A Novel Formalization of Symbolic Trajectory Evaluation Semantics in Isabelle/HOL

Yongjian Li∗, a, William N. N. Hungb, Xiaoyu Songc

aState Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
bSynopsys Inc., Mountain View, California 94043, USA
cDept. ECE, Portland State University, Portland, Oregon 97207, USA

Abstract

This paper presents a formal symbolic trajectory evaluation theory based on a structural netlist circuit model, instead of an abstract next state function. We introduce an inductive definition for netlists, which gives an accurate and formal definition for netlist structures. A closure state function of netlists is formally introduced in terms of the formal netlist model. We refine the definition of defining trajectory and the STE implementation to deal with the closure state function. The close correspondence between netlist structures and properties is discussed. We present a set of novel algebraic laws to characterize the relation between structures and properties of netlists. Finally, the application of the new laws is demonstrated by parameterized verification of properties of content addressable memories.

1. Introduction

Symbolic trajectory evaluation (STE) is an efficient formal hardware verification method that has grown from the combination of multi-valued simulation and symbolic simulation [1]. It has shown great promise in verifying medium to large scale industrial hardware designs with a higher degree of automation. STE has been in active use in Intel, Motorola, and IBM. In Intel, for instance, STE was used to verify a floating point arithmetic unit against IEEE standard 754 and a complex IA instruction length decoder unit [2, 3]. In addition, the FORTE formal hardware verification tool, which combines STE and theorem proving in a higher-order logic, has been developed at Intel[4].

In the classical STE literature, a circuit is a set of logical gates and storage element connected by nodes (wires). A state of the circuit is a function from

∗Corresponding author
Email addresses: lyj238@ios.ac.cn (Yongjian Li), william_hung@alumni.utexas.net (William N. N. Hung), song@ece.pdx.edu (Xiaoyu Song)
1The first author is supported by grants (No. 60833001, 60496321, 60421001) from National Natural Science Foundation of China.
its nodes to their values. The behaviors of the circuit is commonly modelled by some abstract next-state function, usually written $Y$ \cite{1, 5}. Given a state of the nodes at the current time, the $Y$ function returns the states of the nodes at the next time. For convenience, we informally call these classical semantics $Y$-semantics. However, this work does not formally explain how a corresponding $Y$ function is derived from a netlist structure. Besides, a next-state function only expresses a relation between nodes in successive points in time, while ignoring the relation between nodes in the circuit at the same time point. Therefore, a semantics based on next-state functions cannot deal with assertions that express a relation between circuit nodes at the same time point.

![Figure 1: A netlist example](image)

For instance, consider the 2-bit comparator circuit drawn by Quartus II \cite{6} in Figure. 1. The circuit consists of two XNOR-gates and an AND-gate. Provided that the delay time of all the gates is zero, and input primitives $a_0, b_0, a_1, b_1$ of the circuit are driven by new values $0, 0, 1, 1$, then nodes $c_0, c_1, out$ should be $1, 1, 1$ immediately, not at the next time. Because the above change on values of nodes is finished at the current time, it is very cumbersome for a $Y$-semantics to cover such information calculation because it only depicts state transition between successive time points.

Recently Roorda and Claessen clarify the semantics of STE model checking by providing closure semantics \cite{7, 8}. The closure semantics of STE takes as an input a state of the circuit, and calculates all information about the circuit state at the same point in time that can be derived by propagating the information in the input state in a forwards fashion. Subsequently, the definition of defining trajectory and the STE implementation are refined to deal with the closure functions rather than the next-state function. However, they did not formally define the structure of netlist. Their definition is just a sketchy property description of a circuit, that is, there is neither a cycle in the combinational part nor name conflict between two output nodes of two gates in the circuit. However it does not tell us how the circuit is constructed. From such definition, it is very difficult to naturally formalize the closure function of a circuit as a form of primitive recursive function or a total recursive function. In addition, many interesting properties of circuits are closely related with its structures. For example, the output node of an AND-gate will be set low if one of its input
nodes is driven by a low value. A good netlist formalization is a base on which we can conveniently explore these interesting properties. To sum up, a formalization of netlist structures is the base of the STE theory framework which has a netlist computational model.

1.1. Our Contribution

The main contributions of this paper are twofold. The first one is to continue to develop a *formal* STE theory based on a netlist computation model. Our work gives a more formal closure semantics which faithfully explains how STE model checker (or symbolic simulator) work. Here we not only formally explain how a next-state function $Y$ is derived from a netlist structure, but also deal with combinational properties. This semantics has netlist as a solid background, therefore makes STE easier to be understood formally.

- We introduce an inductive definition for netlists. It not only provides us an accurate and constructive formulation for a netlist, but also introduces an effective and rigorous technique of rule induction to prove properties of netlists. In particular, we use the induction principle on the structures of netlists to formally prove that the output of a logical entity in a netlist is uniquely defined.

- We formally define the closure semantics of netlists based on the formal netlist model. The simulation result of a netlist in a driven state is defined as a relation between nodes and values. The relation is formally proved to be single-valued, and naturally used to derive the closure function of the netlist on driven states.

- We refine the definition of defining trajectory and the STE implementation to deal with the newly introduced closure functions.

- We introduce symmetry between netlist structures in our formal netlist model, and relate it with symmetry between STE assertions. We prove the close correspondence between the two kinds of symmetry. This result resembles a similar symmetric reduction methodology shown in [9].

- We show a set of algebraic laws which relates a netlist structure with its properties. These laws can be seen as an algebraic semantics for STE, and used to verify interesting STE trajectory assertions on circuit netlists.

The second contribution is to formalize the STE theory in a theorem prover, with the hope that the theoretical improvement can make it feasible to mechanize the fundamental STE theory based on a netlist circuit model. By using a theorem prover to formalize the meta-theory of STE, we hope to raise the standard of rigor of, and hence our confidence in STE. We formalize our theory in Isabelle/HOL, an instantiation of generic theorem prover Isabelle/HOL to higher-order logic [10]. The formalized theories in Isabelle/HOL are available in [11]. Isabelle/HOL is appropriate because of its support for inductively defined
sets and its automatic tools. However, the fact that we used Isabelle is not especially relevant for the topic, and the formalization proposal in this work can also be implemented by other higher-order theorem provers such as PVS and COQ.

1.2. Related Work

Besides the inheritance from the proposal of closure semantics in [7, 8], our work is also closely with [12, 13, 14, 15, 16]. Works in [12, 13, 14] have demonstrated that higher-order logic is well suited for modelling and reasoning about hardware, so we decide to use higher-order logic to formalize the STE theory. The work in [15] outlined the theoretical foundation for linking the general logic of STE with higher order logic. The main result is a formal translation from trajectory evaluation’s temporal operators over lattices to a shallow embedding of the temporal operators over Boolean streams. Any result verified by the trajectory evaluation algorithm will hold in the relational world. In [16], Darbari did the machine based formalization in HOL for a theory whose details were described in [3], and he extended the work by proving the soundness of a symmetry reduction method in his framework [17]. The above work provides a formal framework to formalize the lattice value, the syntax and the semantics of trajectory formulas. These formalizing techniques are still used in our work. But all of this work formalizes a kind of Y-semantics in which a circuit is modelled by an abstract next-state function $Y$.

In [18, 19, 20, 21, 22], functional program languages have been advocated for hardware verification. Especially, useful insights of using inductive data types to formally describe circuit structures are provided in the work on $\mu$fp [18], Hydra [19, 20], Lava [21]. Other important features of a functional programming language such as Haskell: monads, type classes, polymorphism, and higher order functions are employed to model, verify, and implement a circuit in these work. However, combinational cycles and name conflicting between different entries should be eliminated in a legal netlist structure, it may be not very easy to directly use an inductive data type to formalize the two legal requirements. Instead, we use an inductively defined set to model all legal netlists. The corresponding induction rules formally specifies the legal requirement when a legal netlist is constructed.

Our formalization technique on the closure semantics is inspired by the work by Nipkow and Paulson in [23],[24]. Nipkow proposed an induction approach to formalize the first 100 pages of Winskel’s textbook [25], which covers the operational and denotational and axiomatic semantics of an imperative language called IMP. For instance, the natural semantics of IMP is inductively defined by a set of configurations each of which is a triple. We borrow the induction principle to formally specify the closure semantics of a netlist. Namely, we define the simulation result of a netlist by a relation which is also an inductively defined set of pairs between nodes and values. Furthermore, we use the technique proposed in [24] to define the unique closure function in such a relation, and prove that the corresponding function is well-defined because the closure relation is single-valued.
In the classical literature of STE, some laws have already been introduced to decompose a complex STE assertion [1, 17]. However, these laws usually hold for any circuit and can’t relate properties of a circuit with their special structure due to the lack of a formalization on circuit structures. Different from their work, a set of novel laws are introduced to formally explore the special structures of a circuit in our formal netlist model. To the best of our knowledge, these laws has never been discussed in previous STE work.

Darbari proposed a symmetry reduction method for STE model checking using a structured model [17]. Our symmetry reduction method is deeply inspired by his work. However, he used Y–semantics, and avoided discussing symmetry between netlist structures directly. He proposed a higher-level design language which allows to record symmetry of a circuit, and make a connection to the theory of STE logic. This connection is made by giving functions that derive a next-state function from the structured models and proving lemmas that guarantee that if the structured models have symmetry, then the corresponding derived next-state function will have symmetry as well. In our theory, the high-level modeling language, and the connection is not needed, we directly discuss symmetry between netlist structures in our formal netlist model, and relate it with symmetry between STE properties. Here our motivation is to provide a symmetry reduction method when we face a netlist model which is directly compiled from a popular hardware language such as Verilog and VHDL which still does not support a type system to record symmetry in a design.

1.3. Presentation of the paper

As mentioned before, our work involves both developments on the STE theory itself and the formalization of the theory in a theorem prover in order to provide mechanical support for the new STE theory. Because formalization is one of our main objectives in this paper and our implementation is tailored to Isabelle/HOL, we directly use parts of our Isabelle’s theories to introduce definitions and lemmas to convey the main idea of the formalization. In order to make the formalized theories readable for the readers who are not familiar with Isabelle, we also try to give a detailed text account for the formalized theories by using usual mathematical notations. Thus our work is interesting not only for the Isabelle/HOL users, but also for those who either are interested in STE theory or in theorem proving work by using other higher-order theorem provers such as HOL.

Isabelle/HOL has a polymorphic type system as in ML [26]. Type inference eliminates the need to specify types in expressions. Lemmas about lists, sets, etc., are polymorphic, and the prover uses the appropriate types automatically. Besides, a function in Isabelle/HOL syntax is usually defined in a curried form instead of a tupled form, that is, we often use the notation $f x y$ to stand for $f(x, y)$. The advantage of a curried function is to allow a partial function application [26]. We use the notation $[A_1; A_2; \ldots; A_n] \Rightarrow B$ to mean that with assumptions $A_1, \ldots, A_n$, we can derive a conclusion $B$. For a pair $(a, b)$, $\text{fst}(a, b) \equiv a$ and $\text{snd}(a, b) \equiv b$. We write $x\#xs$ for the list that extends $xs$ with $x$, $[x_1, \ldots, x_n]$ for a list $x_1\#\ldots\#x_n\#[]$, $xs@ys$ for the result list by concatenating
xs with ys, xs!i for the ith element of the list xs (counting from 0 as the first element), set xs for the set of all the elements in xs, x mem ls for x ∈ (set ls) and length xs for the length of the list xs. We also need a definite description THE x.P(x) to denote the x such that P(x) is true, provided that there exists a unique such x; otherwise, it returns an arbitrary value of the expected type.

In the appendix, we provide detail introduction for Isabelle/HOL notations which formalize the concepts in the paper.

