Mathematical framework for checking equivalence of logic circuits *Technical Report*

Wilayat Khan[1,⋆], Muhammad Kamran[2], Farrukh Aslam Khan[3], Syed Rameez Naqvi[1], Habib Ullah[4], Abdelouahid Derhab[3]

[1] *Department of Electrical Engineering, COMSATS University Islamabad, Wah Campus, Pakistan; {wilayat,rameeznaqvi}@ciitwah.edu.pk*

[2] *Department of Computer Science, COMSATS University Islamabad, Wah Campus, Pakistan; muhammad.kamran@ciitwah.edu.pk*

[3] *Center of Excellence in Information Assurance, King Saud University, Riyadh, Saudi Arabia; {fakhan,abderhab}@ksu.edu.sa*

[4] *College of Computer Science and Engineering, University of Hail, Saudi Arabia; h.ullah@uoh.edu.sa*

[*] *Corresponding author*

Abstract

Digital circuits are designed using high level notations of hardware description languages (register-level) or lower level Boolean functions (gate-level). Description languages abstract away the design process and is easy to use, however, the description has to be transformed into gate-level representation for final fabrication. Gate-level representation using Boolean functions is an integral part of circuit design process and is used in classrooms to understand basic logic design. Bringing the mathematical tools and techniques into the circuit design paradigm at gate level, we define a mathematical model of the Boolean algebra in higher-order logic of theorem prover Coq. Using the Coq proof facility, we mathematically prove using computer that the algebra indeed is Boolean. The mathematical structure allows to define Boolean functions thereby providing a framework for specifying and proving correctness of digital logic circuits modelled as Boolean functions. To demonstrate the significance of the mathematical framework, a 32-bytes ROM memory chip is designed, simplified using K-map method and proved the simplification process does not alter the behaviour of the circuit.

Keywords: Boolean algebra; Coq; circuit design; equivalence checking; formal framework; formal verification; theorem prover

1. Introduction

The design process of an electronic circuit begins with its functional specification of the user requirements using truth table all the way to its physical layout on the circuit board. To ease the embedded system design process, the design of the intended circuit goes

Email address: `wilayat@ciitwah.edu.pk` (Wilayat Khan[1,⋆])

through a number of transformations: the initial design described at register-transfer-logic (RTL) level in a hardware description language (HDL) is transformed into gate level representation. HDLs are used to represent digital circuits at a high level which is easy for the designers to specify the behaviour of the circuit. Synthesis tools are used to translate the RTL representation into gate level representation often described as Boolean functions. The gate level representation as Boolean functions is commonly used in techniques and frameworks [1, 2, 3, 4] and in classrooms for the design and analysis of simple logic circuits [5].

While different computer tools [6, 7, 8, 9, 10] and mathematical techniques [11, 12] are available to manipulate Boolean functions, it must be ensured that such tools and techniques do not alter the intended behaviour of the (model of the) circuit. In other words, for each transformation, the design *after* the transformation must be checked equivalent to the design *before* the transformation. A common approach to check two circuits are functionally equivalent is to extensively simulate two circuits against many inputs and the results are compared. Using simulation based approach, combinational circuits can be fully verified in principle, however, the simulation time is exponential to the number of inputs. Another popular approach to equivalence checking is to use formal methods based on mathematical tools and techniques. Mathematical tools and techniques can be used, in addition to equivalence checking, to prove other properties for all possible inputs. The prominent difference between the simulation and formal methods based verification is that input vectors are used in the former while the later uses mathematical techniques to carry proofs without input vectors [13].

The most common form of formal verification is to use *model checking* [14], however, tools based on model checking may be restricted by the *state explosion problem* [15, 16]. A different formal approach is to use interactive theorem provers such as Coq [17] and Isabelle/HOL [18], to build formal model of the circuit and carry a proof of equivalence in the tool. Such computer tools are interactive: human assistance is required to guide the tools by providing them proof commands. In this paper, a formal framework is built in the Coq proof assistant to formally check equivalence of two electronic circuits. Tailored towards circuits in canonical compact form, the framework is described by formally proving that Boolean functions simplification (the K-map or Tabulation transformations) preserves the intended behaviour. In order to formally verify if two circuits (their symbolic representations as Boolean functions) are equivalent, a formal model of the Boolean algebra is defined in the formal tool Coq. Coq [17] is an interactive theorem prover that allows to build formal models of systems and reason about them. To check *correctness* of a circuit, its different representations (as Boolean functions) are encoded in the formal logic developed. The proof facility of the theorem prover Coq is used to carry out the proof of equivalence of the functions described in the logic. The major contributions of this paper are the following.

- A formal model of two-valued algebra is defined in the Coq proof assistant.

- The formalized algebra is proved in Coq to be Boolean.
- A 32x8 ROM chip is designed using the algebra and simplified using K-map method.
- A formal proof of correctness of the simplification is carried in Coq.

The rest of the paper is organized as the following: In the next section (Section 2), theorem proving using Coq, Boolean algebra, functions and circuit implementation using the algebra are described. Formal model of the Boolean algebra in Coq and its major postulates are described in Section 3. The basic theorems of Boolean algebra and their mechanical proofs are listed in Section 4. The design of a 32-bytes memory and its proof of correctness is carried in Section 5. A summary of the related work is given in the Section 7. The paper is concluded in Section 8.

2. Background

To mechanically reason about Boolean algebra, functions or circuit models implemented in the algebra, the algebra must be defined in the logic of a proof assistant. Such a formal definition of Boolean algebra is given in Section 3. To understand the formal definition, the algebra itself and the computer tool Coq used to formalize the algebra are discussed in this section.

2.1. Boolean algebra

Defined by 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 [20] postulates. Later on, Shannon [21] introduced a two-valued version of the algebra to represent properties of switching circuits. In a two-valued algebra, the set of values include elements 1 and 0 and rules for the two binary operations together with a unary complement operation are shown in Table 1.

