool :=
(3)   match x, y with
(4)   | false, false ⟹ false
(5)   | _, _ ⟹ true
(6)   end.
(7) Definition prod (x y: bool): bool :=
(8)   match x, y with
(9)   | true, true ⟹ true
(10)  | _, _ ⟹ false
(11)  end.
(12) Definition not (x: bool): bool :=
(13)  match x with
(14)  | false ⟹ true
(15)  | true ⟹ false
(16)  end.
(17) Definition circuit := list bool.
```

Listing 2: Formal definition of Boolean operations.

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

Listing 3: Shorthand notations.

```
(1) Notation "x ⊕ y" := (x∗¬y + ¬x∗y)
(2)             (at level 40, left associativity) : bool_scope.
```

Listing 4: Notation for operation xor.

```
(1) Definition circuit := list bool.
```

Listing 5: Definition of `circuit`.

```
(1) Fixpoint eval_cir (c: circuit) : list bool :=
(2) match c with
(3) | nil ⟹ nil
(4) | cons c tl ⟹ cons c (eval_cir tl)
(5) end.
```

Listing 6: Function eval_cir for evaluating circuit.

```
(1) Definition F(x y: bool) : circuit := ¬x∗¬y::nil.
```

Listing 7: Example of circuit description in Coq notations.

```
(1) Inductive exp : Type :=
(2)     | sumexp: exp ⟶ exp ⟶ exp
(3)     | prodexp: exp ⟶ exp ⟶ exp
(4)     | compexp: exp ⟶ exp
(5)     | bexp: bool ⟶ exp.
```

Listing 8: Definition of data type exp.

duality. The derivation carried out this way is believed to be valid according to the principle of duality in literature and textbooks [29]; however, no formal proof has been provided. We bridge this gap by providing a formal proof of duality principle.

A function `changeident` (Listing 9) is defined to interchange identity elements in the Boolean expression. It gets an expression and swaps the identity elements `true` and `false` and leaves the operators and variables unchanged. The keyword `Fixpoint` is used to define recursive functions. Another operation for interchanging operators is defined as a recursive function `changeop` in Listing 10. Similarly, the function `changeop` gets an expression and swaps the operators `sum` and `prod`.

5. Formal Proofs of Boolean Algebra

The formal definitions in Section 4 enable one to formally reason about both Boolean algebra and combinational circuits described as Boolean functions. This section contains proof of duality principle and demonstrates with examples that our formal setting mechanizes the informal algebraic manipulation used in textbooks on Digital Logic Design. Furthermore, all the basic theorems and postulates have been defined and proven in Coq and given in the Coq script available from our repository.

```
(1) Fixpoint changeident (e : exp) : exp :=
(2) match e with
(3)     | trueexp ⟹ falseexp
(4)     | falseexp ⟹ trueexp
(5)     | bexp e′ ⟹ e
(6)     | sumexp e1 e2 ⟹
(7)       sumexp (changeident e1) (changeident e2)
(8)     |prodexp e1 e2 ⟹
(9)       prodexp (changeident e1) (changeident e2)
(10)    | compexp e′ ⟹ compexp (changeident e′)
(11) end.
```

Listing 9: Function changeident for interchanging identity elements.

```
(1) Fixpoint changeop (e : exp) : exp :=
(2) match e with
(3)     | sumexp e1 e2 ⟹ prodexp e1 e2
(4)     | prodexp e1 e2 ⟹ sumexp e1 e2
(5)     | _ ⟹ e
(6) end.
```

Listing 10: Function changeop for interchanging operators.

5.1. Principle of Duality. The duality principle states that expressions derived from postulates by interchanging the operators (+ and .) and identity elements (0 and 1) are valid. After defining the interchange operations over the Boolean expressions, the duality principle can now be stated as a lemma as shown in Listing 11. The lemma states that if a part of theorem holds (two arbitrary expressions are equal), then it implies that the second part of the theorem can be derived by changing the identity elements and operators. To check that the duality property can be used in proofs, given the first part of commutative property ($x + y = y + x$), the second part of commutative property ($x \cdot y = y \cdot x$) is proven (Listing 12) by applying the duality property.

5.2. Mechanizing Algebraic Manipulation. To get simplified logic circuit, the number of literals and terms must be reduced in the description of the circuit. The algebraic manipulation method is one of the most popular methods used in textbooks [29] for this purpose. This kind of manipulation is informal and hence is error prone. The formal structure including the postulates and basic theorems defined in previous sections can be applied, as in the informal algebraic manipulation, using Coq proof assistant. The main advantage of mechanizing algebraic manipulation in proof assistant is that proof scripts can be mechanically read, checked, and maintained.

To demonstrate that our formal framework is fit for mechanically checking correctness of algebraic manipulation, proof of the theorem `absorption_sum` has been listed in Listing 13. This proof is carried out by applying (using the `rewrite` tactic) the postulates and theorems. This proof

```
(1) Lemma duality: ∀ lhs rhs,
(2)   lhs = rhs ⟶
(3)   changeop (changeident lhs) = changeop (changeident rhs).
```

Listing 11: Lemma stating the duality property.

```
(1) Lemma duality_check: ∀ x y,
(2)   sumexp (bexp x) (bexp y) = sumexp (bexp y) (bexp x) ⟶
(3)     prodexp (bexp x) (bexp y) = prodexp (bexp y) (bexp x).
```

Listing 12: Example of proof using the duality property.