1.4. Structure of the Paper

The remainder of this paper is organized as follows: Section 2 formalizes preliminary definitions on the four-valued lattice. Section 3 introduces the structure of a netlist and its formal model. Section 4 formalizes the syntax and semantics of trajectory formulas. Section 5 formalizes the closure function induced from a netlist. Section 6 introduces the most fundamental result of STE: the soundness of using defining trajectories and defining sequence to verify STE assertions. Section 7 discusses sub-netlists of a netlist. Section 8 explores the close correspondence between symmetry in circuit structures and symmetry in circuit properties. Section 9 presents some interesting algebraic laws to explore the close relation between the structure and properties of a circuit. Section 10 demonstrates how to apply symmetry reduction and these new laws to decompose STE assertions by a case study on CAMs, which is a typical example used in STE literature. Section 11 concludes the paper.

2. Background

Four values ff, tt, X, and ⊤ are used in STE simulation [1]. ff and tt are standard binary values false and true. The third value X stands for an unknown value, while the fourth value ⊤ a clash value. Formally, we define \[\forall = \{\text{ff}, \text{tt}, \text{X}, \top\}\]

It is common to introduce a truth information ordering \(\sqsubseteq\) on \(\forall\) as follows: \(\text{X} \sqsubseteq \text{ff}, \text{X} \sqsubseteq \text{tt}\), while \(\text{ff}\) and \(\text{tt}\) are incomparable, \(\text{ff} \sqsubseteq \top\), and \(\text{tt} \sqsubseteq \top\). Namely, the unknown value \(\text{X}\) contains no truth information; the mutually incomparable values \(\text{ff}\) and \(\text{tt}\) contain sufficient information to determine truth exactly, and the top value \(\top\) contains inconsistent truth information. We can easily see that \(\forall\) with the ordering relation \(\sqsubseteq\) forms a lattice. We can introduce a join or a least-upper bound operator \(\sqcup\) with respect to the ordering \(\sqsubseteq\). Its rather routine to check that \(a \sqsubseteq b\) if and only if \(a \sqcup b = a\). For other operators on the domain \(\forall\), there are natural definitions for negation \(\text{NOT}(\neg_4)\), conjunction \(\text{AND}(\land_4)\), disjunction \(\text{OR}(\lor_4)^2\), etc. The classic definitions of these operators are shown in Figure 3.

\[2\text{Here we use the subscript 4 to distinguish the x-symbols of these operators from their counterparts in boolean domain.}\]
Figure 2: STE lattice

<table>
<thead>
<tr>
<th></th>
<th>¬a</th>
<th>a</th>
<th>b</th>
<th>a ∧₄ b</th>
<th>a</th>
<th>b</th>
<th>a ∨₄ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff</td>
<td>tt</td>
<td>ff</td>
<td>v₃</td>
<td>ff</td>
<td>ff</td>
<td>v₄</td>
<td>v₄</td>
</tr>
<tr>
<td>tt</td>
<td>ff</td>
<td>tt</td>
<td>v₄</td>
<td>v₄</td>
<td>tt</td>
<td>v₃</td>
<td>tt</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>ff</td>
<td>ff</td>
<td>X</td>
<td>ff</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>T</td>
<td>v₄</td>
<td>T</td>
</tr>
<tr>
<td>v₄</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>v₄</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Figure 3: Operators over four-valued lattice (v₃ ∈ {tt, ff, X}, v₄ ∈ {tt, ff, X, ⊤})

In order to define the set of four lattice values \( V \), we use strategy of dual-rail encoding [15, 16]. Thus, we introduce a type `boolPairs`, and encode the four values in \( V \) as four constants of type `boolPairs`.

\[
\text{types boolPairs} = \text{bool} \times \text{bool}
\]

\[
\top \equiv (\text{False}, \text{False})
\]

\[
\text{tt} \equiv (\text{True}, \text{False})
\]

\[
\text{ff} \equiv (\text{False}, \text{True})
\]

\[
X \equiv (\text{True}, \text{True})
\]

The least-upper bound operator \( ⊔ \) and the partial ordering relation \( ⊑ \) are defined as follows:

\[
a ⊔ b \equiv (\text{fst a} ∧ \text{fst b}, \text{snd a} ∧ \text{snd b})
\]

\[
a ⊑ b \equiv a ⊔ b = b
\]

Due to limitation of space, more formal definitions of other operators can be found in [11].

3. Circuit Netlist Formalization

3.1. An informal model of circuit netlists

A circuit is modelled by a netlist, which is a set of nodes connected by logical entities such as I/O devices, gates and one-phase delays. I/O devices are pins connected to its environment. For simplicity, only input devices are
used in this work. Gates describe combinational logics deciding the relationship between values of nodes. Delays refer to all sequential elements which can keep "state". In real-world VLSI designs, there are different types of sequential devices, some of which can be more complex than our delay devices in both structures and behaviors. However, we will see that real-world sequential devices can be modelled by our simple delay elements in later discussion.

In a netlist description language such as BLIF [27], input pins of a circuit are defined as follows:

```
.inputs x y
```

A gate is specified by a truth table, as shown below:

```
.names in\_1 in\_2 \ldots out
in\_1.value \_1 in\_2.value \_1 \ldots out.value \_1
in\_1.value \_2 in\_2.value \_2 \ldots out.value \_2
```

where \( \text{in}_1, \text{in}_2, \ldots \) are names of the inputs of the gate, \( \text{out} \) is the name of its output. The subsequent lines define the on-off sets: \( \text{in}_1.\text{value}_1 \) is one of 0, 1, or \(-\) (don't care), and \( \text{out}.\text{value}_1 \) is one 0 or 1.

A truth table encapsulates a programmable logical array (PLA), which is expanded to AND gates driving an OR gate. So it is natural for us to associate a truth table with a function on \( \mathbb{V} \). For example, the table of the XNOR gate is corresponding to a function \( \lambda a \ b. a \lor_4 b \lor_4 \neg_4 a \lor_4 \neg_4 b \). Informally we write \( F_{\text{tab}} \) for the induced function from the table \( \text{tab} \).

For instance, a two-input AND gate with inputs \( a \) and \( b \) and output \( \text{foo} \), and a two-input XNOR gate with the same inputs and output of a netlist could be defined as follows:

```
.names a b foo
.name a b foo
11 1 00 1
```

A latch is defined as follows:

```
.latch latch\_input latch\_output
```

where a latch has a data input and an output node. As mentioned before, our latch is simply a one-phase delay element. The value of node \( \text{latch\_output} \) in the next time is the value of \( \text{latch\_input} \) in the current time.

**Remark 1.** In fact, the definition of a latch in BLIF is more complex than ours. In BLIF, a latch is defined as the following statement: \( .\text{latch } \text{latch\_input } \text{latch\_output } \text{type } \text{control\_signal } [\text{latch\_control\_list}] \), where \text{type} specifies whether the latch is edge-sensitive or level-sensitive. Latch control constructs specifies the set or reset or enable control signals of the latch. For example, \( .\text{latch } \text{ini outl re clk as= set ar= reset en= eni} \) specifies a flip-flop which is driven at the rising edge of signal \( \text{clk} \) with an input signal \( \text{ini} \), an output signal \( \text{outl} \), an asynchronous reset signal \( \text{reset} \) and a asynchronous set signal \( \text{set} \). But any type of latch can be modelled by combinational gates and delay elements. Figure 4 gives an example to show how a rising-edge triggered flipflop is modelled by delay elements and combinational gates, where an inverted triangle stands for a delay element. In Forte, \( d \) and \( d\_## \) stands for the input and output node of the delay element respectively.
### 3.2. Formalization of netlists

We first use the type `nat` as the type of nodes in our theory.

```plaintext
types node=nat
```

To formally define a truth table, we use an enumerating type `LIT` to specify a literal for defining on or off-sets, a type `LINE` to specify a line of a table, and `PLA` to define a table.

```plaintext
datatype LIT= One | Zero | DontCare
```

```plaintext
types LINE=LIT list
```

```plaintext
types PLA=LINE list
```

Input pins and gates and delays are three kinds of logical entities in a circuit, and are formally defined as follows:

```plaintext
datatype entity = Input node | Gate node "node list" PLA | Delay node node
```

Here we assume `inp`, `out` are node names, `inps` is a list of node names, `tab` is a table of type `PLA`. `Input inp` means that `inp` is an input pin of a netlist under study which is an interface between the netlist and its environment. `Gate out inps tab` refers to a gate which has `out` as its output node, and `inps` as its input nodes, `tab` as its truth table. As does the library function `get_node_truth_table` in Forte, a PLA in this paper lists clauses for inputs when an output is to go high only. For example, `Gate c1 [a1, b1] [[ONE, ONE]]` formally defines an AND gate. `Delay out inp` defines a delay which has `inp` as its input and `out` as its output respectively.

For a logical entity `g`, we define a function `fanOut` to map `g` to its output node, namely, `fanOut g ≡ inp`, if `g = Input inp`, or `fanOut g ≡ out` if `g = Gate out inps tab` or `g = Delay out inp`. Similarly, we also define a function `fanIn`
to map $g$ to the list of all its input nodes, that is, $\text{fanIn } g \equiv []$, if $g = \text{Input } inp$, or $\text{fanIn } g \equiv \text{inps}$ if $g = \text{Gate out inps tab}$, or $\text{fanIn } g \equiv \text{inp}$ if $g = \text{Delay out inp}$.

Consider a node $n$, a logical entity set $nl$, we say $\text{isDefinedIn } n \text{ nl}$ if $n$ is defined as an output of a logical entity in the $nl$. More formally, $\text{isDefinedIn } n \text{ nl} \equiv l \in nl \land \text{fanOut } l = n$. The set of all the nodes defined in the $nl$ is denoted by $\text{defAsOuts } nl \equiv \{ n.\text{isDefinedIn } n \text{ nl} \}$.

Now we come to the a crucial point, the formalization of netlists. Intuitively, a netlist is simply a set of logical entities connected by nodes, but adding entities into a netlist should follow some restriction rules to guarantee the legality of the structure of the netlist. Here we introduce an inductive definition for the set of all the netlists, as shown below:

$$
\text{const netlists :: (entity set) set}
$$

$$
\text{inductive netlists}
$$

$$
\text{intros}
$$

$$
\text{nilNetlist :} \emptyset \in \text{netlists};
$$

$$
\text{addInput :}
$$

$$
[ nl \in \text{netlists}; \neg \text{isDefinedIn } n \text{ nl} ] \implies \{ \text{Input } n \} \cup nl \in \text{netlists};
$$

$$
\text{addDelay :}
$$

$$
[ nl \in \text{netlists}; \neg \text{isDefinedIn } n \text{ nl} ] \implies \{ \text{Delay } n \text{ inp} \} \cup nl \in \text{netlists};
$$

$$
\text{addGate :}
$$

$$
[ nl \in \text{netlists}; \neg \text{isDefinedIn } n \text{ nl}; \forall \text{inps}, (\text{inps}, \text{mem inps}) \rightarrow \text{isDefinedIn } \text{inps}, nl; \\
\forall l. (l \text{ mem tab} \rightarrow \text{length } l = \text{length } \text{inps}) ] \\
\implies \{ \text{Gate } n \text{ inps tab } \} \cup nl \in \text{netlists}.
$$

In the above definition, rule $\text{nilNetlist}$ specifies an empty netlist. Other rules specify the order which should be followed to add a logical entity into a netlist. In the last three rules, the condition $\neg \text{isDefinedIn } n \text{ nl}$ requires that the output node $n$ of the newly added logical entity should not be an output of the existing entities in $nl$. This resolves the name conflicting of output nodes between two different logical entities in a netlist. In rule $\text{addGate}$, the third condition requires that all the input nodes of the newly added combinational gate must have been defined in the existing netlist. Combining this condition and the condition $\neg \text{isDefinedIn } n \text{ nl}$ can eliminate combinational cycle in a netlist. Unlike rule $\text{addGate}$, $\text{addDelay}$ rule allows that the input node of a delay can be used before the node is defined. Formally, when a component $\text{Delay } n \text{ inp}$ is added in the rule, $\text{inp}$ is a free variable which is only restricted by its type. If a delay’s output node is in the fanin cone of the delay, then a cycle passes the delay. Therefore, a cycle is allowed to pass a delay element.