Table 1: Truth tables for product (a), sum (b) and complement (c) operations

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

(a) Product

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

(b) Sum

x	x’
0	0
0	1

(c) Complement

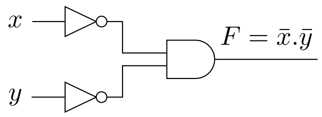

Figure 1: Implementation of Boolean function $F = \bar{x}.\bar{y}$ with gates

Using Boolean functions in the two-valued Boolean algebra, logic circuits can be modelled as shown in the Figure 1. The function $F = \bar{x}.\bar{y}$ models the circuit where the AND gate represents the product operation (Table 1.b) and the NOT gate represents the complement operation (Table 1.c). When logic circuits are represented with functions as in the Figure 1, they can be manipulated and reasoned about using tools and techniques developed for Boolean algebra.

2.2. Coq proof assistant

Coq [17] is a formal tool (interactive theorem prover) with calculus of construction as the underlying calculus. It is similar to other programming languages, however, in addition to programming, Coq can be used to reason about programs defined in the logic of Coq. To prove theorems about a system, a model of the system and property are defined in the logic of Coq and then a proof that (model of) the system satisfies the property is carried and checked in Coq. The selection of proof assistant Coq for formal development is a matter of taste and any other proof assistant may also be used.

To describe formal developments and proofs using proof assistant, a simple system of numbers is defined and reasoned about using tool Coq. To begin with, numbers are inductively defined as data type `nat` using the Coq keyword `Inductive` with two constructors for generating elements of the type `nat` (Figure 2). The definition **nat** states that `O` (for 0) is the first element of type `nat` and any number preceded by letter `S` (for successor) is also `nat`. The term `S (S (S O))`, for example is a number (3) in `nat`.

```
Inductive nat : Type :=
 | O : nat
 | S : nat → nat.
```

Figure 2: Data type nat

Next we define a function on the numbers just defined. The recursive function `add` in Figure 3 adds two numbers of type `nat`. Using pattern matching (with keyword `match`) on the first argument `n`, it returns the other if `n` is `O`. If `n` is of the form `S n'`, then it returns `S (add n' m)`, where `m` is the second argument of the function and `add n' m` is a recursive call to itself. After defining model (type `nat`) and a function over it, different properties can be stated and proved in Coq. A lemma, that `add m O = m` holds for any value of `m` is stated and proved in Figure 4. The lemma is proved using induction on the construction of `m`. During the proof process, the Coq tool is guided interactively by providing Coq commands *tactics* (lines 3–8).

```
Fixpoint add (n m : nat) : nat :=
 match n with
  | O ⇒ m
  | S n' ⇒ S (add n' m)
 end.
```

Figure 3: Function add

```
1 Lemma add_m_o: ∀ m, add m O = m.
2 Proof.
3  induction m.
4  simpl.
5  reflexivity.
6  simpl.
7  rewrite IHm.
8  reflexivity.
9 Qed.
```

Figure 4: Interactive proof in Coq

The first command `Proof` tells Coq to enter the proof mode. The command *induction m* uses induction on the argument m. As m of type nat can be constructed in two ways (two constructors on lines 2–3 in Figure 2), induction on m creates two sub-goals: one for each constructor. The first goal is simple and is closed by simplification (command simpl) followed by command reflexivity. After the proof for the first case (often referred to as the base case) is completed, an induction hypothesis IHm is automatically generated. Using the rewrite command, the hypothesis IHm is re-written in the goal which makes the left and right sides of the goal similar and is closed with reflexivity command. The final command Qed adds this proof to Coq repository for future reference. For further background in Coq, readers are recommended to refer to book Software Foundations [22] and Coq reference manual [17].

3. Formalizing Boolean Algebra

To formally reason about logic circuits described as Boolean function, the Boolean algebra is formalized in Coq theorem prover. Formalizing the algebra in a theorem prover enables one to reason mechanically about the algebra itself as well as the designs described in the algebra. An algebraic structure, defined over a set of elements B and two binary operations + and ., is Boolean algebra if Huntington six postulates [20] are satisfied. To define a Boolean algebra, one must first define elements of a set B, some rules for the two operations + and . on the elements in set B and proof that the Huntington postulates are satisfied.

3.1. Two-Valued Boolean Algebra

To describe digital circuits, with two logical values, a two-valued Boolean algebra is defined. To begin with, the set B is modelled by defining a type bool using the Coq keyword Inductive as shown in the Figure 5. As the algebra is two-valued, the type bool has only two members true (for logic 1) and false (for logic 0).

```
Inductive bool: Type := true | false.
```

Figure 5: Data type bool

```
1 Definition sum (x y: bool) : bool :=
2   match x, y with
3  | false, false ⇒ false
4  | _, _ ⇒ true
5   end.
```

Figure 6: Operation sum

Next is to define the rules of two operations, + and ., on the two elements of the type bool. The rules for the first operation are given in the definition sum as shown in the Figure 6. The function sum gets two arguments (line 1) of type bool and returns false (line 3) if values of both arguments are false, otherwise, it returns true (line 4). The keyword match (line 2) is used to pattern-match on the two arguments of the function sum. The second operation . is defined as function prod (Figure 7). The function prod gets two arguments of type bool as input (line 1) and returns true (line 3) when values of both inputs are true, otherwise, it returns false (line 4).

```
Definition prod (x y: bool) : bool :=
  match x, y with
 | true, true ⇒ true
 | _, _ ⇒ false
  end.
```

Figure 7: Operation product

```
Definition not (x: bool) : bool :=
   match x with
 | false ⇒ true
 | true ⇒ false
   end.
```

Figure 8: Operation complement

An operation complement is also defined over the type `bool` which simply returns the other value if one is given. The complement operation is defined as a single argument function `not` (Figure 8) which returns `true` when input is `false` and vice versa. To simplify expressions, Coq allows to define symbolic notations using the keyword `Notation`: the symbols `+`, `.` and ¬ are used as shorthand for the operation `sum`, `prod` and `not`, respectively, with operation ¬ has the highest and `+` has the lowest precedence.