```
(1) Lemma absorption_sum: ∀ x y, x + x * y = x.
(2)   Proof.
(3)     intros.
(4)     pattern x at 1.
(5)     rewrite ⟵ pos_identity_prod.
(6)     rewrite ⟵ pos_dist_over_sum.
(7)     rewrite ⟵ pos_comm_sum.
(8)     rewrite identity_sum_1.
(9)     rewrite pos_identity_prod.
(10)    auto.
(11)  Qed.
```

Listing 13: Proof of absorption law (sum).

script is a mechanized version of the informal proof of Theorem 6 (a) in the textbook [29].

Proof of the second part `abroption_prod` of the same theorem is believed to hold by the duality principle. This principle has been proven to hold in the above section. Every postulate or basic theorem has two parts, where one is the dual of the other. This is interesting to show that, given step-by-step proof of one theorem, proof of the other (dual) part can be carried out by applying the dual of postulate/theorem applied at the corresponding step. The proof of the theorem `abroption_prod` (dual of theorem `absorption_sum`) has been listed in Listing 14. The postulates/theorems applied on lines 5–9 in both proofs are the dual of each other.

6. Describing and Verifying Combinational Circuits

UAVs or drones are like flying computers with Linux or Windows operating system, flight controllers, main boards, memory units, and thousands of lines of programmable code. Formal methods, and in particular our framework, can be applied to the analysis of UAVs in different ways. As mentioned, it is a complex computer system, and hence formal techniques can be used to verify the operating system, protocols, logic components, memory units, and any algorithm used in the UAV system. As arithmetic and logic components (e.g., adder and comparator) and memory (e.g., ROM) are the main components of any typical computer (including UAVs), we apply our formal framework to analyse a memory and an adder circuit. In this section, functional equivalence and reversibility properties of few combinational circuits are proven. Furthermore, it is also proven that all the described circuits hold the basic reliability property called type-safety.

6.1. Verifying Equivalence of Memory Circuits. To begin with, a 32×8 ROM memory chip with content *Trusted designs*, stored as the ASCII data on the first sixteen locations, is specified in a truth table as shown in Table 1 (entries for the first 16 inputs are defined and shown in the table. The rest of entries are *do not care* conditions.). The letters A–E represent the five-bit input (adders lines) and O_7–O_0 show the eight-bit output (data lines) of the memory. The table specifies the input-output relationship: a five-bit input selecting any of the first sixteen locations; the chip returns ASCII code of a letter of the content *Trusted designs*.

To design the memory chip, first each output of the memory chip is defined as a Boolean function of the five input variables A–E. A function for each output is formed by combining (through operator +) all the minterms, where the value of the corresponding output column is 1. This would result in eight Boolean functions $O_7 - O_0$, represented in *sum-of-minterms* [29, 36] forms, as shown in Listing 15 (the

```
(1) Lemma abroption_prod: ∀ x y, x ∗ (x + y) = x.
(2)    Proof.
(3)       intros.
(4)       pattern x at 1.
(5)       rewrite ⟵ pos_identity_sum.
(6)       rewrite ⟵ pos_dist_over_prod.
(7)       rewrite ⟵ pos_comm_prod.
(8)       rewrite identity_prod_0 with (x := y).
(9)       rewrite pos_identity_sum.
(10)      auto.
(11)   Qed.
```

Listing 14: Proof of absorption law (product).

Table 1: Truth table for 32×8 ROM chip.

Address					Data								Symbol
A	B	C	D	E	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0	
0	0	0	0	0	0	1	0	1	0	1	0	0	"T"
0	0	0	0	1	0	1	1	1	0	0	1	0	"r"
0	0	0	1	0	0	1	1	1	0	1	0	1	"u"
0	0	0	1	1	0	1	1	1	0	0	1	1	"s"
0	0	1	0	0	0	1	1	1	0	1	0	0	"t"
0	0	1	0	1	0	1	1	0	0	1	0	1	"e"
0	0	1	1	0	0	1	1	0	0	1	0	0	"d"
0	0	1	1	1	0	0	1	0	0	0	0	0	
0	1	0	0	0	0	1	1	0	0	1	0	0	"d"
0	1	0	0	1	0	1	1	0	0	1	0	1	"e"
0	1	0	1	0	0	1	1	1	0	0	1	1	"s"
0	1	0	1	1	0	1	1	0	1	0	0	1	"i"
0	1	1	0	0	0	1	1	0	0	1	1	1	"g"
0	1	1	0	1	0	1	1	0	1	1	1	0	"n"
0	1	1	1	0	0	1	1	1	0	0	1	1	"s"
0	1	1	1	1	0	0	1	0	1	1	1	0	"."

product and sum terms of each function are given in the source code.). The 32×8 ROM chip is defined (Listing 16) as list of seven Boolean functions in Listing 15. This is a 5×8 combinational circuit with five inputs and eight outputs. The five-input bits of the chip identify one of the 2^5 memory locations and the output lines give the 8-bit ASCII code of the character stored at that location.

To design a simple and efficient memory chip, all the eight functions (for outputs O_7–O_0) are simplified using K-map method to functions $O'7$–$O'0$ as shown in Listing 17. The simplified Boolean functions have fewer literals and terms (Listing 17) as compared to original functions (Coq Definitions $O'7$–$O'0$ in source code) and would produce a simple and efficient circuit layout (see Section 7 for a circuit layout example). The simplified Boolean functions in Listing 17 representing outputs of the memory chip are combined together in a list to form the simplified combinational circuit as shown in Listing 18.

The formal model of the combinational circuit in Listing 16 has been simplified following the well-known simplification method K-map to circuit in Listing 18; however, it is not guaranteed that the circuits are functionally equivalent. In other words, there is no mathematical guarantee that the

