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

Yongjian Li[*,a], William N. N. Hung[b], Xiaoyu Song[c]

[a]*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences , Beijing, China*
[b]*Synopsys Inc., Mountain View, California 94043, USA*
[c]*Dept. 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)

[1]The 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 [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.

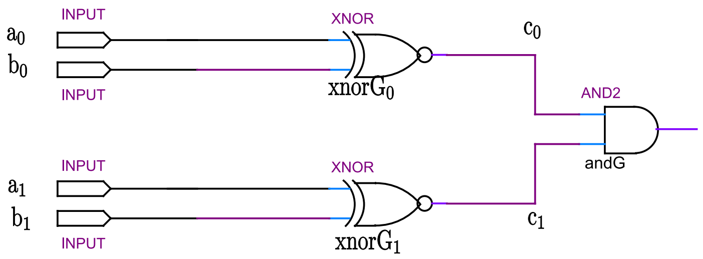

Figure 1: A netlist example

For instance, consider the 2-bit comparator circuit drawn by Quartus II [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 [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 μfp [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 langauge, 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 $[\![A_1; A_2; ...; A_n]\!] \Longrightarrow B$ to mean that with assumptions $A_1, \ldots, A_n$, we can derive a conclusion B. For a pair (a, b), $\mathsf{fst}(a, b) \equiv a$ and $\mathsf{snd}(a, b) \equiv b$. We write $x\#xs$ for the list that extends xs with x, $[x_1, ..x_n]$ for a list $x_1\#..x_n\#[]$, $xs@ys$ for the result list by concatenating

xs with ys, $xs!i$ for the i^{th} element of the list xs (counting from 0 as the first element), $\mathsf{set}\ xs$ for the set of all the elements in xs, $x\ \mathsf{mem}\ ls$ for $x \in (\mathsf{set}\ ls)$ and $\mathsf{length}\ xs$ for the length of the list xs. We also need a *definite description* $\mathsf{THE}\ x.P(x)$ 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 $\top$ 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 $\top$ a clash value. Formally, we define $\mathbb{V} =_{df} \{\mathsf{ff}, \mathsf{tt}, \mathsf{X}, \top\}$

It is common to introduce a truth information ordering $\sqsubseteq$ on $\mathbb{V}$ as follows: $\mathsf{X} \sqsubseteq \mathsf{ff}$, $\mathsf{X} \sqsubseteq \mathsf{tt}$, while ff and tt are incomparable, $\mathsf{ff} \sqsubseteq \top$, and $\mathsf{tt} \sqsubseteq \top$. Namely, the unknown value X contains no truth information; the mutually incomparable values ff and tt contain sufficient information to determine truth exactly, and the top value $\top$ contains inconsistent truth information. We can easily see that $\mathbb{V}$ 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 $\mathbb{V}$, there are natural definitions for negation $\mathsf{NOT}(\neg_4)$, conjunction $\mathsf{AND}(\wedge_4)$, disjunction $\mathsf{OR}(\vee_4)$[2], etc. The classic definitions of these operators are shown in Figure 3.

[2]Here we use the subcript 4 to distinguish the x-symbols of these operators from their counterparts in boolean domain.

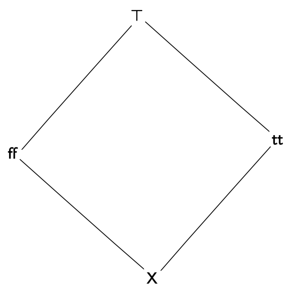

Figure 2: STE lattice

a	$\neg_4 a$
ff	tt
tt	ff
X	X
⊤	⊤

a	b	$a \wedge_4 b$
ff	v_3	ff
tt	v_4	v_4
X	ff	ff
X	tt	X
X	X	X
⊤	v_4	⊤
v_4	⊤	⊤

a	b	$a \vee_4 b$
ff	v_4	v_4
tt	v_3	tt
X	ff	X
X	tt	tt
X	X	X
⊤	v_4	⊤
v_4	⊤	⊤

Figure 3: Operators over four-valued lattice ($v_3 \in \{\mathsf{tt}, \mathsf{ff}, \mathsf{X}\}, v_4 \in \{\mathsf{tt}, \mathsf{ff}, \mathsf{X}, \top\}$