3.2. Huntington Postulates

The final requirement of Boolean algebra is to show that the Huntington postulates are satisfied. The six Huntington postulates are formalized and proved in Coq.

Identity element. The identity element with respect to operation + is 0. That is, for any variable x of type *bool*, $x + 0 = 0 + x = x$. Similarly, the identity element with respect to operation `.` is `1`. For any variable x of type *bool*, $x.1 = 1.x = x$. To formally prove 0 and 1 are the identity elements with respect to operation + and ., respectively, four lemmas are proved (Figure 9).

```
Lemma identity_sum: ∀ x, x + false = x.
Lemma identity_sum′: ∀ x, false + x = x.
Lemma identity_prod: ∀ x, x ∗ true = x.
Lemma identity_prod′: ∀ x, true ∗ x = x.
```

Figure 9: Identity element lemmas

Commutative law. The operations + and . are commutative. For any two variables x and y of type *bool*, $x + y = y + x$ and $x.y = y.x$. In Coq, the commutative properties over + and . are defined in lemmas `comm_sum` and `comm_prod`, respectively, as shown in Figure 10.

```
Lemma comm_sum: ∀ x y, x + y = y + x.
Lemma comm_prod: ∀ x y, x ∗ y = y ∗ x.
```

Figure 10: Commutative lemmas

Distributive law. The distributive . over + law states that for any three variables x, y and z, $x.(y + z) = (x.y) + (x.z)$. Equivalently, the distributive + over . law states that $x + (y.z) = (x + y).(x + z)$. These two distributive laws are defined as lemmas `dist_over_sum` and `dist_over_prod`, respectively, in Figure 11.

```
Lemma dist_over_sum: ∀ x y z, x ∗ (y + z) = x ∗ y + x ∗ z.
Lemma dist_over_prod: ∀ x y z, x + y ∗ z = (x + y) ∗ (x + z).
```

Figure 11: Distributive lemmas

Complement law. This law states that for every variable x of type *bool*, there exists x' of type *bool* such that $x + x' = 1$ and $x.x' = 0$. Both of the forms of complement law are defined as lemmas `compl_sum` and `compl_prod` in Figure 12.

```
Lemma compl_sum: ∀ x, x + ¬x = true.
Lemma compl_prod: ∀ x, x ∗ ¬x = false.
```

Figure 12: Complement lemmas

Closure. The closure postulate states that the values of the results of operations + and . are in the set B. The type of these operations as defined as Coq functions in Figure 6 and 7 is `bool`. The Coq type checker ensures all the expressions in the algebra are *well-formed* and can only return values of type `bool` satisfying the closure property.

The proof of these Huntington postulates (except the closure property) are carried through case analysis on the variables used. All the Coq definitions and proof scripts are available at the link `https://github.com/wilstef/booleanalgebra`.

4. Proof of Boolean Theorems

To ease reasoning about Boolean structures, a set of theorems is used which provides, in addition to postulates, basic relationships in the Boolean algebra. To prove these theorems, two approaches are adopted: using postulates or case analysis. The former is easy when used in pen-and-paper method while the later is easy when a theorem prover is used. Proofs of the basic theorems carried using case analysis are extremely simple, in particular, when the algebra is two-valued. Even though, proof based on the postulates is tedious using theorem prover as it requires careful selection and order of postulate applied at each step, the proof script can be checked mechanically. In either approach, the proofs carried in a theorem prover are more organized and their correctness can be checked using computer.

4.1. Basic theorems

All the basic theorems of Boolean algebra are stated as Coq lemmas as shown in the Figure 13. These lemmas are proved by applying, using the Coq `rewrite` tactic, the six Huntington postulates proved in the Section 3.

```
Lemma idempotent_sum: ∀ x, x + x = x.
Lemma idempotent_prod: ∀ x, x ∗ x = x.
Lemma identity_sum_1: ∀ x, x + true = true.
Lemma identity_prod_0: ∀ x, x ∗ false = false.
Lemma assoc_sum: ∀ x y z, (x + y) + z = x + (y + z).
Lemma assoc_prod: ∀ x y z, (x ∗ y) ∗ z = x ∗ (y ∗ z).
Lemma DeMorg_sum: ∀ x y, ¬(x + y) = ¬x ∗ ¬y.
Lemma DeMorg_prod: ∀ x y, ¬(x ∗ y) = ¬x + ¬y.
Lemma absorption_sum: ∀ x y, x + x∗y = x.
Lemma abroption_prod: ∀ x y, x ∗ (x + y) = x.
Lemma simpl_sum: ∀ x y, x + ¬x ∗ y = x + y.
Lemma simpl_prod: ∀ x y, x ∗ (¬x + y) = x ∗ y.
Lemma adjacency_sum: ∀ x y, x∗y + x∗¬y = x.
Lemma adjacency_prod: ∀ x y, (x + y) ∗ (x + ¬y) = x.
Lemma involution: ∀ x, ¬¬x = x.
Lemma consensus:
   ∀ x y z, x∗y + ¬x∗z + y∗z = x∗y + ¬x∗z.
```

Figure 13: Basic theorems of Boolean algebra

To highlight the significance of formalizing Boolean algebra in a theorem prover, we show, with an example, that the proofs carried using case analysis in theorem prover are short and easy as compared to using postulates. The *idempotent law* states that for any variable *x*, *x * x = x*. This theorem is proved using both approaches, postulates and case analysis, as shown in the Figures 14[1] and 15, respectively.

```
Lemma idempotent_prod: ∀ x, x ∗ x = x.
Proof.
  intro x.
  pattern (x ∗ x) at 1.
  rewrite ← identity_sum. (∗by postulate in Figure 9∗)
  rewrite ← compl_prod with (x:=x).
                          (∗by postulate in Figure 12∗)
  rewrite ← dist_over_sum. (∗by postulate in Figure 11∗)
  rewrite compl_sum. (∗by postulate in Figure 12∗)
  rewrite identity_prod. (∗by postulate in Figure 9∗)
  auto.
Qed.
```