Example 2. Let $\text{xnorTab} = [[\text{ZERO}, \text{ZERO}], [\text{ONE}, \text{ONE}]], \text{xnorG}_0 = \text{Gate } c_0 [a_0, b_0] \text{xnorTab}, \text{xnorG}_1 = \text{Gate } c_1 [a_1, b_1] \text{xnorTab}, \text{andTab} = [[\text{ONE}, \text{ONE}]], \text{andG} = \text{Gate out } [c_0, c_1] \text{andTab},$ then the set

$$
\text{nl} = \{ \text{Input } a_0, \text{Input } b_0, \text{Input } a_1, \text{Input } b_1, \text{xnorG}_0, \text{xnorG}_1, \text{andG} \}$$
stands for the netlist shown in Figure 1. In figure 4, let \( \text{tab}_1 = [\text{ONE, ZERO}] \), \( G_1 = \text{Gate sel [clk, clk.##,]} \text{tab}_1 \), and \( \text{tab}_2 = [[\text{ONE, ONE, DontCare}], \text{ZERO, DontCare, ONE}] \). \( G_2 = \text{Gate s [sel, d.##, s.##,]} \text{tab}_2, \text{delay}_1 = \text{Delay d.##, d}, \text{delay}_2 = \text{Delay s.##, s}, \text{nl}_2 = \{G_1, G_2, \text{delay}_1, \text{delay}_2\} \), \( \text{nl}_2 \) is also a netlist.

Our netlist model is sound in the sense that for any defined node \( n \) in a netlist, there is an unique logical entity in the netlist whose output node is \( n \).

In Isabelle, unique existence quantifier is denoted by \( \exists! \).

**Lemma 3.** \([n l \in \text{netlists}; \text{isDefinedIn } n nl] \Rightarrow \exists! l. l \in nl \land \text{fanOut } l = n \).

Because of the existence of the one-to-one mapping from a logical entity to its output node name, formally, we define \( \text{lookUp } nl n \equiv \text{THE } g.g \in nl \land \text{fanOut } g = n \).

Definition of netlists itself can not guarantee that each node of a netlist is defined because an input of a \( \text{delay} \) can be used without being defined. In real circuit designs, an input of a \( \text{delay} \) needs to be defined. If each input node of each logical entity in a netlist is defined as an output of another logical entity, then we call the netlist closed.

**Definition 4.** \( \text{isClosed } nl \equiv \forall m n. \text{isDefinedIn } m nl \rightarrow n \in \text{set } (\text{fanins }) ((\text{lookUp } nl m)) \rightarrow \text{isDefinedIn } n nl \)

**Example 5.** In Example 2, the netlist \( nl \) is closed; \( nl_2 \) is a netlist, but it is not closed because nodes \( s \) and \( d \) are not defined in \( nl_2 \).

We are mainly interested in closed netlists in our work, so we always assume that \( nl \in \text{netlists} \) and \( \text{isClosed } nl \) in the following discussion when we meet a word \( nl \). To save space, we omit the two side conditions when we present lemmas about a netlist \( nl \).

### 4. Syntax and Semantics of Trajectory Formula

**States.** A circuit state is an instantaneous snapshot of a circuit behavior given by an assignment of lattice values to nodes of the circuit. Therefore, type \( \text{state} = \text{node } \Rightarrow \text{boolPairs} \) is defined. A state sequence assigns a state to a time point. Here we still use \( \text{nat} \) to define the type \( \text{time} \). Thus, we define \( \text{stateSeq} = \text{time } \Rightarrow \text{state} \).

Naturally, we extend the ordering relation on the \( \text{state} \) and \( \text{stateSeq} \) types. we define \( s_1 \sqsubseteq s_2 \equiv \forall n.s_1 x \sqsubseteq s_2 x \), and \( sq_1 \sqsubseteq sq_2 \equiv \forall t.sq_1 t \sqsubseteq sq_2 t \).

**Trajectory Evaluation Logic.** Specifications in STE are symbolic trajectory formulas. In order to formalize the syntax of trajectory formulas, we introduce a datatype \( \text{trajForm} \) as follows:

```plaintext
datatype trajForm = Is1 node | Is0 node | chaos
| Next trajForm
| When bool trajForm (infixr \( \rightarrow \) 65)
| TAND trajForm trajForm (infixr and 65)
```
For convenience in reasoning, we introduce a novel formula chaos in our theory to represent that the values of all the nodes are unknown at all time. In the above definition, the definition of trajectory formulas is naturally symbolic in the sense that the Boolean guard $P$ can be simply defined as a boolean formula in HOL.

The semantics of trajectory formulas is formally defined as a primary recursion function \texttt{valid} on datatype \texttt{trajForm}.

\begin{verbatim}
consts
valid :: stateSeq \Rightarrow trajForm \Rightarrow bool
((\_ \models \_)[80,80][80])
primrec
sq \models (Is1 n) = tt \sqsubseteq (sq 0 n)
sq \models (Is0 n) = ff \sqsubseteq (sq 0 n)
sq \models chaos = True
sq \models (A\_1 \&\& T A\_2) = (sq \models A\_1 \&\& sq \models A\_2)
sq \models (P \rightarrow T A) = (P \rightarrow sq \models A)
sq \models (\text{Next } A) = ((\text{suffix } 1 \ sq) \models A)
\end{verbatim}

where notation ((\_ \models \_)[80,80][80]) stands for an infix notation \models for function \texttt{valid}, and \texttt{suffix } i sq \equiv \lambda t.\ sq(t+i).

5. Formalization of Closure Functions over Netlists

During STE simulation, information is propagated forwards through both a circuit structure and time. By simulation, we mean that a circuit \texttt{nl} takes a stimulating sequence as input and returns a result sequence. We first illustrate the meaning of information propagation forwards through the circuit structure at a time point. Namely, the circuit takes a state of the stimulating sequence at some time point, then calculates all information about the circuit at the same point that can be derived by propagating the information from any combination gate's input nodes to its output. After this propagation is finished, a new state of the circuit is returned as a simulation result of this time point. More specifically, given a state $s$, for an input node $n$ of the circuit, or a delay node, $s\ n$ is simply the value of $n$ after simulation. For an internal node $n$ which is an output of a gate with a truth table \texttt{tab}, provided that the returned values of inputs of the gate are $v_1, \ldots, v_i$ after simulation, the value of $n$ is returned as the upperbound of $s\ n$ and $F_{\text{tab}}\ v_1 \ldots v_i$.

For instance, suppose that $s\ a_0 = tt$, $s\ b_0 = tt$, $s\ a_1 = tt$, and $s\ n = X$ for any other nodes, a simulation for the circuit in Figure 1 is started at $s$, then in the end of time point 0, the result state $s'$ after simulation satisfies that $s'\ n = s\ n$ if $n \in \{a_0,b_0,a_1,b_1\}$, $s'\ c_0 = tt$, $s'\ c_1 = X$, and $s'\ out = X$. Formally, the information propagation can be represented as a set of value assignments as follows: $\{(a_0, tt), (b_0, tt), (a_1, tt), (b_1, X)(c_0, tt), (c_1, X), (out, X)\}$.

In order to define the closure semantics of netlists, we need some preliminary formalization on semantics of literals, lines, and truth tables. These are defined
rather straightforward: \( \text{funOfLit}(v, \text{lit}) \) returns the input value \( v \) if \( \text{lit} \) is on, else if \( \text{lit} \) is off, then returns the negation of \( v \), else just returns \( \text{tt} \). Here we briefly explain why \( \text{tt} \) is returned when the literal is \textit{DontCare}. Because \( \text{tt} \) is an unit for the operator AND in the four-valued domain, and \( \text{funOfLine vs line} \) is a conjunction of the values of literals in this line. At a state, if a value of a literal in a line is returned as \( \text{tt} \), then the value of this line will not be care of the value of this literal. \( \text{funOfLine vs line} \) returns the conjunction of the values of literals in a line provided that the values assigned to inputs are \textit{vs}. \( \text{funOfTab tab vs} \) returns the disjunction of the values of lines of a table provided that the values assigned to inputs are \textit{vs}. 

\[
\text{funOfLit} :: \text{boolPairs} \times \text{Lit} \Rightarrow \text{boolPairs}
\]
\[
\text{funOfLit} \ x \equiv \begin{cases} \text{snd} \ x = \text{ONE} & \Rightarrow \text{fst} \ x \\ \text{snd} \ x = \text{ZERO} & \Rightarrow \text{NOT} \ (\text{fst} \ x) \\ \text{else} & \Rightarrow \text{tt} \end{cases}
\]

\[
\text{funOfLine} :: \text{boolPairs list} \Rightarrow \text{LINE} \Rightarrow \text{boolPairs}
\]
\[
\text{primrec funOfLine bps} \ [\] = \text{tt}
\]
\[
\text{funOfLine} \ (bps) \ (\text{el0} \# \text{ls}) = \text{AND} \ (\text{funOfLit} \ ((\text{hd} \ bps), \text{el0})) \ (\text{funOfLine} \ (\text{tl} \ bps) \ \text{ls})
\]

\[
\text{funOfTab} :: \text{PLA} \Rightarrow \text{boolPairs list} \Rightarrow \text{boolPairs}
\]
\[
\text{primrec funOfTab} \ [] = \text{ff}
\]
\[
\text{funOfTab} \ (\text{line} \# \text{tbl}) \ bps = \text{OR} \ (\text{funOfLine} \ \text{bps} \ \text{line}) \ (\text{funOfTab} \ \text{tbl} \ bps)
\]

Now we formally introduce a so-called closure relation \textit{rclosure}, which is defined on a netlist and a state. \textit{rclosure} \( nl \ s \) returns the closure set of information propagated forwards in the simulation of the netlist \( nl \) at the state \( s \), and formally is a pair set and inductively defined as follows:

\[
\text{consts rclosure} :: \text{entity set} \Rightarrow \text{state} \Rightarrow (\text{node} \times \text{boolPairs}) \text{ set}
\]
\[
\text{inductive rclosure} \ nl \ s \\
\text{intros}
\text{stAddInput} : \\
[\text{Input} \ n \in \ nl] \implies (n, s n) \in \text{rclosure} \ nl \ s
\text{stAddLatch} : \\
[\text{Delay} \ n \ \text{inp} \in \ nl] \implies (n, s n) \in \text{rclosure} \ \text{nls}
\text{stAddGate} : \\
[\text{Gate} \ n \ \text{inps} \ \text{tab} \in \ nl; \text{length stateLs} = \text{length inps}; \\
\forall l.(l \in \text{mem tab}) \implies \text{length} \ l = \text{length} \ \text{inps}; \\
\forall \text{pair}. \text{pair mem} (\text{zip} \ \text{inps} \ \text{stateLs}) \implies \text{pair} \in \text{rclosure} \ nl \ s] \\
\implies (n,((\text{funOfTab} \ \text{tab} \ \text{stateLs}) \cup (s n))) \in \text{rclosure} \ nl \ s
\]

The relation \textit{rclosure} \( nl \ s \) is corresponding to a function, namely, for any node \( n \) such that \textit{isDefinedIn} \( n \ nl \), there is pair \( p \) such that \( \text{fst} \ p = n \), furthermore, if both \( (n, v_1) \) and \( (n, v_2) \) are in \textit{rclosure} \( nl \ s \), then \( v_1 = v_2 \). Intuitively, \textit{rclosure} \( nl \ s \) is single-valued because output node of a logical entity is uniquely defined and the combination logic of a netlist is acyclic. More formally,
Lemma 6. [isDefinedIn n nl] ⇒ ∃! pair.pair ∈ rclosure nl s ∧ (fst pair) = n

Therefore, we define a function fclosure on a netlist nl and a state s. fclosure nl s returns the result state of nl after simulation at the driving state s.

fclosure nl s n ≡
if isDefinedIn n nl then let pair = (THE pair.pair ∈ rclosure nl s ∧ (fst pair) = n) in (snd pair)
else s

In this definition, if n is defined as an output of a logical entity, then the value of n in the result is the second element of the unique element pair which is in the closure set rclosure nl s and fst pair = n.