```
(1) Definition O7(A B C D E : bool) : bool := false.
(2) Definition O6(A B C D E : bool) : bool :=
(3)    ∑(0–6, 8–14).
(4) Definition O5(A B C D E : bool) : bool :=
(5)    ∑(1–15).
(6) Definition O4(A B C D E : bool) : bool :=
(7)    ∑(0–4, 10, 14).
(8) Definition O3(A B C D E : bool) : bool :=
(9)    ∑(11, 13, 15).
(10) Definition O2(A B C D E : bool) : bool :=
(11)   ∑(0, 2, 4–6, 8, 9, 12, 13, 15).
(12) Definition O1(A B C D E : bool) : bool :=
(13)   ∑(1, 3, 10, 12–15).
(14) Definition O0(A B C D E : bool) : bool :=
(15)   ∑(2, 3, 5, 9–12, 14).
```

Listing 15: Boolean functions for ROM outputs O_7–O_0.

K-map simplification process has not altered the original behaviour of the circuit. The initial design built from the truth table specification serves as the "golden" or "reference" design. To prove the simplified circuit is functionally equivalent to the "golden" design, the individual functions are proven to be equivalent in Theorem 1 (this theorem corresponds to a set of eight lemmas `equiv_O0' 0-equiv_O0' 7` in Coq script.). Functional equivalence of the two circuits is checked in Theorem 2. The theorem states that the circuit *before* simplification is equivalent to the circuit *after* simplification, which guarantees that the simplification process preserves the functionality.

Theorem 1 (Equivalence of Boolean Functions): For any function O_i, where $i = 0, 1, 2, \ldots, 7$, Boolean function simplification operation S_f and functional equivalence relation $\approx$, $O_i \approx S_f(O_i)$.

Proof: This theorem is proven using case analysis on the Boolean variables used. The Coq proof of this theorem is listed at our GitHub repository at https://github.com/wilstef/booleanalgebra.

Theorem 2 (Functional Equivalence of Circuits): The combinational circuit C described in Listing 16 and its simplified version C_s in Listing 18 are functionally equivalent. More formally, $C \approx C_s$, where $C_s = S_{Kmap}(C)$ and S_{Kmap} is K-map simplification operation.

Proof: This theorem is proven by applying (rewriting) the eight lemmas already proven in Theorem 1. The Coq proof of this theorem is listed at our GitHub repository at https://github.com/wilstef/booleanalgebra.

6.2. Verifying Equivalence of Combinational Circuits. Arithmetic component, such as adder, is an integral part of the arithmetic and logic unit (processor) of any microprocessor-based system. To design energy-efficient and fast components, their design description (normally in a description language) is often transformed and mathematically manipulated. However, it must be ensured that such transformations do not alter the intended behaviour of these components. This surety is even more necessary when the

```
(1) Definition ROM32x8_orig (A B C D E: bool): circuit :=
(2)     O7 A B C D E::O6 A B C D E::O5 A B C D E::O4 A B C D E::O3 A B C D E::O2 A B C D E::
(3)     O1 A B C D E::O0 A B C D E::nil.
```

Listing 16: Formal definition of 32×8 ROM chip.

```
(1) Definition O′7(A B C D E:bool) : bool := false.
(2) Definition O′6(A B C D E:bool) : bool :=
(3)     ¬C +¬D + ¬E.
(4) Definition O′5(A B C D E:bool) : bool :=
(5)     B + C + D + E.
(6) Definition O′4(A B C D E:bool) : bool :=
(7)     ¬B∗¬C + ¬B∗¬D∗¬E + B∗D∗¬E.
(8) Definition O′3(A B C D E:bool) : bool :=
(9)     B∗C∗E + B∗D∗E.
(10) Definition O′2(A B C D E:bool) : bool :=
(11)     ¬B∗¬E + B∗¬D + C∗¬D +¬D∗¬E + B∗C∗E.
(12) Definition O′1(A B C D E:bool) : bool :=
(13)     ¬B∗¬C∗E + B∗C + B∗D∗¬E.
(14) Definition O′0(A B C D E:bool) : bool :=
(15) ¬B∗C∗¬D∗E + ¬C∗D + B∗¬C∗E + B∗C∗¬E.
```

Listing 17: Simplified Boolean functions for ROM outputs.