Figure 14: Proof using postulates

```
Lemma idempotent_prod: ∀ x, x ∗ x = x.
Proof.
  destruct x; simpl; auto.
Qed.
```

Figure 15: Proof using case analysis

The Coq command (tactic) `destruct` breaks the goal into multiple sub-goals, one for each value of the variable `x`. In two-valued Boolean algebra, two sub-goals are created for each Boolean variable. The semicolon `;` combines two tactics and applies the second tactic to each sub-goal created by the first tactic (`destruct` in this case). The tactic `simpl` simplifies the function `sum` against each value of the variable `x` (`true` and `false`). The tactic `auto` automatically solves simple goals. In case of equal Boolean functions, the result is always `true` = `true` or `false` = `false` after tactics `destruct` and `simpl` are used in sequence. If two functions are equal, they must pro-

[1] The text inside (* and *) represent comments.

duce the same value for the same input. In other words, when the tactics `destruct` and `simpl` are applied, they both must result either `true` or `false` on both sides, which is closed by the tactic `auto`.

All the basic theorems of Boolean algebra [5] of the form `lhs` = `rhs` can be solved using three tactics `destruct`, `simpl` and `auto`. If these tactics are combined together using the symbol `;`, all the basic theorems can be proved in just one line script as shown in the Figure 15. If there are more than one variable, the case analysis can still be performed in one occurrence of the tactic `destruct`, with variables separated by `,`. The theorem for *Associative Law* with variable `x, y, z`, for example, can be proved using the script `destruct x, y, z; simpl; auto.`.

4.2. Principle of duality

The proof of a property of Boolean algebra, called the *duality principle*, is not listed in the Figure 13. The duality principle states that expressions derived from postulates by interchanging the operators (+ and .) and identity elements (0 and 1) are valid. All the postulates, except the *closure*, have two parts: one part is for the operation + (sum) and the other is for . (product). One part of the postulate can be derived from the other if the operators and identity elements are interchanged. The duality principle is significant in proving the theorems using postulates. In particular, when the proof of one part of a theorem is given, the other can easily be carried following the principle of duality. This derivation carried is believed to be valid according to the principle of duality in literature and textbooks[5], however, no formal proof has been provided. In this section, a formal proof of duality principle is provided.

As the derivation requires operating on the Boolean expressions, the principle of duality can not 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 the Figure 16. The first two constructors (line 2 and 3) correspond to the two boolean values (identity elements `true` and `false`) and the constructor `bexp` (on line 4) models boolean variables. The rest of constructors represent operations `sum`, `prod` and `not`, respectively.

```
1  Inductive exp : Type :=
2    trueexp: exp
3  | falseexp: exp
4  | bexp: exp → exp
5  | sumexp: exp → exp → exp
6  | prodexp: exp → exp → exp
7  | compexp: exp → exp.
```

Figure 16: Data type exp

Next, a function `changeident` (Figure 17) is defined to interchange identity elements in the Boolean expression. It gets an expression and replaces the identity element `true` with `false` and vice versa and leaves the operators and variables unchanged. The keyword `Fixpoint` is used to define recursive functions. Another operation, interchanging operators, is defined as a recursive function `changeop` in the Figure 18. The function `changeop` gets an expression and replaces the operator `sum` with `prod` and vice versa.

```
Fixpoint changeident (e : exp) : exp :=
 match e with
  | trueexp ⇒ falseexp
  | falseexp ⇒ trueexp
  | bexp e′ ⇒ e
  | sumexp e1 e2 ⇒
    sumexp (changeident e1) (changeident e2)
  | prodexp e1 e2 ⇒
    prodexp (changeident e1) (changeident e2)
  | compexp e′ ⇒ compexp (changeident e′)
 end.
```

Figure 17: Function for interchanging identity elements

```
Fixpoint changeop (e : exp) : exp :=
 match e with
  | sumexp e1 e2 ⇒ prodexp e1 e2
  | prodexp e1 e2 ⇒ sumexp e1 e2
  | _ ⇒ e
 end.
```

Figure 18: Function for interchanging operators

After defining Boolean expressions and the interchange operations, the duality principle can now be stated as a lemma as shown in the Figure 19. 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 ($x.y = y.x$) is proved (Figure 20) by applying the duality property.

```
Lemma daulity: ∀ lhs rhs,
  lhs = rhs →
  changeop (changeident lhs) = changeop (changeident rhs).
```

Figure 19: The duality property

```
Lemma daulity_check: ∀ x y,
  sumexp (bexp x) (bexp y) = sumexp (bexp y) (bexp x) →
   prodexp (bexp x) (bexp y) = prodexp (bexp y) (bexp x).
```

Figure 20: Using the duality property

The formalism given in the Figures 16, 17 and 18 are built to prove the duality principle, though, the principle itself is not used to prove basic theorems given in the Figure 13. The reason is that it would require further developments to evaluate the expressions by defining a function operating on the type `exp`. The concept of duality can directly be applied in proofs in the first formal setting (without the type `exp`) which is valid as it is accepted by the Coq type checker.

```
Lemma absorption_sum: ∀ x y, x + x∗y = x.
 Proof.
  intros.
  pattern x at 1.
  rewrite ← identity_prod.
  rewrite ← dist_over_sum.
  rewrite ← comm_sum.
  rewrite identity_sum_1.
  rewrite identity_prod.
  auto.
 Qed.
```

Figure 21: Proof of absorption law (sum)

```
1  Lemma abroption_prod: ∀ x y, x*(x + y) = x.
2   Proof.
3    intros.
4    pattern x at 1.
5    rewrite ← identity_sum.
6    rewrite ← dist_over_prod.
7    rewrite ← comm_prod.
8    rewrite identity_prod_0 with (x:= y).
9    rewrite identity_sum.
10   auto.
11  Qed.
```

Figure 22: Proof of absorption law (product)

To demonstrate this fact, proofs of two parts of absorption theorem are given in Figures 21 and 22. The proof step in one part corresponds to the dual of the proof step in the other part. The first two tactics in the proofs of both parts are the common: `intros` introduces the variables `x` and `y` to the hypothesis and the `pattern` chooses the first (from left) occurrence of the variable `x` in the conclusion for rewrite (by the `rewrite` on next line). The postulates/theorems applied on the rest of lines (lines 5–9) are exactly the dual of each other. That is if a part of postulate/theorem is applied in the proof of a theorem, exactly the dual of the postulate/theorem is applied in the same step of proof of the dual of the theorem.