Roughly speaking, “a closure function f” means that applying f once can derive a closure of information in some form. In detail, (1) f is monotonic, f x ⊑ f y if x ⊑ y. (2) f is idempotent: f x = f (f x); (3) f is extensive: x ⊑ f x. Function fclosure nl is a closure function.

Function fclosure is a closure function. More formally, we have

1. [s₁ ⊑ s₂] ⇒ fclosure nl s₁ n ⊑ fclosure nl s₂ n
2. s n ⊑ fclosure nl s n
3. fclosure nl (fclosure nl s) n = fclosure nl s n

Now we show how simulation information is propagated forwards through time given a stimulating sequence σ, i.e., from each time step t to time step t + 1. Recall that each delay has an output node data_## and input node data. For the delay, the value of node data at time point t is denoted as data_t after the simulation at time t, and the information data will be propagated to node data_## at time t + 1, i.e., the simulator initially sets the value of node data_## at time point t + 1 as the upper bound of data and σ(t+1)(data_##) , then starts the simulation over the circuit at time point t + 1. In order to model this forwards propagation of information through time, we define a function of over a logical entity and time fSeq nl σ, which returns a result sequence after simulation of nl given a stimulating sequence σ. fSeq nl σ is another sequence and defined as a primary recursion on time t basing on the definition of fclosure. In the following discussion, we use isDelayName x nl to denote that x is an output node of a delay in the netlist nl.

fSeq nl σ 0 = fclosure nl (σ 0)
fSeq nl σ (t + 1) = (let s =
(λn. if (isDelayName n nl)
then (let l = (lookUp nl n) in
let ins = fanins l in
((fSeq nl σ t) (hd ins)) ∪ (σ (t + 1) n))
else σ (t + 1) n)
in fclosure nl s)

Similarly, we also can prove that fSeq is also a closure function, namely,
1. \[ [nl \in \text{netlists}; \text{isClosed} nl; sq_1 \sqsubseteq sq \sqsupseteq sq_2] \quad \Rightarrow \quad \text{fSeq} nl \sqsubseteq sq_1 \sqsubseteq sq_2 \text{fSeq} nl \sqsubseteq sq_2 \]

2. \[ [nl \in \text{netlists}; \text{isClosed} nl] \quad \Rightarrow \quad sq \sqsubseteq sq \text{fSeq} nl \sqsubseteq sq \]

3. \[ [nl \in \text{netlists}] \quad \Rightarrow \quad \text{fSeq} nl \sqsubseteq (\text{fSeq} nl \sqsubseteq sq) = \text{fSeq} nl \sqsubseteq sq \]

5.0.1. Trajectories

A trajectory is a result state sequence of some circuit netlist \( nl \) after a run of simulation. It is a sequence in which no more information can be derived by forwards propagation. Namely, the result sequence returned by a simulation run of \( nl \) is the same as the stimulating sequence fed into the simulator. We define \( \text{trajOfCirc} \) \( nl \) as the set of all trajectories of a netlist \( nl \):

\[
\text{trajOfCirc} :: \text{entity set} \Rightarrow \text{stateSeq set}
\]

\[
\text{trajOfCirc} \equiv \{ \text{sq}. \text{fSeq} nl \sqsubseteq \text{sq} \}
\]

6. Semantics of STE

Now we define the semantics of a STE assertion \( A \sim C \), where both \( A \) and \( C \) are trajectory formulas. \( A \) is called the antecedent, which specifies with what values we should drive the simulation. \( C \) is called the consequent, which specifies the expected results of the simulation. A circuit \( nl \) satisfies a trajectory assertion, written \( \text{cktSat} \) \( nl \; A \sim C \), if for every trajectory \( \tau \) of \( nl \), it holds that \( \tau \models A \) implies \( \tau \models C \).

We define a type assertion to formalize the syntax of a STE assertion.

\[
\text{datatype assertion} = \quad \text{Leadsto trajForm trajForm (infixr \sim 50)}
\]

We introduce a predicate \( \text{cktSat} \) that checks the validity of a STE assertion.

\[
\text{cktSat} :: \text{entity set} \Rightarrow \text{assertion} \Rightarrow \text{bool}
\]

\[
\text{primrec cktSat} \; nl \; (A \sim C) = \quad (\forall \tau. \tau \in (\text{trajOfCirc} \; nl) \quad \Rightarrow \quad (\tau \models A \quad \Rightarrow \quad \tau \models C))
\]

The key feature of STE logic is that there is a unique weakest sequence that satisfies \( f \) for any boolean symbolic variable assignment \( \phi \). This sequence is called the defining sequence of \( f \). To define the defining sequence of a formula, we introduce a primary recursive function \( \text{defSqOfTF} \) which operates on a trajectory formula, and returns a symbolic sequence.

**Definition 7 (Defining Sequence).** Given a trajectory formula \( A \), the defining sequence of \( A \), written \( \text{defSqOfTrForm} \; A \), is defined as a primary recursive function on type \( \text{trajForm} \).
In the above definition of defSqOfTrForm, \( \rightarrow \) denotes the implication operator in Boolean domain in the case of guard trajectory formula.

From the definition of the defining sequence of \( A \), we can easily prove that the sequence satisfies \( A \) by induction.

**Lemma 8.** \( \text{defSqOfTrForm} \ A \models A \)

Furthermore, for any sequence \( \sigma \) that satisfies \( A \), the defining sequence is the weakest of all.

**Lemma 9.**

1. \( \text{defSqOfTrForm} \ A \sqsubseteq \sigma \models A \rightarrow \sigma \models A \).
2. \( \sigma \models A \models \text{defSqOfTrForm} \ A \sqsubseteq \sigma \sqsubseteq A \)

Now we introduce the defining trajectory of trajectory formula \( A \) w.r.t. \( nl \), which is the weakest trajectory that satisfies \( A \). The defining trajectory of \( A \) w.r.t. \( nl \) is naturally the result sequence by driving \( nl \) with the defining sequence of \( A \).

**Definition 10 (Defining Trajectory).** Given a trajectory form \( A \), a netlist \( nl \), the defining trajectory of \( A \) w.r.t. \( nl \), denoted by \( \text{defTrajOfCirc} \ A \ nl \), is defined as follows:

\[
\text{defTrajOfCirc} \ A \ nl \equiv \text{fSeq} \ nl \ (\text{defSqOfTrForm} \ A)
\]

Similarly, we can prove that a defining trajectory of \( A \) w.r.t. \( nl \) satisfies \( A \).

**Lemma 11.** \( (\text{defTrajOfCirc} \ A \ nl) \in \text{trajOfCirc} \ nl \land (\text{defTrajOfCirc} \ A \ nl) \models A \)

The following lemma proves that the defining trajectory of \( nl \) is indeed the weakest trajectory of \( nl \) that satisfies \( A \).

**Lemma 12.**

1. \( [\tau \in \text{trajOfCirc} \ nl; \ \tau \models A] \models (\text{defTrajOfCirc} \ A \ nl) \sqsubseteq sq \ \tau \)
The following lemma is the most fundamental result of STE theory, which states that \( \text{defSqOfTrForm } C \sqsubseteq \text{defTrajOfCirc } A nl \) if and only if \( \text{cktSat } nl \ (A \sim C) \), for a closed netlist \( nl \). This result guarantees an effective way to check validity of a STE assertion. In order to check an STE assertion \( \text{cktSat } nl \ (A \sim C) \), we only need consider whether \((\text{defSqOfTrForm } C) \sqsubseteq (\text{defTrajOfCirc } A nl)\) holds.

Lemma 13.

(1) \( [(\text{defSqOfTrForm } C) \sqsubseteq (\text{defTrajOfCirc } A nl)] \implies \text{cktSat } nl \ (A \sim C) \)

Proof. In order to prove \( \text{cktSat } nl \ (A \sim C) \), we need fix a trace \( tr \) such that (a) \( tr \in \text{trajOfCirc } nl \) and \( tr \models A \), and we need prove that \( tr \models C \). By lemma 9 (1), we only need prove that \( \text{defSqOfTrForm } C \sqsubseteq tr \). From (a), by lemma 12 (1), we have \((\text{defTrajOfCirc } A nl) \sqsubseteq tr \). From the assumption \((\text{defSqOfTrForm } C) \sqsubseteq (\text{defTrajOfCirc } A nl)\), and the transitivity of \( \sqsubseteq \), we have \( \text{defSqOfTrForm } C \sqsubseteq tr \).

(2) \( [\text{cktSat } nl \ (A \sim C)] \implies (\text{defSqOfTrForm } C) \sqsubseteq (\text{defTrajOfCirc } A nl) \)

Proof. By lemma 11, we have \((\text{defTrajOfCirc } A nl) \in \text{trajOfCirc } nl \) and \((\text{defTrajOfCirc } A nl) \models A \). From this, by the definition of \( \text{cktSat } nl \ (A \sim C) \), we have (a) \((\text{defTrajOfCirc } A nl) \models C \). By lemma 9 (2), we easily show \( \text{defSqOfTrForm } C \sqsubseteq (\text{defTrajOfCirc } A nl) \).

7. Sub-netlists

It is interesting to note that the evaluation of a STE assertion in a netlist may be only related with a part of the netlist, and this part is also a netlist itself. Therefore, we introduce the concept of sub-netlist, given two logical entity sets \( nl \) and \( nl' \), usually \( nl' \subseteq nl \), a sub-netlist derived from \( nl' \) in \( nl \) is an closure set of entities which is defined as follows:

**Definition 14.** Let \( nl, nl' \) be two set of devices, a sub-netlist closure function \( \text{subNet } nl nl' \), which is an inductively defined set by the following rules:

\[
\begin{align*}
\text{consts} & \quad \text{subNet} :: \text{entity set} \Rightarrow \text{entity set} \Rightarrow \text{entity set} \\
\text{inductive} & \quad \text{subNet } nl nl' \\
\text{intros} & \quad \text{subAddSelf} : [\\text{entr} \in nl'; \text{entr} \in nl ] \implies \text{entr} \in \text{subNet } nl nl' \\
& \quad \text{subAddLink} : [\\text{entr}_0 \in \text{subNet } nl nl'; \text{entr}_1 \in nl; \\
& \quad (\text{fanout} (\text{entr}_1) \in \text{set} (\text{fanins} (\text{entr}_0))) ] \implies \text{entr}_1 \in \text{subNet } nl nl' \\
\end{align*}
\]

In the rule \( \text{subAddLink} \), \( (\text{fanout} (\text{entr}_1) \in \text{set} (\text{fanins} (\text{entr}_0))) \) means that the output node of \( \text{entr}_1 \) is driving one input node of \( \text{entr}_0 \). This rule
guarantees that all the fanin cones of entities in \( nl' \) is defined in \( \text{subNet} \; nl \; nl' \). Obviously, it holds that \( \text{subNet} \; nl \; nl' \subseteq nl \) for any \( nl' \). Infomally we call \( nl_1 \) is a sub-netlist of \( nl \) if \( nl_1 = \text{subNet} \; nl \; nl_0 \) for some \( nl_0 \).