```
(1) Definition ROM32x8_simpl (A B C D E: bool): circuit :=
(2)     O′7 A B C D E::O′6 A B C D E::O′5 A B C D E::O′4 A B C D E::O′3 A B C D E::O′2 A B C D E::O′1 A B C D E::O′0 A B C D E::nil.
```

Listing 18: Formal definition of 32×8 ROM chip (simplified).

component being designed is used in critical systems, such as UAVs. To demonstrate that our formal definitions can also be used to prove functional equivalence of two combinational circuits, we describe two versions of a Binary Coded Decimal (BCD) adder (Figure 2) in Coq. Such a BCD adder has been designed and described in the popular textbook [29] on the subject Digital Logic Design; however, no formal proof of equivalence has been provided.

A BCD adder adds two decimal digits and their sum results as a BCD number. A 4-bit BCD adder may be designed using two 4-bit binary parallel adders as shown in Figure 2 (right). The conventional 4-bit binary adder, comprised of four full adders, is a serial adder that adds two 4-bit binary numbers in serial fashion. Each full adder gets two bits (the augend and addend bits) and an input carry bit and gives a sum bit and an output carry bit. The output carry of one full adder is given as input carry to the next one and the input carry of the first (right most) adder is set to logic 0. The output carry of the last adder is the output carry of the 4-bit binary adder.

The initial carry, given as input to the first adder, propagates to the last adder, which results in a *propagation delay*. To avoid this propagation delay in the binary adder, a look-ahead carry generator [29] is used to predict the input carries. The look-ahead carry generator is a small-scale integrated circuit added in front of the 4-bit adder to make it fast. The transformation avoids propagation delay; however, no formal proof that the functional properties of the adder are preserved is provided. In this section, we formally prove that two implementations of the BCD adder (Figure 2), one with and the other without look-ahead carry generator circuit, are functionally equivalent.

First, we design a BCD adder using binary adders without look-ahead carry generator. The 4-bit parallel full adder `FA4bit` is defined as a function in Listing 19. It applies the full-adder function `fa` to the 4-bit pairs of the two 4-bit numbers and input carry (inputs D–A, d–a and e), and returns a 5 bit tuple. The first bit in the tuple is the final carry and the next four bits are the sum bits. The binary adder `FA4bit` is used to implement the BCD adder as shown in Listing 20. The let-expressions on lines 2, 3, and 4 calculate the result of the first binary adder, output carry, and second binary adder, respectively. The additional circuitry between the two binary adders is needed to generate the output carry of the BCD adder and to adjust the sum of the first binary adder to BCD sum.

To generate a BCD adder with look-ahead carry generator, we must first design binary parallel adder with look-ahead

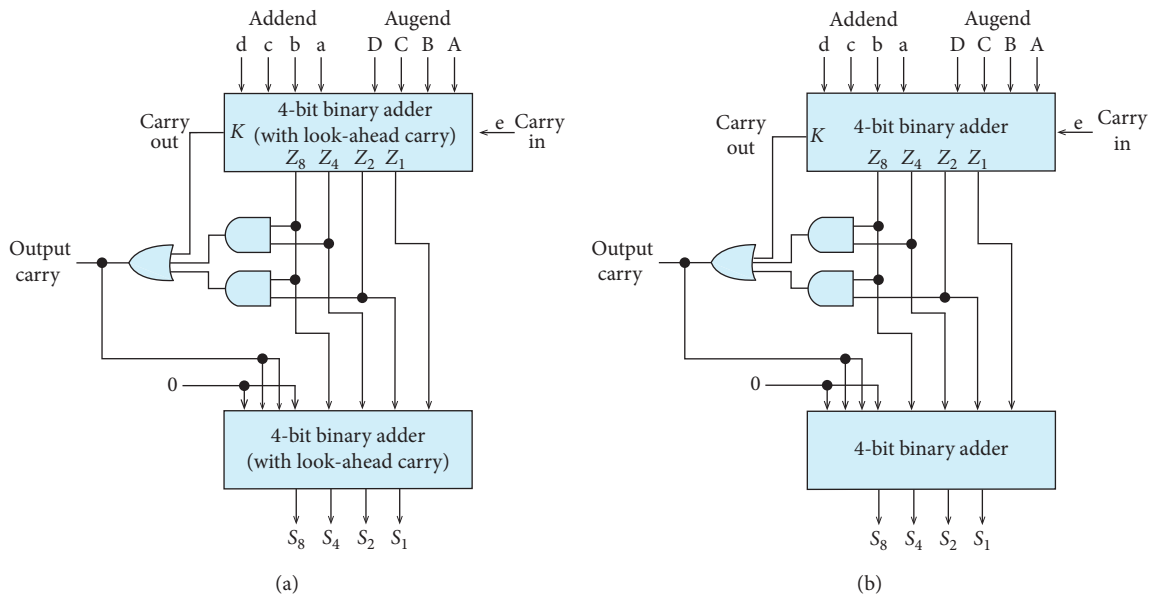

Figure 2: Block diagrams of two versions of a BCD adder. The BCD adder on the left is with look-ahead carry generator and the one on the right is without look-ahead carry generator [29].

```
(1) Definition fa (a A e: bool) : (bool * bool) :=
(2)     (A * a+(A ⊕ a) * e,A ⊕ a ⊕ e).
(3) Definition FA (f: bool ⟶ bool ⟶ bool ⟶ (bool * bool)) (a A e: bool)
(4) : (bool * bool) := fa a A e.
(5) Definition FA4bit (D C B A d c b a e: bool)
(6) : (bool * bool * bool * bool * bool) :=
(7) let ss1 := snd(FA fa a A e) in
(8) let cc1 := fst(FA fa a A e) in
(9) let ss2 := snd(FA fa b B cc1) in
(10) let cc2 := fst(FA fa b B cc1) in
(11) let ss3 := snd(FA fa c C cc2) in
(12) let cc3 := fst(FA fa c C cc2) in
(13) let ss4 := snd(FA fa d D cc3) in
(14) let cc4 := fst(FA fa d D cc3) in
(15) (cc4,ss4,ss3,ss2,ss1).
```

Listing 19: Formal definition of 4-bit binary parallel adder (without look-ahead carry).