5. Verifying Functional Equivalence

In previous sections (Section 3 and 4), a formal model of algebra was developed and it was proved that the algebra indeed is a two-valued Boolean algebra. Such a formalism provide a framework to specify logic circuits using Boolean functions and reason about them using the proof facility of Coq. In this section, a 32×8 ROM memory chip is specified in the formalized Boolean algebra as Boolean functions, simplified using Karnuagh-map method and proved that the original design of the chip is functionally equivalent to its simplified version. In other words, it is proved that the simplification process does not alter the behaviour of the circuit.

To begin with, the 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 2. The letters E−A represent the five-bit input (address lines) and O_7–O_0 shows the eight-bit output (data lines) of the memory. The table specifies the input and output relation: a five-bit input selecting any of the first sixteen locations, the chip returns a letter of the content *Trusted designs.*.

Table 2: Contents stored in a 32 × 8 ROM

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	’.’

```
Definition O6(A B C D E : bool) : bool :=
¬A∗¬B∗¬C∗¬D∗¬E + ¬A∗¬B∗¬C∗¬D∗E +
¬A∗¬B∗¬C∗D∗¬E + ¬A∗¬B∗¬C∗D∗E +
¬A∗¬B∗C∗¬D∗¬E + ¬A∗¬B∗C∗¬D∗E +
¬A∗¬B∗C∗D∗¬E + ¬A∗B∗¬C∗¬D∗¬E +
¬A∗B∗¬C∗¬D∗E + ¬A∗B∗¬C∗D∗¬E +
¬A∗B∗¬C∗D∗E + ¬A∗B∗C∗¬D∗¬E +
¬A∗B∗C∗¬D∗E + ¬A∗B∗C∗D∗¬E +
A∗¬B∗¬C∗¬D∗¬E + A∗¬B∗¬C∗¬D∗E +
A∗¬B∗¬C∗D∗¬E + A∗¬B∗¬C∗D∗E +
A∗¬B∗C∗¬D∗¬E + A∗¬B∗C∗¬D∗E +
A∗¬B∗C∗D∗¬E + A∗B∗¬C∗¬D∗¬E +
A∗B∗¬C∗¬D∗E + A∗B∗¬C∗D∗¬E +
A∗B∗¬C∗D∗E + A∗B∗C∗¬D∗¬E +
A∗B∗C∗¬D∗E + A∗B∗C∗D∗¬E.
```

Figure 23: Boolean function for output O_6

```
Definition O4(A B C D E:bool) : bool:=
¬A∗¬B∗¬C∗¬D∗¬E + ¬A∗¬B∗¬C∗¬D∗E +
¬A∗¬B∗¬C∗D∗¬E + ¬A∗¬B∗¬C∗D∗E +
¬A∗¬B∗C∗¬D∗¬E + ¬A∗B∗¬C∗D∗¬E +
¬A∗B∗C∗D∗¬E + A∗¬B∗¬C∗¬D∗¬E +
A∗¬B∗¬C∗¬D∗E + A∗¬B∗¬C∗D∗¬E +
A∗¬B∗¬C∗D∗E + A∗¬B∗C∗¬D∗¬E +
A∗B∗¬C∗D∗¬E + A∗B∗C∗D∗¬E.
```

Figure 25: Boolean function for output O_4

```
Definition O3(A B C D E:bool) : bool:=
¬A∗B∗¬C∗D∗E + ¬A∗B∗C∗¬D∗E + ¬A∗B∗C∗D∗E +
A∗B∗¬C∗D∗E + A∗B∗C∗¬D∗E + A∗B∗C∗D∗E.
```

Figure 26: Boolean function for output O_3

```
Definition O5(A B C D E:bool) : bool:=
¬A∗¬B∗¬C∗¬D∗E + ¬A∗¬B∗¬C∗D∗¬E +
¬A∗¬B∗¬C∗D∗E + ¬A∗¬B∗C∗¬D∗¬E +
¬A∗¬B∗C∗¬D∗E + ¬A∗¬B∗C∗D∗¬E +
¬A∗¬B∗C∗D∗E + ¬A∗B∗¬C∗¬D∗¬E +
¬A∗B∗¬C∗¬D∗E + ¬A∗B∗¬C∗D∗¬E +
¬A∗B∗¬C∗D∗E + ¬A∗B∗C∗¬D∗¬E +
¬A∗B∗C∗¬D∗E + ¬A∗B∗C∗D∗¬E +
¬A∗B∗C∗D∗E + A∗¬B∗¬C∗¬D∗E +
A∗¬B∗¬C∗D∗¬E + A∗¬B∗¬C∗D∗E +
A∗¬B∗C∗¬D∗¬E + A∗¬B∗C∗¬D∗E +
A∗¬B∗C∗D∗¬E + A∗¬B∗C∗D∗E +
A∗B∗¬C∗¬D∗¬E + A∗B∗¬C∗¬D∗E +
A∗B∗¬C∗D∗¬E + A∗B∗¬C∗D∗E +
A∗B∗C∗¬D∗¬E + A∗B∗C∗¬D∗E +
A∗B∗C∗D∗¬E + A∗B∗C∗D∗E.
```