**Example 15.** In Example 2, let \( nl' = \{ \text{xnor}G_0 \} \), \( \text{subNet} \; nl \; nl' = \{ \text{Input} \; a_0, \; \text{Input} \; b_0, \; \text{xnor}G_0 \} \).

Supposed that \( nl' \) is a sub-netlist of \( nl \). At a time point, if \( n \) is a node defined in \( nl' \), then the same value will be propagated into node \( n \) after simulations for \( nl \) and \( nl' \) respectively from a state \( s \).

**Lemma 16.**

1. \[ [nl' \subseteq nl; \text{isDefinedIn} \; n \in nl'] \implies (n, v) \in (\text{rclosure} \; nl' \; s) = (n, v) \in (\text{rclosure} \; nl \; s) \]
2. \[ [nl' \subseteq nl; \text{isDefinedIn} \; n \in nl'] \implies \text{fclosure} \; nl \; s \; n = \text{fclosure} \; nl' \; s \; n \]

Similarly, supposed that \( n \) is defined in \( nl' \), node \( n \) will be updated with the same value at any time point after two simulations for \( nl \) and \( nl' \) from a same state \( s \).

**Lemma 17.**

Using lemma 17, we can prove that two sequences \( \text{defTrajOfCirc} \; B \; nl \) and \( \text{defTrajOfCirc} \; B \; nl' \) agree the same value on a node \( n \) at any time point if \( n \) is defined in \( nl' \).

**Lemma 18.**

\[ [nl' \subseteq nl; \text{isDefinedIn} \; n \in nl'] \implies \text{defTrajOfCirc} \; B \; nl \; t \; n = \text{defTrajOfCirc} \; B \; nl' \; t \; n \]

Provided that \( nl' \) is a sub-netlist of \( nl \), and all the nodes specified in the consequent \( C \) of an STE assertion are defined in \( nl' \), then it can be safely concluded that \( \text{cktSat} \; nl \; A \; \sim \; C \) iff \( \text{cktSat} \; nl' \; A \; \sim \; C \).

**Lemma 19 (subsetI).**

\[ [nl' \subseteq nl; \forall n. n \in (\text{onNodes} \; C) \implies \text{isDefinedIn} \; n \; nl'] \implies \text{cktSat} \; nl' \; A \; \sim \; C = \text{cktSat} \; nl \; A \; \sim \; C \]

The proof of this lemma is rather straightforward. We mainly combine Lemma 18 and lemma 13 to prove this result. The key point is that for any node \( n \in (\text{onNodes} \; C) \), we have that \( \text{defTrajOfCirc} \; A \; nl \; t \; n = \text{defTrajOfCirc} \; A \; nl' \; t \; n \). Therefore, \( (\text{defSqOfTrForm} \; C) \; t \; n \subseteq (\text{defTrajOfCirc} \; A \; nl) \; t \; n \) iff \( (\text{defSqOfTrForm} \; C) \; t \; n \subseteq_{sq} (\text{defTrajOfCirc} \; A \; nl') \; t \; n \) for any \( t \), any node \( n \in (\text{onNodes} \; C) \). We are only interested in the evaluation of nodes \( n \in (\text{onNodes} \; C) \) because \( (\text{defSqOfTrForm} \; C) \; t \; n = X \) for any node \( n \notin (\text{onNodes} \; C) \) and \( X \subseteq v \) for any value \( v \).
We need two preliminary definitions before we continue.

**Definition 20.** Let \( A \) be a trajectory formula, onNodes \( A \), which returns the set of nodes which occur in \( A \), is defined as follows:

\[
onNodes :: \text{trajForm} \Rightarrow \text{node set}
\]

\[
\text{onNodes} (\text{Is1} \ n) = \{n\}
\]

\[
\text{onNodes} (\text{Is0} \ n) = \{n\}
\]

\[
\text{onNodes} (A \ text{ and} \ T \ B) = (\text{onNodes} A) \cup (\text{onNodes} B)
\]

\[
\text{onNodes} (\text{Next} \ A) = \text{onNodes} A
\]

\[
\text{onNodes} \text{ chaos} = \emptyset
\]

Next definition \( \text{InducedNet} nl ns \), where \( nl \) is a netlist and \( ns \) is a node set. \( \text{InducedNet} nl ns \) return a sub-netlist which includes the logical entities which has a node in \( ns \) as an output node.

**Definition 21.**

\[
\text{InducedNet} :: \text{entity set} \Rightarrow \text{node set} \Rightarrow \text{entity set}
\]

\[
\text{InducedNet} nl ns \equiv \text{subNet} nl \{g.\exists n.\text{isDefinedIn} n nl \land n \in ns \land g = \text{lookUp} nl n\}
\]

The next lemma says that if an antecedent \( B \) has nothing to do with nodes which may affect the nodes in the consequent \( C \), more specifically, \((\text{onNodes} B) \cap \text{defAsOuts} (\text{InducedNet} nl (\text{onNodes} C)) = \emptyset\), then \( B \) has nothing with the truth of this assertion.

**Lemma 22 (steEqAnt).**

\[
\begin{align*}
\forall n.n \in (\text{onNodes} C) \rightarrow \text{isDefinedIn} n nl & \Rightarrow \text{cktSat} nl (A \text{ and} \ B) \sim C
\end{align*}
\]

For instance, let \( A = (\text{Is1} \ a_0) \text{ and} \ T (\text{Is1} \ b_0) \), \( B = (\text{Isb} a_1 B a_1) \text{ and} \ T (\text{Isb} b_1 B b_1) \), \( C = \text{Is1} c_0, nl \) be the netlist as shown Fig. 1, \( nl' = \text{InducedNet} nl (\text{onNodes} C) \), then we have \( \text{onNodes} B = \{a_1, b_1\}, \text{onNodes} C = \{c_0\}, nl' = \{\text{Input} a_0, \text{Input} b_0, \text{norG} c_0\} \), because \((\text{onNodes} B) \cap \text{defAsOuts} nl' = \emptyset\), \text{cktSat} nl (\(A \text{ and} \ B) \sim C\) is equivalent to \text{cktSat} nl A \sim C. Usually the \((A \text{ and} \ B) \sim C\) has more symbolic variables than \(A \sim C\) does, so we often use the following law which tells us the heuristics to simplify an assertion by eliminating unnecessary antecedents.

**Lemma 23 (steDelAnt).**

\[
\begin{align*}
\text{cktSat} nl (A \text{ and} \ B) \sim C & \Rightarrow \text{cktSat} nl A \sim C
\end{align*}
\]

This result tells us the heuristics to simplify an assertion by eliminating some unnecessary antecedents without affecting the truth of the assertion under study.
8. Symmetry in Circuit Structure and STE

In this section, we introduce the concept of structure symmetry. Due to formalization on the structure of circuits, it is rather straightforward to formalize structure symmetry.

**Definition 24.** Let $nl$ and $nl'$ be two closed netlists, $nl$ and $nl'$ are symmetric w.r.t. a function $f$, written by $\text{sym} \ nl \ nl' \ f$, which is defined as follows:

\[
\text{sym} 
\text{:: (node => node) => entity set => entity set => bool}
\]

\[
\text{sym} \ f \ M \ N \equiv \text{bij} \ f \land (\text{defAsOuts} M) = (\text{defAsOuts} N) \land \\
(\forall m.\text{isDefinedIn} m \ M \rightarrow \text{isDefinedIn} (f m) \ N) \land \\
(\text{let} lx = (\text{lookUp} M m) \ \text{in} \\
\text{let} ly = (\text{lookUp} N (f m)) \ \text{in} \\
(\text{case} (lx) \ of \\
\text{Input} x \Rightarrow ly = \text{Input} (f x) | \\
\text{Delay out data} \Rightarrow ly = \text{Delay} (f out) (f data) | \\
\text{Gate out inps tab} \Rightarrow \\
ly = \text{Gate} (f out) (\text{map} f \text{ inps} \text{ tab})))
\]

Roughly speaking, $\text{sym} \ f \ nl \ nl'$ says that $f$ is an isomorphism mapping from the structure of $nl$ to that of $nl'$. Namely, if $n$ is an output of a logical entity $l$ in $nl$, then $f n$ is an output of a similar logical entity $l'$ and the fanins of $l$ is also mapped to those of $l'$ under $f$. Informally, $l$ and $l'$ are similar in the sense that they are both input devices, or both delays, or both gates with the same truth table.

Usually we need discuss the symmetry between two nodes in one netlist, which is defined by symmetry between sub-netlists induced by the two node sets. The predicate $\text{nodeSetSym} f M N nl$ specifies that the subnetlists induced from node sets $M$ and $N$ in an entity set $nl$ are symmetric w.r.t. some function $f$. Informally, we call that node set $M$ and $N$ are symmetric in $nl$ w.r.t $f$.

**Definition 25.**

\[
\text{nodeSetSym} :\ (\text{node => node}) => \text{node set => node set => entity set => bool}
\]

\[
\text{nodeSetSym} \ f \ M \ N \ nl \equiv \text{sym} \ f \ (\text{InducedNet} nl M) \ (\text{InducedNet} nl N)
\]

**Example 26.** Let $nl_0 = \{\text{Input} a_0, \text{Input} b_0, \text{xnorG}_0\}$, $nl_1 = \{\text{Input} a_1, \text{Input} b_1, \text{xnorG}_1\}$, $N_0 = \{c_0\}$, $N_1 = \{c_1\}$, and $f = \lambda x.\{(\text{if} \ x = a_0 \ \text{then} \ a_1 \ \text{else} \ x = a_1, \text{then} \ a_0 \ \text{else} \ x = a_0 \ \text{then} \ a_1 \ \text{else} \ x = a_1, \text{then} \ a_0 \ \text{else} \ x = b_0 \ \text{then} \ b_1 \ \text{else} \ x = b_1 \ \text{then} \ b_0 \ \text{else} \ x = c_0 \ \text{then} \ c_1 \ \text{else} \ x = c_1 \ \text{then} \ c_0 \ \text{else} \ x)\}$. $\text{InducedNet} \ nl \ N_0 = nl_0$, $\text{InducedNet} \ nl \ N_1 = nl_1$. We have that $\text{sym} \ nl_0 \ nl_1 \ f$ and $\text{nodeSetSym} \ f \ N_0 \ N_1 \ nl$.

Next we define permutations on states, sequences, and formulas. These are similar to their counterparts in [17].
Definition 27. Permutation on states.
\[ \text{appSym2State :: (node } \Rightarrow \text{ node) } \Rightarrow \text{ state } \Rightarrow \text{ state} \]
\[ \text{appSym2State } f \ s = \lambda n.\ s (f n) \]

Definition 28. Permutation on sequences.
\[ \text{appSym2Seq :: (node } \Rightarrow \text{ node) } \Rightarrow \text{ stateSeq } \Rightarrow \text{ stateSeq} \]
\[ \text{appSym2Seq } f \ sq \equiv \lambda t.\ \text{appSym2State } f \ (sq \ t) \]

Definition 29. Permutation on formulas.
\[ \text{applySym2Form :: (node } \Rightarrow \text{ node) } \Rightarrow \text{ trajForm } \Rightarrow \text{ trajForm} \]
primrec
\[ \text{appSym2Form } f \ (\text{Is0 } n) = \text{Is0 } (f n) \]
\[ \text{appSym2Form } f \ (\text{Is0 } n) = \text{Is1 } (f n) \]
\[ \text{appSym2Form } f \ (A \text{ and } B) = \text{(appSym2Form } f \ A) \ \text{and} \ (\text{appSym2Form } f \ B) \]
\[ \text{appSym2Form } f \ (P \rightarrow T \ A) = P \rightarrow T \ (\text{appSym2Form } f \ A) \]
\[ \text{appSym2Form } f \ (\text{Next } A) = \text{Next } (\text{appSym2Form } f \ A) \]
\[ \text{appSym2Form } f \ \text{chaos} = \text{chaos} \]

Each permutation can be defined in terms of a composition of swap functions. Here we use a predicate isSwap to specify that a function is a swap function:
\[ \text{isSwap } f \equiv \forall a \ b.\ f a = b \rightarrow f b = a. \]
It is equivalent to apply a swap permutation \( f \) on a defining sequence of a formula and to compute the defining sequence of the permutation of a formula, provided that \( f \) is a swap function.