```
(1) Definition BCDadder (D C B A d c b a e:bool) : circuit :=
(2)     let t1 := FA4bit D C B A d c b a e in
(3)     let outcarry := FST t1 + (SND t1) * (TRD t1) + (SND t1) * (FRT t1) in
(4)     let t2 :=
(5)         FA4bit (SND t1) (TRD t1) (FRT t1) (FFT t1) false outcarry outcarry false e in
(6)     [outcarry; SND t2; TRD t2; FRT t2; FFT t2].
```

Listing 20: Formal definition of 4-bit BCD adder (without look-ahead carry).

carry generator. Formal definition of a 4-bit binary parallel adder has been listed in Listing 21. The definitions `C2–C5` (lines 1–7) calculate the four output carries of the four full adders in the binary adder. Unlike the adder `FA4bit`, every internal output carry is independent of the output carry of the previous full adder. This allows every full adder to produce output sum simultaneously without waiting for the output carry from the previous adder. The definitions `S1–S4`

calculate the four sum bits in terms of the look-ahead carry bits C2–C5 and initial carry e. Finally, the 4-bit binary parallel adder with look-ahead carry generator is defined as a function FA4bitLA (lines 17–20). As in BCDadder, two instances of binary parallel adder FA4bitLA are combined together with the additional circuitry for output carry to form a BCD adder. The formal definition of 4-bit BCD adder with look-ahead carry generator has been shown in Listing 22. Readers are advised to refer to books [29, 36] for further details about BCD adder and look-ahead carry generator.

The formal definitions BCDadderLA and BCDadder model 4-bit BCD adders with and without look-ahead carries, respectively. Their functional equivalence is stated in theorem check_equiv_BCD_adder in Listing 23. The theorem states that both adders produce equal outputs for all equal inputs. This theorem is proven using case analysis on the input variables. The formal proof of Theorem check_equiv_BCD_adder demonstrates that our framework can effectively be used to check equivalence of combinational circuits other than memory circuits.

6.3. Proof of Reversibility Property. To demonstrate that our formal framework can also be used to verify properties other than equivalence, we formally verify reversibility of a simple circuit description. To this end, we prove that the circuit Circuit defined in Listing 24 is reversible. The theorem reversible_circuit states that, for all different (2-bit) inputs, the outputs are different. Additionally, this theorem also states that the numbers of inputs and outputs are the same. In this theorem, the lists [w; x] and [y; z] on line 4 in Listing 24 model the inputs.

7. Evaluation and Discussion

Automated tools, such as model checkers, sometimes stuck due to memory or state explosion problems [38] and never return. To evaluate the performance of Coq tool running over our framework, we tested the Coq proof checker to check proof scripts of functional equivalence of Boolean functions with multiple Boolean variables. The results in Figure 3 show that the Coq proof checker takes around 12 seconds to check proof scripts for functional equivalence with functions up to 45 variables (over a billion input cases).

The formal model developed in Coq provides a formal foundation for defining and reasoning about Boolean functions and logic circuits using the calculus of constructions behind the Coq theorem prover. The following is a list of the major advantages of our formal model:

(i) Boolean function definitions or models of logic circuits can be automatically checked (for type errors) by the Coq type checker.

(ii) The correctness of the mathematical manipulation used in the analysis of Boolean functions can be checked automatically.

(iii) Properties, such as principle of duality, functional equivalence, and reversibility, of Boolean algebra and (models of) logic circuits can be proven interactively using the Coq theorem prover.

(iv) All the proofs carried out can be mechanically checked by the Coq proof checker using computer.

Formal verification using proof assistant is very tedious and requires expertise; however, many researchers have recently used proof assistants to build formal languages and frameworks for hardware verification [49–53]. Proof assistants include benefits of both manual and automated theorem provers and are more powerful and expressive [50]. Our framework allows defining Boolean functions and combinational circuits in a natural style similar to that used in many popular textbooks [29, 36] and most of the proofs can be carried out easily by applying Coq's destruct tactic.

8. Related Work

The most common approach to reliable digital circuit design is to test designs using simulators such as VCS [54] and Icarus [55] by providing all possible inputs. While simulation-based verification can show presence of errors, it fails to guarantee their absence [56]. Approaches based on formal verification [57], such as *model checking* [32, 58, 59] and *theorem proving* [49, 52, 53, 60, 61], are more popular in the literature. Readers are recommended to refer to [62] for a detailed comparison of simulation and formal method-based approaches. Tools based on model checking can be used to check equivalence of two functions (models); however, they are constrained by the popular *state explosion* [37, 38] problem. There is a body of research works in the literature on formal verification of software systems [63–65]; however, literature review of hardware verification and simulation tools for checking Boolean functions equivalence is included in this section.

8.1. Formal Hardware Verification. As mentioned earlier, tools and techniques based on formal methods can be used to prove absence of faults in hardware components. Osman et al. [51] defined a formal framework in the higher-order logic of HOL theorem prover for proving reliability property of combinational circuits. While their work is mainly tailored towards checking reliability property, our framework facilitates functional equivalence checking in a stronger and expressive logic calculus of inductive constructions of Coq theorem prover [50]. Kabat et al. [66] advocate the use of automated theorem provers for the synthesis of combinational logic. They used demodulation as the rewriting logic to simplify canonical circuit structures. Automated theorem provers are more popular in industry; however, they are not as powerful and expressive as proof assistants. Proof assistants, on the other hand, combine the benefits of automated and manual theorem provers and are currently investigated and encouraged for hardware verification [50, 51]. The work in [50] highlights the effectiveness and power of interactive theorem prover Coq in hardware verification which further supports our framework embedded in Coq for logic circuit verification. While their work is more focused towards synthesis based on Coq's code

```
(1) Definition C2(D C B A d c b a e: bool) := A ∗ a+(A ⊕ a) ∗ e.
(2) Definition C3(D C B A d c b a e: bool) :=
(3)    B ∗ b + (B ⊕ b) ∗ (A ∗ a) + (B ⊕ b) ∗ (A ⊕ a) ∗ e.
(4) Definition C4(D C B A d c b a e: bool) :=
(5)    C ∗ c + (C ⊕ c) ∗ (B ∗ b) + (C ⊕ c) ∗ (B ⊕ b) ∗ (A ∗ a) + (C ⊕ c) ∗ (B ⊕ b) ∗ (A ⊕ a)∗e.
(6) Definition C5(D C B A d c b a e: bool) :=
(7)    D ∗ d + (D ⊕ d) ∗ (C4 D C B A d c b a e).
(8) Definition S1(D C B A d c b a e: bool) := (A ⊕ a) ⊕ e.
(9) Definition S2(D C B A d c b a e: bool) :=
(10)   (B ⊕ b) ⊕ (C2 D C B A d c b a e).
(11) Definition S3(D C B A d c b a e: bool) :=
(12)   (C ⊕ c) ⊕ (C3 D C B A d c b a e).
(13) Definition S4(D C B A d c b a e: bool) :=
(14)   (D ⊕ d) ⊕ (C4 D C B A d c b a e).
(15) Definition FA4bitLA (D C B A d c b a e: bool)
(16) : (bool ∗ bool ∗ bool ∗ bool ∗ bool) :=
(17)   (C5 D C B A d c b a e,S4 D C B A d c b a e,S3 D C B A d c b a e,
(18)   S2 D C B A d c b a e, S1 D C B A d c b a e).
```

Listing 21: Formal definition of 4-bit binary parallel adder (with look-ahead carry).