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

$$\begin{array}{l}\texttt{types boolPairs} = \texttt{bool} \times \texttt{bool} \\ \top \equiv (\texttt{False}, \texttt{False})\ \texttt{tt} \equiv (\texttt{True}, \texttt{False}) \\ \texttt{ff} \equiv (\texttt{False}, \texttt{True})\ \texttt{X} \equiv (\texttt{True}, \texttt{True})\end{array}$$

The least-upper bound operator $\sqcup$ and the partial ordering relation $\sqsubseteq$ are defined as follows:

$$\begin{array}{l}\texttt{a} \sqcup \texttt{b} \equiv (\texttt{fst a} \wedge \texttt{fst b}, \texttt{snd a} \wedge \texttt{snd b}) \\ \texttt{a} \sqsubseteq \texttt{b} \equiv \texttt{a} \sqcup \texttt{b} = \texttt{b}\end{array}$$

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` $\texttt{in}_1$ $\texttt{in}_2$ ...`out`

$\texttt{in}_1\texttt{\_value}_1$ $\texttt{in}_2\texttt{\_value}_1$ ...$\texttt{out\_value}_1$

$\texttt{in}_1\texttt{\_value}_2$ $\texttt{in}_2\texttt{\_value}_2$ ...$\texttt{out\_value}_2$

where $\texttt{in}_1, \texttt{in}_2, ...,$ are names of the inputs of the gate, `out` is the name of its output. The subsequent lines define the on-off sets:: $\texttt{in}_\texttt{i}\texttt{\_value}_\texttt{j}$ is one of 0, 1, or − (don't care), and $\texttt{out\_value}_\texttt{i}$ 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 $\lambda\ a\ b.a \wedge_4 b \vee_4 \neg_4 a \wedge_4 \neg_4 b$. Informally we write F_{tab} for the induced function from the table *tab*.

For instance, a two-input AND gate with inputs `a` and `b` and output `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
                 11 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 `latch_output` in the next time is the value of `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:* `.latch latch-input latch-output type control-signal [latch-control-list]`*, where* `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,* `.latch in1 out1 re clk as=set ar=reset en=en1` *specifies a flip-flop which is driven at the rising edge of signal* `clk` *with an input signal* `in1`*, an output signal* `out1`*, an asynchronous reset signal* `reset` *and a asynchronous set signal* `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.*

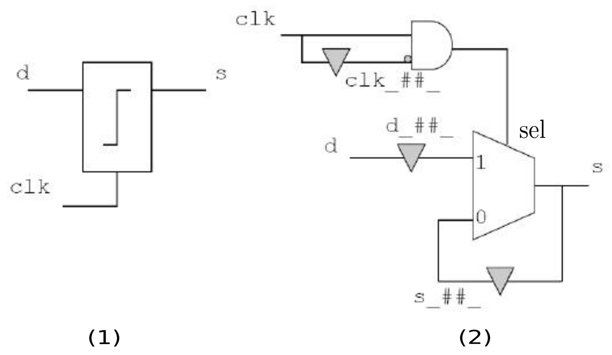

Figure 4: A rising-edge triggered flipflop

3.2. Formalization of netlists

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

`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 .

```
datatype LIT= One | Zero | DontCare
types LINE=LIT list
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:

```
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 c_1 $[a_1, b_1]$ $[[\mathsf{ONE}, \mathsf{ONE}]]$ 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 $g \equiv inp$, if $g =$ Input inp, or fanOut $g \equiv out$ 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, $\mathsf{fanIn}\ g \equiv []$, if $g = \mathsf{Input}\ inp$, or $\mathsf{fanIn}\ g \equiv inps$ if $g = \mathsf{Gate}\ out\ inps\ tab$, or $\mathsf{fanIn}\ g \equiv [inp]$ if $g = \mathsf{Delay}\ out\ inp$.