Lemma 30.
\[ \text{isSwap } f \Rightarrow \text{appSym2Seq } f \ (\text{defSqOfTrForm } A) = \text{defSqOfTrForm } (\text{appSym2Form } f \ A) \]

Suppose that \( nl \) and \( nl' \) are symmetric w.r.t. \( f \), then a swap permutation on the defining trajectory of \( A \) w.r.t. \( nl \) is equivalent to the defining trajectory of \( \text{appSym2Form } f \ A \) w.r.t. \( nl' \).

Lemma 31.
\[ [nl \in \text{netlists}; \ nl' \in \text{netlists}; \ \text{sym } f \ nl \ nl'; \text{isSwap } f] \]
\[ \Rightarrow \text{appSym2Seq } f \ (\text{trajOfCirc } A \ nl) = \text{trajOfCirc } (\text{appSym2Form } f \ A) \ nl' \]

With the help of Lemma 13 and 30 and 31, we can derive an important result which encapsulates the relation between symmetric netlists and the symmetric STE assertions.

Lemma 32.
\[ [\text{sym } f \ nl \ nl'; \text{isSwap } f; ] \Rightarrow \]
\[ \text{cktSat } nl \ (A \sim C) = \text{cktSat } nl' \ (\text{appSym2Form } f \ A \sim \text{appSym2Form } f \ C) \]
This result guarantees us that we only need verify one representative STE assertion from an equivalence class, and deduce the correctness of the entire class for symmetric circuits.

Provided that all the nodes in `onNodes C` and `onNodes (f C)` are defined in `nl`, and they are symmetric in `nl` w.r.t. `f`, `cktSat nl (A \sim C)` implies `cktSat nl (appSym2Form f A \sim appSym2Form f C)`. The proof of this result needs the combination of Lemma 32 and Lemma 19. Because we often meet the case of symmetry between two subnetlists in a netlist, the following lemma is very useful in our verification.

Lemma 33 (symReduce2).

\[
\begin{align*}
\text{isSwap } f; &\forall n. n \in (\text{onNodes } C) \rightarrow \text{isDefinedIn } n \text{ } nl; \\
\forall n. n \in (\text{onNodes } (\text{appSym2Form } f C)) \rightarrow \text{isDefinedIn } n \text{ } nl; \\
\text{nodeSetSym } f \rightarrow (\text{onNodes } C) \rightarrow (\text{onNodes } (\text{appSym2Form } f C)) \rightarrow nl \Rightarrow \\
\text{cktSat } nl \ (A \sim C) = \text{cktSat } nl \ (\text{appSym2Form } f A \sim \text{appSym2Form } f C)
\end{align*}
\]

9. Novel Algebraic Laws

In this section, we introduce a set of algebraic laws. The novelty of our laws lies in that they relate properties of some circuits with their special structures. In the classical literature of STE, some laws have already been introduced, and they usually are general in the sense that they are independent in the structures of circuits. For instance, the `steConjI` rule, \([ nl \in \text{netlists}; \text{isClosed } nl; \text{cktSat } nl (A \sim B); \text{cktSat } nl (A \sim C) \Rightarrow \text{cktSat } nl (B \text{ and } C) \]`, has already been introduced in [1, 17], and it holds for any netlist `nl`. Different from their laws such as `steConjI`, our laws, which are introduced below, formally explore the special structures of some circuits in our formal netlist model.

We need some preliminary definitions before we continue. `andFormLists tfs` returns the conjunction of a list of trajectory formulas:

\[
\begin{align*}
\text{andLists } [] & = \text{chaos} \\
\text{andLists } (A \# \text{listA}) & = A \text{ and } \text{andLists } \text{listA}
\end{align*}
\]

Two predicates `isFullAndLine :: \text{LINE} \Rightarrow \text{bool}` and `isAndTab :: \text{PLA} \Rightarrow \text{bool}` are introduced to define a truth table of an AND-gate:

\[
\begin{align*}
isFullAndLine \text{line} & \equiv \forall l. l \in \text{line} \rightarrow l = \text{ONE} \\
isAndTab \text{tab} & \equiv \text{length } \text{tab} = 1 \land \text{isFullAndLine} (\text{hd } \text{tab})
\end{align*}
\]

The first lemma says that if all the input nodes of an AND-gate are set high, then its out should be high too.

Lemma 34 (andTabPropT).

\[
\begin{align*}
\text{isAndTab } \text{tab}; \text{Gate } \text{out } \text{inps } \text{tab} \in \text{nl}; \\
\forall l. (l \in \text{mem } \text{tab}) \rightarrow \text{length } l = \text{length } \text{inps} \Rightarrow \\
\text{cktSat } \text{nl } ((\text{andLists } (\text{map } (\lambda n. \text{Is1 } n) \text{inps})) \sim (\text{Is1 } \text{out}))
\end{align*}
\]
The second lemma says that if one input node of an and-gate are set low, then its out turns low.

Lemma 35 (andTabPropF).

\[
\text{[isAndTab tab; Gate out inps tab } \in \text{ nl; inps, mem inps; } \\
\forall l (l \text{ mem tab} \implies \text{length } l = \text{length inps}] \implies \\
\text{cktSat } nl (\text{Is0 inps}) \leadsto (\text{Is0 out})
\]

Naturally a table, whose length is greater than 1, is a disjunction of lines. We need not deliberately define an OR-gate. However, we need formally define a function which specifies value assignments of all inputs in a line before we go on. The function \text{posAssertOfLine inps lit} returns a list of trajectory formulas, each of which specifies a special value of each node according to the literal \text{lit}. If \text{lit} is ZERO, then \text{inps} is specified as \text{ff} by an \text{Is0} formula, else if \text{lit} is ONE, then \text{inps} is specified as \text{tt} by an \text{Is1} formula, otherwise it is set as X by chaos. Let \text{inps} = [i_1, i_2], \text{line} = [\text{ONE, ONE}], then \text{posAssertOfLine inps line} = [\text{Is1 i_1}, \text{Is1 i_2}].

\[
\text{posAssertOfLine :: node list } \Rightarrow \text{ Literal list } \Rightarrow \text{ trajForm list}
\]

\[
\text{primrec}
\text{posAssertOfLine inps } [] = []
\text{posAssertOfLine inps (l#line) =}
\text{let otherAss = posAssertOfLine (tl inps) line in}
\text{(case l of ZERO } \Rightarrow (\text{Is0 (hd inps)})#\text{otherAss}| \\
\text{ONE } \Rightarrow (\text{Is1 (hd inps)})#\text{otherAss}|
\text{DONTCARE } \Rightarrow \text{chaos#otherAss})
\]

Obviously, if there exists a line \text{l} in the table \text{tab} of a gate, and the values assigned to the inputs of the gate satisfy the formula \text{posAssertOfLine inps l}, then the output of the line is \text{tt}, thus the output of the gate is also set \text{tt}.

Lemma 36 (orTabPropT).

\[
\text{[Gate out inps tab } \in \text{ nl; l mem tab; } \\
\forall l (l \text{ mem tab} \implies \text{length } l = \text{length inps}] \implies \\
\text{cktSat } nl (\text{andLists (posAssertOfLine inps l)}) \leadsto (\text{Is1 out})
\]

Next we introduce a function \text{isNegAssOfLine A line inps}, the function returns true if a formula \text{A} specifies a proper value for some node \text{inps}, according to the literal \text{lit}. If \text{A} is \text{Is1 n}, then the literal is \text{ZERO}, else if \text{A} is \text{Is1 n}, then the literal is \text{ONE}. For simplicity, \text{isNegAssOfLine A line inps} is defined to be False for any other formula.
isNegAssOfLine :: trajForm ⇒ node list ⇒ Literal list ⇒ bool
primrec
isNegAssOfLine (Is1 n) inps line = n mem inps ∧
∃ pair. (pair ∈ zip inps line ∧ fst pair = n ∧ snd pair = ZERO)
isNegAssOfLine (Is0 n) inps line = n mem inps ∧
∃ pair. (pair ∈ zip inps line ∧ fst pair = n ∧ snd pair = ONE)
isNegAssOfLine A inps line = False,
for any other formula A

For a trajectory formula list asList, for any line l in the table tab of a
gate, it holds that there exists a formula A which is a member of asList and
isNegAssOfLine A line inps, then the value of the output of each line is ff, thus
the output of the gate is set ff. For instance, let tab = [[ONE, ONE], [ZERO, ZERO]],
and asList = [Is1 i1, Is0 i2], we have ∃ A. (A mem asList) ∧ isNegAssOfLine A
inps l for any l such that l mem tab.

Lemma 37 (orTabPropF).

[Gate out inps tab ∈ nl; ∀l.(l mem tab) → length l = length inps;
∀l.(l mem tab) → (∃ A. (A mem asList) ∧ isNegAssOfLine A inps l ) ] ⇒
cktSat nl (andLists asList) ~> (Is0 out))

For convenience, we define a syntactical abbreviation: lsb n a ≡ (a →> Is1 n)
and (¬a →> Is0 n). Roughly speaking, lsb n a means that node n is set a
boolean value a. If an input node n of a delay is set a boolean value a at time
0, then the output of the delay will be set a at the next time point.

Lemma 38.

[Delay out data ∈ nl; nl ∈ netlists; isClosed nl] ⇒
cktSat nl (lsb n a) ~> Next (lsb out a))

10. Illustrative Case Studies

In this section, we use illustrative examples to demonstrate the power of
our new laws. We choose content addressable memories (CAMs), which is a
classical example used in STE literature. CAMs are widely used wherever fast
parallel search operations are required. Pandey used symbolic indexing tech-
niques to verify CAMs, which is regarded as a classical work in STE literature
[28]. He reported a logarithmic reduction in the number of variables required
if the symbolic indexing encoding style is adopted. Darbari took advantage of
a type-checking approach for symmetry detection based on a high-level HDL
description, where he used a richer type system to record the symmetry [9, 17].
Using the symmetry type information, he combined symmetry reduction with
other decomposition rules. CAMs could be verified using a fixed number of
BDD variables since he only had to verify one line at a time, and the other lines
can be verified by symmetry reduction. The amount of time used in verification is linear with respect to the tag width, number of CAM lines and the number of CAMs.

The structure and property of a CAM circuit is rather complex, and the core of a CAM is a list of comparators whose outputs are driving an OR-gate. So we start from a N-bits comparator.

10.1. N-bits Comparator

The structure of a N-bits comparator is a natural extension of 2-bits comparator, which is shown in Figure 1. For convenience, we need define some syntactical abbreviation: \([0.. < N] \equiv [0, ..., N - 1]\) if \(N > 0\). Let \(f\) be a function over natural number, \([f \ i. \ i < N] \equiv map \ f \ [0.. < N]\). In this work, we usually call such \(f\) a vector, \(f \ i\) is denoted by \(f_i\). If \(f_i\) is still a vector, we write \(f_i j\) for \(f \ i \ j\).