```
(1) Definition BCDadderLA (D C B A d c b a e:bool) : circuit :=
(2)   let t1 := FA4bitLA D C B A d c b a e in
(3)   let outcarry := FST t1 + (SND t1) ∗ (TRD t1) + (SND t1) ∗ (FRT t1) in
(4)   let t2 :=
(5)     FA4bitLA (SND t1) (TRD t1) (FRT t1) (FFT t1) false outcarry outcarry false e in
(6)   [outcarry; SND t2; TRD t2; FRT t2; FFT t2].
```

Listing 22: Formal definition of 4-bit BCD adder (with look-ahead carry).

```
(1) Theorem check_equiv_BCD_adder: forall D C B A d c b a e,
(2)   BCDadder D C B A d c b a e = BCDadderLA D C B A d c b a e.
```

Listing 23: Proof of equivalence of BCD adders (with and without look-ahead carry).

```
(1) Definition Circuit(w x: bool) : circuit := [¬w; ¬x].
(2) Theorem reversible_circuit: for all w x y z,
(3)   [w; x] < > [y; z] ⟶
(4)     (Circuit w x) < > (Circuit y z)
(5)   ∧ length [w; x] = length (Circuit w x)
(6)   ∧ length [y; z] = length (Circuit y z).
```

Listing 24: Proof of reversibility.

extraction feature, our work embeds Boolean algebra as a gate-level description language for circuit's description. Our gate-level description language provides a general framework for describing combinational circuits. Furthermore, we build metatheory for expressing and proving duality principle and carry out proofs of basic theorems of Boolean algebra.

Meredith et al. [67] defined executable semantics for Verilog by embedding it in the tool Maude [68] with rewriting logic as the underlying logic. Inspired from [67], Wilayat et al. [49, 69] introduced a formal language, VeriFormal: a hardware description language with mathematical foundation. VeriFormal is a formal replica of description language Verilog deeply embedded in Isabelle/HOL theorem prover. VeriFormal and the formal simulator available with it were used later to prove functional equivalence of multiple logic circuits [70]. Braibant et al. [71] defined Fe-Si by deeply embedding the simplified version of functional

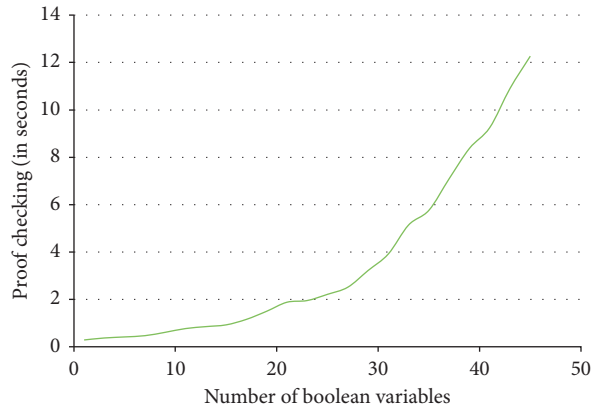

Figure 3: Performance evaluation.

hardware description language Bluespec in theorem prover Coq. Building upon the concepts behind proof-carrying code, Love et al. [72] implemented a framework by formalizing a synthesizable subset of Verilog in proof assistant Coq. Some researchers [52, 60] targeted programmable logic controllers by formalizing the semantics of the programming languages used in the controllers, while others specified circuits as operations on bit-vectors. Coquet [53] is a high-level specification of circuits using deep embedding; however, this level of abstraction is achieved through advanced types, such as parametric types, which makes the formal definitions of circuits short but with increased complexity. All these research contributions add formal verification at higher level of abstraction, while our work analyses the circuits at gate level. Formal verification at higher levels, such as register-transfer level, is equally important but the main focus of this paper is to target circuits at gate level. Furthermore, in addition to reasoning about digital circuits, our Coq framework can effectively be used to reason about Boolean algebra as well.

8.2. Boolean Equivalence Checking Tools. There are a number of other tools developed specifically for Boolean functions manipulation. Ronjom et al. [23] developed an online database of Boolean functions. Their tool can be used to check different properties of a Boolean function and convert between different representations. The WolframAlpha computational engine [21] translates logical function as input to a truth table and different minimal forms. Moreover, the tool generates a Venn diagram and logic circuit for the input function. The WolframAlpha engine has been included as Boolean algebra calculator by the company TutorVista. Another tool [22] was developed to minimize Boolean functions using Karnaugh maps [26]. The function can be entered as a sequence of notations or as a truth table (up to six variables).

Among the most recent tools is 32x8 [25] which has been built for logic circuit simplification. It accepts a function (up to eight variables) in the form of a truth table and returns a Karnaugh map, Boolean function (as sum of product or product of sums), truth table, and logic circuit for the input. Lean and Marxel developed a solver, QMSolver [24], based on Quine-McCluskey algorithm for simplification of Boolean functions. The solver gets number of minterm indices (separated by spaces) and returns a simplified function. All these tools manipulate circuits at gate level; however, none of them have formal foundation and hence the manipulation of circuit designs cannot be proven correct.

9. Conclusions

When (model of) a logic circuit is transformed or mathematically manipulated (most often for optimization purposes), it must be guaranteed that the transformation does not alter the desired behaviour of the circuit. In this paper, a formal framework for describing and verifying Boolean functions and logic circuits at gate level was defined in the Coq theorem prover. To demonstrate the significance of the framework, basic theorems of the Boolean algebra and the duality principle were proven. Furthermore, multiple basic hardware components were described in the formal notations and functional equivalence and reversibility properties were verified. Our formal developments can be used to describe other logic components used in critical systems and can be formally proven correct using the Coq theorem prover tool.

As a future work, we plan to build a translator to automatically translate circuit designs as Boolean functions in a natural style to the formalized (in Coq) Boolean algebra. Furthermore, to complete a formal electronic design flow, we intend to build a logic synthesis tool for translation from formal register-transfer level representation of the circuit (e.g., in VeriFormal, [49, 69]) to gate-level representation as Boolean functions.

Abbreviations