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

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:

```
consts netlists :: (entity set) set
inductive netlists
intros
nilNetlist :∅ ∈ netlists;
addInput :
  ⟦nl ∈ netlists; ¬isDefinedIn n nl⟧
  ⟹ {Input n} ∪ nl ∈ netlists;
addDelay :
  ⟦nl ∈ netlists; ¬isDefinedIn n nl⟧
  ⟹ {Delay n inp} ∪ nl ∈ netlists;
addGate :
  ⟦nl ∈ netlists; ¬isDefinedIn n nl;
  ∀inps_i. (inps_i mem inps) ⟶ isDefinedIn inps_i nl;
  ∀l.(l mem tab) ⟶ length l = length inps⟧
  ⟹ {Gate n inps tab } ∪ nl ∈ netlists.
```

In the above definition, rule `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 $\neg\mathsf{isDefinedIn}\ n\ nl$ 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 `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 $\neg\mathsf{isDefinedIn}\ n\ nl$ can eliminate combinational cycle in a netlist. Unlike rule `addGate`, `addDelay` rule allows that the input node of a delay can be used before the node is defined. Formally, when a component $\mathsf{Delay}\ n\ inp$ is added in the rule, 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* $xnorTab = [[\mathsf{ZERO}, \mathsf{ZERO}], [\mathsf{ONE}, \mathsf{ONE}]], xnorG_0 = \mathsf{Gate}\ c_0\ [a_0, b_0]\ xnorTab$, $xnorG_1 = \mathsf{Gate}\ c_1\ [a_1, b_1]\ xnorTab$, $\mathsf{andTab} = [[\mathsf{ONE}, \mathsf{ONE}]]$, $andG = \mathsf{Gate}\ out\ [c_0, c_1]\ andTab$, *then the set*

$$nl = \{\mathsf{Input}\ a_0, \mathsf{Input}\ b_0, \mathsf{Input}\ a_1, \mathsf{Input}\ b_1, xnorG_0, xnorG_1, andG\}$$

stands for the netlist shown in Figure 1. In figure 4, let $tab_1 = [\mathsf{ONE}, \mathsf{ZERO}]$*,* $G_1 = \mathsf{Gate}\ sel\ [clk, clk\_\#\#\_]\ tab_1$*, and* $tab_2 = [[\mathsf{ONE}, \mathsf{ONE}, \mathsf{DontCare}], [\mathsf{ZERO}, \mathsf{DontCare}, \mathsf{ONE}]]$*,* $G_2 = \mathsf{Gate}\ s\ [sel, d\_\#\#\_, s\_\#\#\_]\ tab_2$*,* $delay_1 = \mathsf{Delay}\ d\_\#\#\_\ d$*,* $delay_2 = \mathsf{Delay}\ s\_\#\#\_\ s$*,* $nl_2 = \{G_1, G_2, delay_1, delay_2\}$*,* 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. $[\![nl \in netlists;\ \mathsf{isDefinedIn}\ n\ nl]\!] \Longrightarrow \exists! l. l \in nl\ \wedge \mathsf{fanOut}\ l\ = n.$

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

Definition of netlists itself can not guarantee that each node of a netlist is defined because an input of a delay can be used without being defined. In real circuit designs, an input of a 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. $\mathsf{isClosed}\ nl \equiv \forall\ m\ n.\mathsf{isDefinedIn}\ m\ nl \longrightarrow n \in \mathsf{set}\ (\mathsf{fanins}\ ((\mathsf{lookUp}\ nl\ m))) \longrightarrow \mathsf{isDefinedIn}\ n\ nl$

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 $nl \in$ netlists and 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 `state`= `node` $\Rightarrow$ `boolPairs` is defined. A state sequence assigns a state to a time point. Here we still use nat to define the type time. Thus, we define `stateSeq` = `time` $\Rightarrow$ `state`. Naturally, we extend the ordering relation on the state and stateSeq types. we define $s_1 \sqsubseteq_s s_2 \equiv \forall n.s_1\ x \sqsubseteq s_2\ x$, and $sq_1 \sqsubseteq_{sq} sq_2 \equiv \forall t.sq_1\ t \sqsubseteq_s sq_2\ t$.