Figure 24: Boolean function for output O_5

```
Definition O2(A B C D E:bool) : bool:=
¬A∗¬B∗¬C∗¬D∗¬E + ¬A∗¬B∗¬C∗D∗¬E +
¬A∗¬B∗C∗¬D∗¬E + ¬A∗¬B∗C∗¬D∗E +
¬A∗¬B∗C∗D∗¬E + ¬A∗B∗¬C∗¬D∗¬E +
¬A∗B∗¬C∗¬D∗E + ¬A∗B∗C∗¬D∗¬E +
¬A∗B∗C∗¬D∗E + ¬A∗B∗C∗D∗E +
A∗¬B∗¬C∗¬D∗¬E + A∗¬B∗¬C∗D∗¬E +
A∗¬B∗C∗¬D∗¬E + A∗¬B∗C∗¬D∗E +
A∗¬B∗C∗D∗¬E + A∗B∗¬C∗¬D∗¬E +
A∗B∗¬C∗¬D∗E + A∗B∗C∗¬D∗¬E +
A∗B∗C∗¬D∗E + A∗B∗C∗D∗E.
```

Figure 27: Boolean function for output O_2

```
Definition O1(A B C D E:bool) : bool:=
¬A∗¬B∗¬C∗¬D∗E + ¬A∗¬B∗¬C∗D∗E +
¬A∗B∗¬C∗D∗¬E + ¬A∗B∗C∗¬D∗¬E +
¬A∗B∗C∗¬D∗E + ¬A∗B∗C∗D∗¬E +
¬A∗B∗C∗D∗E + A∗¬B∗¬C∗¬D∗E +
A∗¬B∗¬C∗D∗E + A∗B∗¬C∗D∗¬E +
A∗B∗C∗¬D∗¬E + A∗B∗C∗¬D∗E +
A∗B∗C∗D∗¬E + A∗B∗C∗D∗E.
```

Figure 28: Boolean function for output O_1

```
Definition O0(A B C D E:bool) : bool:=
¬A∗¬B∗¬C∗D∗¬E + ¬A∗¬B∗¬C∗D∗E +
¬A∗¬B∗C∗¬D∗E + ¬A∗B∗¬C∗¬D∗E +
¬A∗B∗¬C∗D∗¬E + ¬A∗B∗¬C∗D∗E +
¬A∗B∗C∗¬D∗¬E + ¬A∗B∗C∗D∗¬E +
A∗¬B∗¬C∗D∗¬E + A∗¬B∗¬C∗D∗E +
A∗¬B∗C∗¬D∗E + A∗B∗¬C∗¬D∗E +
A∗B∗¬C∗D∗¬E + A∗B∗¬C∗D∗E +
A∗B∗C∗¬D∗¬E + A∗B∗C∗D∗¬E.
```

Figure 29: Boolean function for output O_0

To design the memory chip, each output is considered as a Boolean function of the five input variables E−A. 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 six[2] Boolean functions $O_6 - O_0$ as defined in the Figures 23−29.

To design an efficient memory chip, all the seven functions (for outputs $O_6 - O_0$) are simplified using K-map method to functions O′6−O′0 as shown in Figure 30. The simplified Boolean functions has fewer literals and terms (Figure 30) as compared to original functions (Figures 23−29) and would produce a simple and efficient circuit layout. The K-map simplification process provides a systematic mean to simplify functions, however, it does not guarantee preservation of the intended behaviour.

```
Definition O′6(A B C D E:bool) : bool := ¬C + ¬D + ¬E.
Definition O′5(A B C D E:bool) : bool := B + C + D + E.
Definition O′4(A B C D E:bool) : bool :=
  ¬B∗¬C + ¬B∗¬D∗¬E + B∗D∗¬E.
Definition O′3(A B C D E:bool) : bool :=
  B∗C∗E + B∗D∗E.
Definition O′2(A B C D E:bool) : bool :=
  ¬B∗¬E + B∗¬D + C∗¬D + ¬D∗¬E + B∗C∗E.
Definition O′1(A B C D E:bool) : bool :=
  ¬B∗¬C∗E + B∗C + B∗D∗¬E.
Definition O′0(A B C D E:bool) : bool :=
  ¬B∗C∗¬D∗E + ¬C∗D + B∗¬C∗E + B∗C∗¬E.
```

Figure 30: Simplified Boolean functions

```
Lemma equiv_OO′6:
 ∀ A B C D E, O6 A B C D E = O′6 A B C D E.
Lemma equiv_OO′5:
 ∀ A B C D E, O5 A B C D E = O′5 A B C D E.
Lemma equiv_OO′4:
 ∀ A B C D E, O4 A B C D E = O′4 A B C D E.
Lemma equiv_OO′3:
 ∀ A B C D E, O3 A B C D E = O′3 A B C D E.
Lemma equiv_OO′2:
 ∀ A B C D E, O2 A B C D E = O′2 A B C D E.
Lemma equiv_OO′1:
 ∀ A B C D E, O1 A B C D E = O′1 A B C D E.
Lemma equiv_OO′0:
 ∀ A B C D E, O0 A B C D E = O′0 A B C D E.
```

Figure 31: Checking equivalence of functions

[2] As the column O_7 has all 0s, the output O_7 is constant and set to logic 0.

```
Theorem circuit_equivalence: ∀ 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)∗(O1 A B C D E)∗
(O0 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).
```

Figure 32: Checking circuit equivalence

To prove circuit equivalence, first the individual functions are proved equivalent using Coq lemmas shown in the Figure 31. The functions O6−O0 and O′6−O′0 in these lemmas takes variables E−A as input arguments and returns values of type `bool`. These lemmas state that both versions of functions are equivalent. All these lemmas are proved by case analysis on the variables E−A. To prove the circuits are functionally equivalent, a theorem is defined in the Figure 32. The theorem states that the conjunction of functions *before* simplification is equivalent to the conjunction of functions *after* simplification. This theorem is proved by applying (using `rewrite` command) the function equivalence lemmas (Figure 31) proved earlier.