ROM: Read-only memory
RTL: Register-Transfer Logic
ITP: Interactive theorem proving
HOL: Higher Order Logic
ASCII: American Standard Code for Information Interchange
HDL: Hardware description language
BCD: Binary-coded decimal
S: K-map simplification operation
$\approx$: Function equivalence relation.

Data Availability

The Coq script data used to support the findings of this study have been deposited in the GitHub repository at https://github.com/wilstef/booleanalgebra.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this article.

Acknowledgments

The authors would like to extend their sincere appreciation to the Deanship of Scientific Research at King Saud University, Saudi Arabia, for partially funding this

research through the Research Group Project no. RGP-214.

References

[1] F. Al-Turjman, "A novel approach for drones positioning in mission critical applications," *Transactions on Emerging Telecommunications Technologies*, p. e3603, 2019.

[2] F. Al-Turjman and S. Alturjman, "5G/IoT-enabled UAVs for multimedia delivery in industry-oriented applications," *Multimedia Tools and Applications*, vol. 78, pp. 1–22, 2018.

[3] F. Al-Turjman, M. Abujubbeh, A. Malekloo, and L. Mostarda, "UAVs assessment in software-defined IoT networks: an overview," *Computer Communications*, vol. 150, pp. 519–536, 2020.

[4] V. Arumugham, *Vaccine Safety: Learning from the Boeing 737 MAX Disasters*, CERN European Organization for Nuclear Research, Geneva, Switzerland, 2019.

[5] P. Robison, *Boeing's 737 Max Software Outsourced to USD9-An-Hour Engineers*, The Guardian, Bloomberg, London, UK, 2019.

[6] D. P. Shepard, J. A. Bhatti, and T. E. Humphreys, "Drone hack: spoofing attack demonstration on a civilian unmanned aerial vehicle," *GPS World*, vol. 23, no. 8, pp. 30–33, 2012.

[7] S. M. Giray, "Anatomy of unmanned aerial vehicle hijacking with signal spoofing," in *Proceedings of the 2013 6th International Conference on Recent Advances in Space Technologies (RAST)*, pp. 795–800, IEEE, Istanbul, Turkey, June 2013.

[8] D. Davidson, H. Wu, R. Jellinek, V. Singh, and T. Ristenpart, "Controlling UAVs with sensor input spoofing attacks," in *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, USA, August 2016.

[9] K. Hartmann and K. Giles, "UAV exploitation: a new domain for cyber power," in *Proceedings of the 2016 8th International Conference on Cyber Conflict (CyCon)*, pp. 205–221, IEEE, Tallinn, Estonia, May-June 2016.

[10] S. Center, *Drone Wars–The Yemen Review*, Sana'a Center for Strategic Studies, Sana'a, Yemen, June 2019.

[11] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, pp. 8–22, 1990.

[12] E. Adler and J. B. Jeannin, "Formal verification of collision avoidance for turning maneuvers in UAVs," in *Proceedings of the AIAA Aviation 2019 Forum*, p. 2845, Dallas, Texas, USA, June 2019.

[13] M. Webster, M. Fisher, N. Cameron, and M. Jump, "Formal methods for the certification of autonomous unmanned aircraft systems," in *Proceedings of the International Conference on Computer Safety, Reliability, and Security*, pp. 228–242, Springer, Naples, Italy, September 2011.

[14] A. M. Madni, M. W. Sievers, J. Humann, E. Ordoukhanian, B. Boehm, and S. Lucero, "Formal methods in resilient systems design: application to multi-UAV system-of-systems control," in *Disciplinary Convergence in Systems Engineering Research*, pp. 407–418, Springer, Berlin, Germany, 2018.

[15] S. Jo, M. Ikram, I. Jung, W. Ryu, and J. Kim, "Power-driven image compression in wireless sensor networks," *Mobile and Wireless*, vol. 42, pp. 28–33, 2013.

[16] S. K. Jo, M. Ikram, I. Jung, W. Ryu, and J. Kim, "Power efficient clustering for wireless multimedia sensor network," *International Journal of Distributed Sensor Networks*, vol. 10, Article ID 148595, 2014.

[17] M. A. H. Chowdhury, M. Ikram, and K. H. Kim, "Secure and survivable group communication over MANET using CRTDH based on a virtual subnet model," in *Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference*, pp. 638–643, IEEE, Yilan, Taiwan, December 2008.

[18] H. Redwan, M. A. H. Chowdhury, M. Ikram, and K. H. Kim, "Survey of indexing schemes for information retrieval on flash memory based wireless sensor networks," in *Proceedings of the 2009 Conference on Information Science, Technology and Applications*, pp. 14–21, ACM, Kuwait, March 2009.

[19] G. Boole, *Investigation of the Laws of Thought*, Dover, Downers Grove, IL, USA, 1854.

[20] C. E. Shannon, "A symbolic analysis of relay and switching circuits," *Electrical Engineering*, vol. 57, pp. 713–723, 1938.

[21] WolframAlpha computational knowledge engine," February 2018, https://www.wolframalpha.com/.

[22] Online minimization of Boolean functions," February 2018, https://www.tma.main.jp/logic/index_en.html/.

[23] R. Sondre, A. Mohamed, Lars, and D. Eirik, "Online database of Boolean functions," February 2018, http://www.ii.uib.no/~mohamedaa/odbf/index.html.

[24] R. Lean, Kryle, and M. Marxel, "QMSolver," February 2018, http://agila.upm.edu.ph/~kmmolina/qms/index.html.

[25] Logic circuit simplification (SOP and POS)," February 2018, http://www.32x8.com/.

[26] 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, pp. 593–599, 1953.

[27] T. K. Jain, D. S. Kushwaha, and A. K. Misra, "Optimization of the Quine-Mccluskey Method for the Minimization of the Boolean Expressions," in *Proceedings of the ICAS 2008 Fourth International Conference on Autonomic and Autonomous Systems*, pp. 165–168, IEEE, Anchorage, AK, USA, September 2008.

[28] 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, pp. 1377–1394, 2002.

[29] M. M. Mano, *Digital Logic and Computer Design*, Pearson Education India, Delhi, India, 2017.

[30] O. Lhoták, "Program analysis using binary decision diagrams," vol. 68, A thesis submitted to McGill University, McGill University, Montreal, Canada, 2006.