Trajectory Evaluation Logic. Specifications in STE are symbolic trajectory formulas. In order to formalize the syntax of trajectory formulas, we introduce a datatype trajForm as follows:

```
datatype trajForm = Is1 node |Is0 node|chaos
   |Next trajForm
   |When bool trajForm (infixr ⟶T 65)
   |TAND trajForm trajForm (infixr andT 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 valid on datatype trajForm.

```
consts
valid :: stateSeq ⇒ trajForm ⇒ bool
          ((_ ⊨ _) [80, 80]80)
primrec
sq ⊨ (Is1 n) = tt ⊑ (sq 0 n)
sq ⊨ (Is0 n) = ff ⊑ (sq 0 n)
sq ⊨ chaos = True
sq ⊨ (A₁ and_T A₂) = (sq ⊨ A₁∧sq ⊨ A₂)
sq ⊨ (P ⟶_T A) = (P ⟶ sq ⊨ A)
sq ⊨ (Next A) = ((suffix 1 sq) ⊨ A)
```

where notation $((\_ \models \_)\ [80,80]80)$ stands for an infix notation $\models$ for function valid, and 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 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 tab, provided that the returned values of inputs of the gate are $v_1, ..., v_i$ after simulation, the value of n is returned as the upperbound of $s\ n$ and $F_{tab}\ v_1\ ...\ v_i$.

For instance, suppose that $s\ a_0 = \mathsf{tt}$, $s\ b_0 = \mathsf{tt}$, $s\ a_1 = \mathsf{tt}$, and $s\ n = \mathsf{X}$ 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 $n \in \{a_0, b_0, a_1, b_1\}$, $s'\ c_0 = \mathsf{tt}$, $s'\ c_1 = \mathsf{X}$, and $s'\ out = \mathsf{X}$. Formally, the information propagation can be represented as a set of value assignments as follows: $\{(a_0, \mathsf{tt}), (b_0, \mathsf{tt}), (a_1, \mathsf{tt}), (b_1, \mathsf{X})(c_0, \mathsf{tt}), (c_1, \mathsf{X}), (out, \mathsf{X})\}$.

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: funOfLit (v, lit) returns the input value v if lit is on, else if lit is off, then returns the negation of v, else just returns tt. Here we briefly explain why tt is returned when the literal is DontCare. Because tt is an unit for the operator AND in the four-valued domain, and 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 tt, then the value of this line will not be care of the value of this literal. funOfLine vs $line$ returns the conjunction of the values of literals in a line provided that the values assigned to inputs are vs. funOfTab tab vs returns the disjunction of the values of lines of a table provided that the values assigned to inputs are vs.

```
funOfLit :: boolPairs × Lit ⇒ boolPairs
funOfLit x ≡ if (snd x) = ONE then (fst x)
                else if ( snd x) = ZERO
                then (NOT (fst x))
                 else tt
funOfLine :: boolPairs list ⇒ LINE ⇒ boolPairs
primrec funOfLine bps [] = tt
  funOfLine (bps) (el0#ls) =
  AND (funOfLit ((hd bps), el0)) (funOfLine (tl bps) ls)
funOfTab :: PLA ⇒ boolPairs list ⇒ boolPairs
primrec funOfTab [] bps = ff
  funOfTab (line#tbl) bps =
  OR (funOfLine bps line) (funOfTab tbl bps)
```

Now we formally introduce a so-called closure relation rclosure, which is defined on a netlist and a state. 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:

```
consts rclosure :: entity set ⇒ state ⇒ (node × boolPairs) set
inductive rclosure nl s
intros
stAddInput :
⟦Input n ∈ nl⟧ ⟹ (n, s n) ∈ rclosure nl s
stAddLatch :
⟦Delay n inp ∈ nl⟧ ⟹ (n, s n) ∈ rclosure nls
stAddGate :
⟦Gate n inps tab ∈ nl; length stateLs = length inps;
∀l.(l mem tab)) ⟶ length l = length inps;
∀pair.pair mem (zip inps stateLs) ⟶  pair ∈ rclosure nl s⟧
 ⟹ (n, ((funOfTab tab stateLs) ⊔ (s n))) ∈ rclosure nl s
```

The relation `rclosure nl s` is corresponding to a function, namely, for any node n such that isDefinedIn n nl, there is pair p such that fst $p = n$, furthermore, if both (n, v_1) and (n, v_2) are in rclosure nl s, then $v_1 = v_2$. Intuitively, 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. $[\![\textsf{isDefinedIn}\ n\ nl]\!] \Longrightarrow \exists! pair. pair \in \textsf{rclosure}\ nl\ s\ \wedge \textsf{fst}\ pair = n$

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 n
```