6. Results

A ROM memory chip was designed using the formal framework developed, simplified using K-map method and mathematically proved that the initial and simplified circuits are functionally equivalent. That is, we mathematically proved that the logic circuit in Figure 33 behaves exactly the same as the simplified circuit in Figure 34[3].

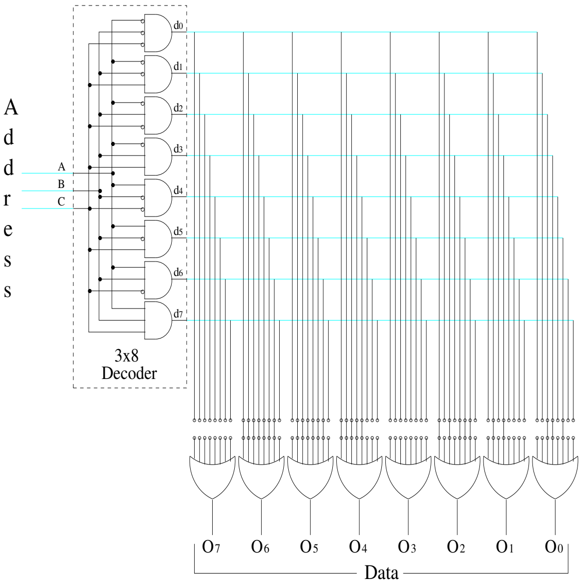

Figure 33: Design of ROM before simplification

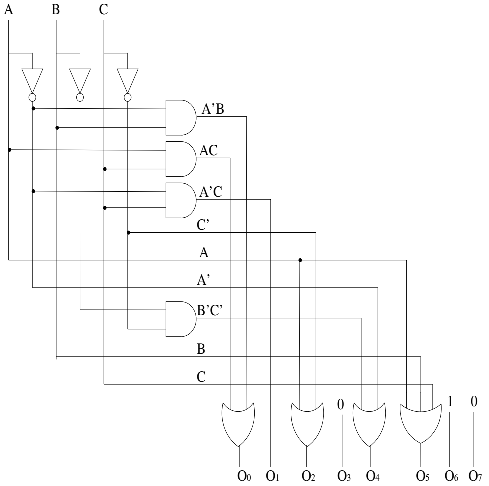

Figure 34: Design of ROM after simplification

For this correctness check, first a formal foundation was developed in Coq and then the circuit was designed and proved correct using Coq resulting a framework for formal specification and reasoning of logic circuits.

[3] The simplification and formal proofs were carried for larger 32×8 ROM and for simplicity, the simplified versions of a 16×8 are sketched here.

7. Related Work

A common approach to check two circuits are functionally equivalent is to extensively simulate two circuits in conventional tools such as VCS [23] and Icarus[24] by providing an input vector to both executions and comparing their results for all inputs. Checking the equivalence of two circuits for all possible inputs with each input of n bits requires simulation of the circuits for 2^n total inputs. Another popular approach to equivalence checking is to use formal methods based on mathematical tools and techniques. W. K. Lam [13] gives a detailed comparison of simulation and formal method-based approaches for hardware design verification.

The most common form of formal verification is to use *model checking* [14]. A model checker verifies whether or not the model (of the circuit) satisfies a property represented as a formula in the form of temporal logic. In model checking, the set of states of the model is represented as a list or table (*explicit model checking*), however, as the number of states may grow exponentially, this may result the *state explosion problem* [15, 16]. To reduce the *intuitive* complexity, the state space is *symbolically* represented implicitly using Boolean functions. Boolean formulas are manipulated efficiently using compact canonical graph representations of Boolean functions [25].

Binary Decision Diagrams (BDDs) [26, 27], a canonical description of Boolean functions, is one such popular symbolic representation used in various techniques and frameworks for the analysis of programs/finite state machines [1, 2, 3] and logic circuits [4]. To get a compact canonical representation of the Boolean function, Bryant [28, 29] proposed two restrictions on the binary diagrams: impose an (total) ordering among the variables and remove redundant vertices. These restrictions are imposed until a *reduced ordered binary decision diagram* (ROBDD) is achieved. Even though, the advent of modern algorithms [28, 29] speeds up the verification process by exploring the structural similarity of the circuits, the required BDD representations may still result memory explosion [30]. Due to its sensitivity to type and size of the system under test, BDD can not represent some designs efficiently, such as multipliers [31]. Furthermore, BDD can not prevent all state explosion problems in all cases [3].

As a by product of advances in tools [32, 33, 34] to solve Boolean Satisfiability (SAT) problem, formal verification based on SAT solver is considered a viable alternative to BDD based approach [31]. To overcome the drawbacks of the BDD, Biere et al. [25] proposed *bounded model checking*, a restricted form of model checking, to search for counter examples in selected executions with lengths *bounded* above by an integer value. Unfortunately, bounded model checking inherits complexity problem of the model checking which limiting it in its capacity. In circuit based bounded model checking, the circuit structure is exploited by using an intermediate representation called And-Inverter-Graph [35] that preserves the structure and allows to implement *sweeping* [36] technique to identity and merge equivalent nodes. Amla et al. [31] analysed the performance of various bounded and unbounded SAT-based model checking techniques in industrial environment.

A model checker automatically proves circuits' equivalence, however, it may not return due to state explosion and has memory and run time limitations [13]. Unlike *automated* proof tools, equivalence can be proved using *human assisted* tools called interactive theorem prover such as Coq [17] and Isabelle/HOL [18]. To reason about circuits described in a description language, Wilayat et al. [37] introduced a formal language Veriformal: a hardware description language with mathematical foundation. The description language VeriFormal, together with the interpreter and input feeder, provides an *hybrid* framework for simulation as well as property checking of the designs. The authors used VeriFormal and proved in Isabelle/HOL, the host language of VeriFormal, that two simple circuits are functionally equivalent. Their language, describes circuits at RTL level and hence can be used to reason about the circuits at that level, however, the verification process is very tedious even for simple circuits.