Let \(a, b, c\) be three vectors of nodes. \(i\) be a node. Let \(N > 1\), \(\texttt{xnorTab} = [[[\texttt{ONE, ONE}], \texttt{ZERO, ZERO}]], \texttt{andLine} = [[[i \ i < N]], \texttt{xnorGLs} = \{\texttt{Gate} \ e_i [\texttt{a}, \texttt{b}] \texttt{xnorTab}, i < N\}, cs = [c, i < N], \texttt{andG} = \texttt{Gate out cs [andLine]}\). Let \(nl\) be a closed netlist such that \(\texttt{xnorGLs} \cup \{\texttt{andG}\} \subseteq \texttt{nl}\). To make our results more general, we only require that \(nl\) has the gate \(\texttt{andG}\) and all the XNOR-gates in \(\texttt{xnorGLs}\).

Let \(\texttt{bvOfAs}\) and \(\texttt{bvOfBs}\) be two vectors of boolean variables to model symbolic values of nodes, \(\texttt{bvOfAs}_i\) is a boolean variable. \(\texttt{antOfAs} = \{\texttt{lsb} \ a_i, \texttt{bvOfAs}_i, i < N\}, \texttt{antOfBs} = \{\texttt{lsb} \ b_i, \texttt{bvOfBs}_i, i < N\}, \texttt{Gp}_0 = \exists i. i < N \land \texttt{bvOfAs}_i \neq \texttt{bvOfBs}_i, \texttt{Gp}_1 = \forall i. i < N \implies \texttt{bvOfAs}_i = \texttt{bvOfBs}_i\). Let \(\texttt{ant} = \texttt{andLists} (\texttt{antOfAs} \texttt{xor} \texttt{antOfBs})\), \(\texttt{consS}_0 = \texttt{Gp}_0 \implies \texttt{ls0 out}, \\texttt{consS}_1 = \texttt{Gp}_1 \implies \texttt{ls1 out}, \\texttt{cons} = \texttt{consS}_0 \texttt{and} \texttt{consS}_1\). Here we want to prove an assertion \(\texttt{cktSat} \ \texttt{nl} (\texttt{ant} \sim \texttt{cons})\). Intuitively, \(\texttt{ant}\) specifies the symbolic values of the nodes to be compared, \(\texttt{consS}_0\) says that \(\texttt{out}\) is low when \(a\) and \(b\) do not agree on a bit \(i\), and \(\texttt{consS}_1\) says \(\texttt{out}\) is high when \(a\) and \(b\) agree on all bits \(i < N\). Due to space limitation, we only give key auxiliary results for the main lemma. Refer to the Isabelle proof scripts [11] for the details.

Lemma 39.

1. \([i < N; \sim \texttt{bvOfAs}_i \land \sim \texttt{bvOfBs}_i] \implies \texttt{cktSat} \ \texttt{nl} (\texttt{ant} \sim \texttt{andLists} \ [\texttt{ls0} \ a_i, \texttt{ls1} \ b_i])\)
2. \([i < N; \texttt{bvOfAs}_i \land \sim \texttt{bvOfBs}_i] \implies \texttt{cktSat} \ \texttt{nl} (\texttt{ant} \sim \texttt{andLists} \ [\texttt{ls1} \ a_i, \texttt{ls0} \ b_i])\)
3. \([i < N] \implies \texttt{cktSat} \ \texttt{nl} (\texttt{andLists} \ [\texttt{ls0} \ a_i, \texttt{ls1} \ b_i]) \sim \texttt{ls0} \ c_i\)
4. \([i < N] \implies \texttt{cktSat} \ \texttt{nl} (\texttt{andLists} \ [\texttt{ls1} \ a_i, \texttt{ls0} \ b_i]) \sim \texttt{ls0} \ c_i\)
5. \([i < N; \texttt{bvOfAs}_i \neq \texttt{bvOfBs}_i] \implies \texttt{cktSat} \ \texttt{nl} (\texttt{ant} \sim \texttt{ls0} \ c_i)\)
6. \([i < N] \implies \texttt{cktSat} \ \texttt{nl} \ \texttt{ls0} \ c_i \sim \texttt{ls0} \ \texttt{out}\)
7. \([i < N; \texttt{bvOfAs}_i \land \texttt{bvOfBs}_i] \implies \texttt{cktSat} \ \texttt{nl} (\texttt{ant} \sim (\texttt{andLists} (\texttt{posAssertOfLine} \ [a_i, b_i] \ [\texttt{ONE, ONE}])))\)
(8) \[ i < N \implies \text{cktSat} \ nl \]
\[(\text{andLists} (\text{posAssertOfLine} [a_i, b_i] [\text{ONE}, \text{ONE}]) \sim \text{ls1} \ c_i) \]
\[(\text{andLists} \ (\text{posAssertOfLine} [a_i, b_i] [\text{ZERO}, \text{ZERO}])) \]

(9) \[ i < N; \neg \text{bvOfAs}_i \land \neg \text{bvOfBs}_i \implies \text{cktSat} \ nl \ \text{ant} \sim \]
\[(\text{andLists} \ (\text{posAssertOfLine} [a_i, b_i] [\text{ZERO}, \text{ZERO}])) \sim \text{ls1} \ c_i) \]

(10) \[ i < N \implies \text{cktSat} \ nl \]
\[(\text{andLists} \ (\text{posAssertOfLine} [a_i, b_i] [\text{ZERO}, \text{ZERO}])) \sim \text{ls1} \ c_i) \]

(11) \[ i < N; \text{bvOfAs}_i \equiv \text{bvOfBs}_i \implies \text{cktSat} \ nl \ \text{ant} \sim \text{ls1} \ c_i \]

(12) \[ \text{Gp1} \implies \text{cktSat} \ nl \ \text{ant} \sim (\text{andLists} \ [\text{ls1} \ c_i, i < N]) \]

(13) \[ \text{cktSat} \ nl \ (\text{andLists} \ [\text{ls1} \ c_i, i < N]) \sim \text{ls1} \ \text{out} \]

In Lemma 39, (1)-(5) prove that the value of node \( c_i \) will be set low if there is a bit \( i \) such that nodes \( a_i \) and \( b_i \) are set by different values, and rule \( \text{orTabPropF} \) is the main rule used to prove these results. (6) says that once \( c_i \) is set low, then the output is set low. (6) is proved by law \( \text{andTabPropF} \). (7)-(11) prove that the value of node \( c_i \) will be set high if nodes \( a_i \) and \( b_i \) agree on the value of a bit \( i \) such that \( i < N \), and rule \( \text{orTabPropT} \) is the main rule used to prove these results. From these, (12) can be easily proved. (13) can be proved by law \( \text{andTabPropT} \).

Lemma 40. \( \text{cktSat} \ nl \ (\text{ant} \sim \text{cons}) \).

Proof. For the main goal, we use rule \( \text{steconjI} \) to decompose it two subgoals: (a) \( \text{cktSat} \ nl \ \text{ant} \sim \text{cons}_0 \) and (b) \( \text{cktSat} \ nl \ \text{ant} \sim \text{cons}_1 \).

In order to prove (a), by rule \( \text{stelmpI} \), we assume that (c) \( \text{Gp}_0 \), and need show \( \text{cktSat} \ nl \ \text{ant} \sim \text{ls0} \ \text{out} \). From (c), we obtain \( i \) where \( i < N \) and (d) \( \text{bvOfAs}_i \neq \text{bvOfBs}_i \). From this and Lemma 39 (5), we have (e) \( \text{cktSat} \ nl \ \text{ant} \sim \text{ls0} \ c_i \).

With Lemma 39 (6), by rule \( \text{steTrans} \), we show \( \text{cktSat} \ nl \ \text{ant} \sim \text{ls0} \ \text{out} \).

In order to prove (b), by rule \( \text{stelmpI} \), we assume that (f) \( \text{Gp}_1 \), and need show \( \text{cktSat} \ nl \ \text{ant} \sim \text{ls1} \ \text{out} \). From (f) and Lemma 39 (12), we have (g) \( \text{cktSat} \ nl \ \text{ant} \sim (\text{andLists} \ [\text{ls1} \ c_i, i < N]) \). With Lemma 39 (13), by rule \( \text{steTrans} \), we can show \( \text{cktSat} \ nl \ \text{ant} \sim \text{ls1} \ \text{out} \).

10.2. \( M - N - \text{CAM} \)

Figure 5 shows a part of a \( M - N - \text{CAMs} \) circuit. It stores \( M \) lines of tags, and the width of each tag is \( N \). Let \( T \) and \( c \) be a vector of vectors of nodes, \( T_{ij} \) be a node, \( \text{Tag} \) and \( \text{match} \) be a vector of nodes. Let \( M > 1, N > 1 \), \( \text{xnorTab} \) and \( \text{andLine} \) be defined as in subsection 10.1, \( \text{css} = \{[c_{ij}, j < N], i < M\}, \text{xnorGs} = \{\text{Gate} \ c_{ij} \ [T_{ij}, \text{Tag}]; \text{xnorTab}; j < M, i < M\}, \text{matches} = \{\text{match} \ c_i, i < M\}, \text{andGs} = \{\text{Gate} \ \text{match} \ c_i \ \text{css} \ [\text{andLine}]; i < M\}, \text{orLine} = \\lambda \ i.\ ((j = i) \ \text{then} \ \text{ONE} \ \text{else} \ \text{DontCare}) \ j, j < M\), \( \text{orTab} = \{\text{orLine} i, i < M\}, \text{orG} = \text{Gate} \ \text{hit} \ \text{matches} \ \text{orTab} \). Let \( nl \) be a closed netlist such that \( \text{xnorGs} \cup \ \text{andGs} \cup \{\text{orG}\} \subseteq nl \).
Let $bvOsT$ be a vector of vectors of boolean variables to model symbolic values of stored tags, $bvTag$ is a vector of boolean variables to model the symbolic value of input tag. $antOfT = \forall i.i < M. GpOfUnHit_i = \exists i.i < M. GpOfHit_i = \exists i.i < M. GpOfUnHit_i \rightarrow Is0hit, cons0 = GpOfUnHit_i \rightarrow \neg Is0hit, cons1 = GpOfHit_i \rightarrow Is1hit, cons = cons0 \land cons1$. Here we want to prove an assertion $cktSat nl (ant \sim cons)$. In this assertion, $ant$ still specifies that the symbolic values of the nodes of the input tag and the stored tags, $cons0$ says that the node $hit$ is set low if no line matches the input tag, and $cons1$ says that the node $hit$ is set high if there exists one line which matches the input tag.

**Lemma 41.**

1. $[i < M; GpOfUnHit_i \] \Rightarrow cktSat nl ant \sim Is0match_i$
2. $[GpOfUnHit_i \Rightarrow cktSat nl ant \sim (andLists [Is0match_i, i < M])]$
3. $cktSat nl (andLists [Is0match_i, i < M]) \Rightarrow Is0hit$
4. $[GpOfUnHit_i \Rightarrow cktSat nl ant \sim Is0hit$
5. $[i < M; GpOfHit_i \] \Rightarrow cktSat nl ant \sim Is1match_i$
6. $[i < M \] \Rightarrow cktSat nl (Is1match_i) \sim (andLists (posAssertOfLine matches (orLine i)))$
7. $[i < M \Rightarrow cktSat nl$
   (andLists (posAssertOfLine matches (orLine i)) \sim Is1hit)$
8. $[i < M; GpOfHit_i \] \Rightarrow cktSat nl ant \sim Is1hit$

---

**Figure 5:** A $M–N$ CAM
In Lemma 41, (1) and (2) are simply derived by the results of a \(N\)-bits comparator when its output \(match_i\) is set low, as is shown in Lemma 39. Here the antecedent \(GpOfUnHit_i\) specifies that the value of \(i\)-th stored tag \(T_i\) does not match with that of the input tag \(Tag\). (3) can be proved by law \(orTabPropF\). (4) can be proved by combining (2) and (3). (5) is the result of a \(N\)-bits comparator when its output \(match_i\) is set high, as is shown in Lemma 39. Here the antecedent \(GpOfHit_i\) specifies that the value of \(i\)-th stored tag \(T_i\) matches with that of the input tag \(Tag\). (6) can be simply proved by unfolding the definitions of \(andLists\) and \(posAssertOfLine\). The assertion \((andLists posAssertOfLine matches (orLine i))\) is a list of trajectory formulas in which the \(i\)-th element is \((Is1 match_i)\) and any other one is \(chaos\). (7) can be proved by law \(orTabPropT\). (8) can be proved by combining (5), (6) and (7). From these results, it is rather easy to derive the following result by using rules \(steimpl\), \(steconjl\), and \(steTrans\).