[31] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using datalog with binary decision diagrams for program analysis," in *Proceedings of the Asian Symposium on Programming Languages and Systems*, pp. 97–118, Springer, Tsukuba, Japan, November 2005.

[32] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation1*, vol. 992, no. 98, pp. 142–170.

[33] B. L. SYNTHESIS, "ABC: a system for sequential synthesis and verification, release 70930," 2007.

[34] A. Mulhern, C. Fischer, and B. Liblit, "Tool support for proof engineering," *Electronic Notes in Theoretical Computer Science*, vol. 174, pp. 75–86, 2007.

[35] 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.

[36] M. M. Mano and M. D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL, VHDL, and System Verilog*, Pearson, Boston, MA, USA, 2018.

[37] A. Valmari, "The state explosion problem," *Lectures on Petri Nets I: Basic Models*, pp. 429–528, Springer, Berlin, Germany, 1998.

[38] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Modelchecking and the state explosion problem," in *Tools for Practical Software Verification*, Springer, pp. 1–30, 2012.

[39] J. L. Boulanger, *Industrial Use of Formal Methods: Formal Verification*, John Wiley & Sons, Hoboken, NJ, USA, 2013.

[40] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: verification of probabilistic real-time systems," in *Proceedings of the International Conference on Computer Aided Verification*, Springer, pp. 585–591, Snowbird, UT, USA, July 2011.

[41] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. M. de Sousa, "An overview of formal methods tools and techniques,"*Rigorous Software Development*, pp. 15–44, Springer, London,UK, 2011.

[42] D. Kroening and M. Tautschnig, "CBMC–C bounded model checker," in *Proceedongs of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 389–391, London, UK, October 2014.

[43] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, Springer Science & Business Media, 2013.

[44] B. Barras, S. Boutin, C. Cornes et al., *The Coq proof assistant reference manual: version 6.1*, Ph.D. thesis, Inria, 1997.

[45] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-rder Logic*, vol. 2283, Springer Science & Business Media, Berlin, Germany.

[46] W. Ahrendt, B. Beckert, R. Hähnle et al., *Integrating Automated and Interactive Theorem Proving*, Kluwer Academic Publisher, Amsterdam, Netherlands, 1998.

[47] B. C. Pierce, C. Casinghino, M. Gaboardi et al., "Software foundations," 2010, http://www.cis.upenn.edu/bcpierce/sf/current/index.html.

[48] 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, pp. 274–304, 1933.

[49] K. Wilayat, T. Alwen, and S. David, "VeriFormal: an executable formal model of a hardware description language. A systems approach to cyber security," in *Proceedings of the 2017 2nd Singapore Cyber Security R&D Conference SGCSC*, pp. 19–36, Singapore, February 2017.

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

[51] O. Hasan, J. Patel, and S. Tahar, "Formal reliability analysis of combinational circuits using theorem proving," *Journal of Applied Logic*, vol. 9, pp. 41–60, 2011.

[52] S. O. Biha, "A formal semantics of PLC programs in Coq," in *Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, pp. 118–127, Munich, Germany, July 2011.

[53] T. Braibant, "Coquet: a coq library for verifying hardware," in *Proceedings of the International Conference on Certified Programs and Proofs*, pp. 330–345, Springer, Kenting, Taiwan, December 2011.

[55] Icarus Verilog,," December 2016, http://www.icarus.com/eda/verilog/.

[56] G. J. Pace and J. He, "Formal reasoning with verilog HDL," in *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems*, Marstrand, Sweden, June 1998.

[57] S. Y. Huang and K. T. T. Cheng, Formal Equivalence Checkingand Design Debugging, vol. 12, Springer Science & Business Media, Berlin, Germany.

[58] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, London, UK, 1999.

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

[60] L. Xiaoa, M. Li, M. Gu, and J. Sun, "The modelling and verification of PLC program based on interactive theorem proving tool Coq," in *Proceedings of the International Conference on Computer Science and Information Technology (ICCSIT)*, Hong Kong, China, December 2012.

[61] L. Arditi and S. Antipolis, "Formal verification of microprocessors: a first experiment with the Coq proof assistant," Technical report, Research report, I3S, Université de Nice–Sophia Antipolis, Nice, France, 1996.

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

[63] 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.

[64] W. Khan, H. Ullah, A. Ahmad et al., "CrashSafe: a formal model for proving crash-safety of android applications," *Human-centric Computing and Information Sciences*, vol. 8, no. 1, p. 27, 2018.

[65] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan, "CookiExt: patching the browser against session hijacking attacks," *Journal of Computer Security*, vol. 23, no. 4, pp. 509–537, 2015.

[66] 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.

[67] P. Meredith, M. Katelman, J. Meseguer, and G. Roşu, "A formal executable semantics of Verilog," in *Proceedings of the 2010 8th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 179–188, IEEE, Grenoble, France, July 2010.

[68] 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, Bergen, Norway, August 2007.

[69] W. Khan, D. Sanan, Z. Hou, and L. Yang, "On embedding a hardware description language in Isabelle/HOL," *Design Automation for Embedded Systems*, vol. 23, no. 3-4, pp. 123–151, 2019.

[70] W. Khan, A. Basim, S. Noman, K. Abdul Moeed, and S. Ahtisham, "Formal verification of digital circuits using simulator with mathematical foundation," *Applied Mechanics and Materials*, vol. 892, pp. 134–142, 2019.

[71] T. Braibant and A. Chlipala, "Formal verification of hardwaresynthesis," in *Proceedings of the International Conference on Computer Aided Verification*, Springer, pp. 213–228, St. Petersburg, Russia, July 2013.

[72] 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.