8. Conclusions

Functional equivalence checking is required to ensure transformation in the design process does not alter the intended behaviour of the circuit. In this paper, a formal framework for circuit equivalence checking based on Boolean algebra is developed in theorem prover Coq. The formal setting is used to reason about the Boolean algebra as well as circuits described in the algebra. To demonstrate the significance of the framework, basic theorems of the Boolean algebra are proved. Furthermore, a 32-byes ROM memory chip is design and simplified using K-map method. It is formally verified that the K-map simplification process does not alter the intended behaviour of the memory.

References

[1] O. Lhoták, Program analysis using binary decision diagrams, Vol. 68, 2006.

[2] J. Whaley, D. Avots, M. Carbin, M. S. Lam, Using datalog with binary decision diagrams for program analysis, in: Asian Symposium on Programming Languages and Systems, Springer, 2005, pp. 97–118.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L.-J. Hwang, Symbolic model checking: 1020 states and beyond, Information and computation 98 (2) (1992) 142–170.

[4] B. L. SYNTHESIS, Abc: a system for sequential synthesis and verification, release 70930 (2007).

[5] M. M. Mano, Digital logic and computer design, Pearson Education India, 2017.

[6] Wolframalpha computational knowledge engine, `https://www.wolframalpha.com/`, accessed: 2018-02-07.

[7] Online minimization of boolean functions, https://www.tma.main.jp/logic/index_en.html/, accessed: 2018-02-07.

[8] R. Sondre, A. Mohamed, D. Lars, Eirik, Online database of boolean functions, http://www.ii.uib.no/~mohamedaa/odbf/index.html, accessed: 2018-02-07.

[9] R. Lean, M. Kryle, Marxel, QMSolver, http://agila.upm.edu.ph/~kmmolina/qms/index.html, accessed: 2018-02-07.

[10] Logic circuit simplification (SOP and POS), http://www.32x8.com/, accessed: 2018-02-07.

[11] M. Karnaugh, The map method for synthesis of combinational logic circuits, Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics 72 (5) (1953) 593–599.

[12] T. K. Jain, D. S. Kushwaha, A. K. Misra, Optimization of the quine-mccluskey method for the minimization of the boolean expressions, in: Autonomic and Autonomous Systems, 2008. ICAS 2008. Fourth International Conference on, IEEE, 2008, pp. 165–168.

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

[14] E. M. Clarke, O. Grumberg, D. Peled, Model checking, MIT press, 1999.

[15] A. Valmari, The state explosion problem, Lectures on Petri nets I: Basic models (1998) 429–528.

[16] E. M. Clarke, W. Klieber, M. Nováček, P. Zuliani, Model checking and the state explosion problem, in: Tools for Practical Software Verification, Springer, 2012, pp. 1–30.

[17] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al., The coq proof assistant reference manual: Version 6.1, Ph.D. thesis, Inria (1997).

[18] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL: a proof assistant for higher-order logic, Vol. 2283, Springer Science & Business Media, 2002.

[19] G. Boole, Investigation of the Laws of Thought. 1854, Dover, 1958.

[20] E. V. Huntington, New sets of independent postulates for the algebra of logic, with special reference to whitehead and russells principia mathematica, Transactions of the American Mathematical Society 35 (1) (1933) 274–304.

[21] C. E. Shannon, A symbolic analysis of relay and switching circuits, Electrical Engineering 57 (12) (1938) 713–723.

[22] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, B. Yorgey, Software foundations, Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html.

[23] VCS Online Simulator, `http://www.edaplayground.com/x/FfR`, accessed: 2015-12-25.

[24] Icarus Verilog, `http://www.icarus.com/eda/verilog/`, accessed: 2016-12-06.

[25] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, Bounded model checking, Advances in computers 58 (2003) 117–148.

[26] S. B. Akers, Binary decision diagrams, IEEE Transactions on computers (6) (1978) 509–516.

[27] F. Somenzi, Binary decision diagrams, NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES 173 (1999) 303–368.

[28] R. E. Bryant, Symbolic boolean manipulation with ordered binary-decision diagrams, ACM Computing Surveys (CSUR) 24 (3) (1992) 293–318.

[29] K. S. Brace, R. L. Rudell, R. E. Bryant, Efficient implementation of a bdd package, in: Proceedings of the 27th ACM/IEEE design automation conference, ACM, 1991, pp. 40–45.

[30] S.-Y. Huang, K.-T. T. Cheng, Formal equivalence checking and design debugging, Vol. 12, Springer Science & Business Media, 2012.

[31] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, K. L. McMillan, An analysis of sat-based model checking techniques in an industrial environment, in: Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Springer, 2005, pp. 254–268.

[32] J. P. Marques-Silva, K. A. Sakallah, Grasp: A search algorithm for propositional satisfiability, IEEE Transactions on Computers 48 (5) (1999) 506–521.

[33] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient sat solver, in: Proceedings of the 38th annual Design Automation Conference, ACM, 2001, pp. 530–535.

[34] E. Goldberg, Y. Novikov, Berkmin: A fast and robust sat-solver, Discrete Applied Mathematics 155 (12) (2007) 1549–1561.

[35] A. Kuehlmann, V. Paruthi, F. Krohm, M. K. Ganai, Robust boolean reasoning for equivalence checking and functional property verification, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21 (12) (2002) 1377–1394.

[36] A. Kuehlmann, Dynamic transition relation simplification for bounded property checking, in: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design, IEEE Computer Society, 2004, pp. 50–57.

[37] K. Wilayat, T. Alwen, S. David, Veriformal: An executable formal model of a hardware description language, in: A Systems Approach to Cyber Security, 2017 2nd Singapore Cyber Security R&D Conference on, SGCSC, 2017, pp. 19–36.