Lemma 42. \(cktSat nl ant \sim cons\)

**Proof.** For the main goal, we use rule \(steconjl\) to decompose it two subgoals: (a) \(cktSat nl ant \sim cons_0\) and (b) \(cktSat nl ant \sim cons_1\).

In order to prove (a), by rule \(steimpl\), we assume that (c) \(GpOfUnHit\), and need show \(cktSat nl ant \sim Is0\;hit\). This can be easily proved by Lemma 41 (4).

In order to prove (b), by rule \(steimpl\), we assume that (f) \(GpOfHit\), and need show \(cktSat nl ant \sim Is1\;hit\). From (f), we can obtain a \(i\) such that \(i < M\) and \(GpOfHit_i\). From this, by Lemma 41 (8), we can easily prove that \(cktSat nl ant \sim Is1\;hit\).

Our proofs are purely algebraic reductions without any symbolic simulation. A distinguishing feature of our approach is the use of laws \(andTabPropT(andTabPropF)\) or \(orTabPropT(orTabPropF)\) to decompose one assertion on the output of an AND-gate or OR-gate to assertions on each branch input node of the gate. This explains why we call the laws the algebraic semantics of STE. Note that any combinational parts of a circuit is combined by AND-gates or OR-gates, therefore, our laws \(andTabPropT(andTabPropF)\) or \(orTabPropT(orTabPropF)\) is proposed for general-purpose in the sense that they can be combined together to analyze any combination parts of a circuit. Second, our proof is a parameterized verification of CAMs, where \(M\) and \(N\) are parameters which are arbitrary positive natural numbers. Based on the results of \(N\)-bits comparator, our parameterized proof is clean deductions which are involved in simple applications of rules \(orTabPropT(orTabPropF)\) and those on quantifiers, and does not suffer from any state explosion problem.

11. Conclusion

The key contribution of our work is to introduce the inductive approach to formalize both the structure and simulation semantics of a netlist. Because the legal structure of a netlist requires the following condition: the conflict between output nodes of two logical entities should be eliminated, and a cycle should not
occur in the combinational part of the netlist, but a cycle is allowed to pass a delay element. It is difficult to simply use a datatype to define the structure because such a cycle exists. The inductive definition of a netlist formally specifies these requirements by a set of intuitive introduction rules.

The inductive approach also provides a satisfying answer to formalize the information propagation through netlist structure in the simulation semantics of a netlist. Essentially such a propagation is a process of value assignments to nodes which spreads from each gate’s inputs to its outputs, and this process is started from the primitive input nodes of the netlists and state-holding nodes of delay entities. The three inductive rules in $\text{rclosure}$ accurately capture the semantics of the information propagation process. Furthermore, we can formally derive function $\text{fclosure}$ and $\text{fSeq}$. Here the function $\text{fSeq}$ can be seen as a concrete version of the abstract next-state Y-function used in classical STE literature. It is sound in the sense that $\text{fSeq}$ is monotonic. Therefore our work not only proves the existence of a special next-state Y-function, but also shows its formal construction by deriving $\text{fSeq}$.

Not only does the inductive approach help us to formally define the structure and simulation semantics of a netlist, but also provides an effective inductive principle to prove useful properties of a netlist. Especially, we use the induction principle to prove two unique-existence results which prove the soundness of the semantical model. The first one says that for any defined node $n$, there is an unique logical entity in the netlist whose output is $n$. The second proves a relation $\text{rclosure \; nl \; s}$ is single-valued, thus the function $\text{rclosure \; nl \; s}$ can be formally induced.

The advantage of introducing a formal netlist model is to explicitly explore the close relation between properties of a circuit and its structure. Two main results of ours are symmetry reduction and a set of novel algebraic laws, and they are introduced to decompose a STE assertion. In our case study, we show how to combine some of our laws for parameterized verification of content addressable memories (CAMs). This experience has demonstrated both theoretical and practical benefits because it provides an alternative effective way - algebraic reduction for STE assertion verification.

In the future, we will extend our research in two directions. (1) We will make our reduction method as automatic as possible. In facts, there is strong heuristics to use some laws. For instance, if the consequent of an assertion specifies that the output node of an AND-gate is set positive value, then rules $\text{steTrans}$ and $\text{andTabPropT}$ should be applied, and a new assertion is introduced to specifies that the values of all the input nodes should also be set positive values if the antecedent of the original assertion holds, as shown in Lemma 40. (2) We look into combining our reduction method with STE model-checking. Using our reduction method, we decompose a complex assertion into small assertions, then use a STE tool like Forte to directly model-check the small assertions. The key to combining the two techniques is to select a proper interface and development environment to integrate them.
References


URL http://www.comlab.ox.ac.uk/tom.melham/pub/Melham-1993-HOL.html


URL http://www.comlab.ox.ac.uk/tom.melham/pub/Grundy-2006-RFL.pdf

A. Isabelle Notations

We briefly present some Isabelle notations and commands used in this work. For more details, we refer to [10].

Types. There are basic types such as bool, the type of truth values - True and False; nat, the type of natural numbers. Standard boolean operators ∧ and ∨ and → are defined as usual. Function types are denoted by ⇒, and product types by ×. Types can also be constructed by type constructors such as list and set. For instance, nat list declares the type of lists whose members are natural numbers.

Terms. Forms of terms used in this paper are rather simple. It is simply a constant or variable identifier, or a function application such as f t, where f is a function of type τ₁ ⇒ τ₂, and t is a term of type τ₁.

Introducing new types. There are three kinds of commands for introducing new types. typedecl name introduces new “opaque” type name without definition; types name = τ introduces an abbreviation name for type τ. datatype command can introduce a recursive data type. A general datatype definition is of the form

\[
datatype (α₁, \ldots, αₙ) = C₁ \; τ₁₁ \ldots τ₁k₁ \; | \; \ldots \; | \; C_m \; τₘ₁ \ldots τₘkₘ
\]

where αᵢ are distinct type variables (the parameters), Cᵢ are distinct constructor names and τᵢⱼ are types. Note that n can be 0, i.e., there is no type parameters in datatype declaration.
**Definition commands.** The `consts` command declares a function’s name and type. The `defs` command gives the definition of a declared function. The `constdefs` command combines the effect of `consts` and `defs`. For instance, the following commands define a square function on `nat`.

Combining a `consts` and `inductive` command, we can give an inductive definition for a set. An inductively defined set `S` is typically of the following form:

```plaintext
consts S :: τ set inductive
intros
rule 1: [ | a₁₁ ∈ S; ...; a₁k₁ ∈ S; A₁1, ..., A₁i₁ | ] ⇒ a₁ ∈ S ...
rule n: [ | a₁n₁ ∈ S; ...; aₙkₙ ∈ S; Aₙ1, ..., Aₙiₙ | ] ⇒ aₙ ∈ S
```

**Lemmas.** Lemmas are presented by the notation `[ [A₁; A₂; ...; Aₙ] ] ⇒ B`, which means that with assumptions `A₁, ..., Aₙ`, we can derive a conclusion `B`.

**B. Other Laws**

In this part, we introduce some other laws which are used in our work. Many of these laws have been introduced in previous STE work. They are general in the sense that they are independent in the structure of a netlist.

The first one is the Reflexivity rule.

**Lemma 43 (steRefl).**

```plaintext
cktSat nl (A ⊳ A)
```

Next is the transitivity rule. It allows us to combine together STE assertions in a transitive way.

**Lemma 44 (steTrans).**

```plaintext
[ cktSat nl (A ⊳ B); cktSat nl (B ⊳ C) ] ⇒ cktSat nl (A ⊳ C)
```

Next rule `steconjI` splits the consequence of a STE assertion into individual conjuncts, which can be verified separately.

**Lemma 45 (steconjI).**

```plaintext
[ cktSat nl (A ⊳ B); cktSat nl (A ⊳ C) ] ⇒ cktSat nl (A ⊳ B and₁ C)
```

Rule `steImpI` takes out the boolean guard `g` in the consequent of a STE assertion, and turns it into a boolean assumption.

**Lemma 46 (steImpI).**

```plaintext
[ g ⇒ cktSat nl (A ⊳ B) ] ⇒ cktSat nl (A ⊳ g −→₁ C)
```

Rule `steEnStrenAnt` says that if `defSqOfTrForm A' ⊑ₚ sq defSqOfTrForm A`, then assertions `A' ⊳ B` implies `A ⊳ B` because the antecedent `A` is stronger than `A'`.

33
Lemma 47 (steEnStrenAnt).

\[
[\text{cktSat} \; nl \; (A' \rightsquigarrow B); \text{defSqOfTrForm} \; A' \sqsubseteq \text{defSqOfTrForm} \; A] \implies \text{cktSat} \; nl \; (A \rightsquigarrow B)
\]

Rule steWeakenCons says that if defSqOfTrForm \( B \sqsubseteq \text{defSqOfTrForm} \; B' \), then assertions \( A \rightsquigarrow B' \) implies \( A \rightsquigarrow B \) because the consequent \( B \) is weaker than \( A' \).

Lemma 48 (steWeakenCons).

\[
[\text{cktSat} \; nl \; (A \rightsquigarrow B'); \text{defSqOfTrForm} \; B \sqsubseteq \text{defSqOfTrForm} \; B'] \implies \text{cktSat} \; nl \; (A \rightsquigarrow B)
\]

Lemma steAndComm and steAndAssoc say that operator and\( _\tau \) satisfies commutative and associative laws.

Lemma 49 (steAndComm). defSqOfTrForm \( (A \text{ and}_\tau B) = \text{defSqOfTrForm} \; (B \text{ and}_\tau A) \)

Lemma 50 (steAndAssoc). defSqOfTrForm \( ((A \text{ and}_\tau B) \text{ and}_\tau C) = \text{defSqOfTrForm} \; (A \text{ and}_\tau (B \text{ and}_\tau C)) \)

A conjunct \( (\text{False} \rightsquigarrow \tau B) \) can be safely eliminated from a trajectory formula.

Lemma 51 (elimFalseGuard). defSqOfTrForm \( (A \text{ and}_\tau (\text{False} \rightarrow \tau B)) = \text{defSqOfTrForm} \; A \)

A trajectory formula \( \text{True} \rightarrow \tau A \) is equivalent to \( A \).

Lemma 52 (simpTrueGuard). defSqOfTrForm \( (\text{True} \rightarrow \tau A) = \text{defSqOfTrForm} \; A \)

chaos is the unit of the operator and\( _\tau \).

Lemma 53 (andChaosId). defSqOfTrForm \( (A \text{ and}_\tau \text{chaos}) = \text{defSqOfTrForm} \; A \)

defSqOfTrForm is congruent for operator and\( _\tau \).

Lemma 54 (steAndCong). defSqOfTrForm \( (A \text{ and}_\tau B') = \text{defSqOfTrForm} \; (A \text{ and}_\tau B') \) if defSqOfTrForm \( B = \text{defSqOfTrForm} \; B' \).