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*, $f\ x \sqsubseteq f\ y$ if $x \sqsubseteq y$. (2) f is *idempotent*: $f\ x = f\ \ (f\ x)$; (3) f is *extensive*: $x \sqsubseteq f\ x$. Function fclosure nl is a closure function.

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

1. $[\![\ s_1 \sqsubseteq_s s_2]\!] \Longrightarrow \textsf{fclosure}\ nl\ s_1\ n \sqsubseteq \textsf{fclosure}\ nl\ s_2\ n$
2. $s\ n \sqsubseteq \textsf{fclosure}\ nl\ s\ n$
3. $\textsf{fclosure}\ nl\ (\textsf{fclosure}\ nl\ s)\ n = \textsf{fclosure}\ nl\ s\ n$

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 $data_t$ 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 $data_t$ and $\sigma(t+1)(data\_\#\#\_)$, 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 inps = fanins l in
                ((fSeq nl σ t) (hd inps)) ⊔ (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. $[\![nl \in \mathsf{netlists}; \mathsf{isClosed}\ nl; sq_1 \sqsubseteq_{sq} sq_2]\!]$
   $\Longrightarrow \mathsf{fSeq}\ nl\ sq_1 \sqsubseteq_{sq} \mathsf{fSeq}\ nl\ sq_2$
2. $[\![nl \in \mathsf{netlists}; \mathsf{isClosed}\ nl]\!] \Longrightarrow sq \sqsubseteq_{sq} \mathsf{fSeq}\ nl\ sq$
3. $[\![nl \in \mathsf{netlists}]\!] \Longrightarrow \mathsf{fSeq}\ nl\ (\mathsf{fSeq}\ nl\ sq) = \mathsf{fSeq}\ nl\ sq$

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 trajOfCirc *nl* as the set of all trajectories of a netlist *nl*:

$$\texttt{trajOfCirc} :: \texttt{entity set} \Rightarrow \texttt{stateSeq set}$$
$$\texttt{trajOfCirc nl} \equiv \{\texttt{sq}.\texttt{fSeq nl sq} = \texttt{sq}\}$$

6. Semantics of STE

Now we define the semantics of a STE assertion $A \leadsto C$, 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 cktSat $nl\ A \leadsto C$, if for every trajectory τ 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.

$$\texttt{datatype assertion} =$$
$$\texttt{Leadsto trajForm trajForm}\ (\texttt{infixr} \leadsto 50)$$

We introduce a predicate cktSat that checks the validity of a STE assertion.

$$\texttt{cktSat} :: \texttt{entity set} \Rightarrow \texttt{assertion} \Rightarrow \texttt{bool}$$
$$\texttt{primrec cktSat nl}\ (\texttt{A} \leadsto \texttt{C}) =$$
$$(\forall \tau. \tau \in (\texttt{trajOfCirc nl}) \longrightarrow (\tau \models \texttt{A} \longrightarrow \tau \models \texttt{C}))$$

The key feature of STE logic is that there is a unique weakest sequence that satisfies f for any boolean symbolic variable assignment ϕ. This sequence is called the defining sequence of f. To define the defining sequence of a formula, we introduce a primary recursive function 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* defSqOfTrForm *A, is defined as a primary recursive function on type* trajForm.

```
defSqOfTrForm ::trajForm⇒stateSeq
primrec
defSqOfTrForm (Is1 n) =(λt m.(if (t=0∧m=n) then tt else X))
defSqOfTrForm (Is0 n) =(λt m.(if (t=0∧m=n) then ff else X))
defSqOfTrForm (A and_T B) =
          (λt m.(defSqOfTrForm A t m)⊔(defSqOfTrForm B t m))
defSqOfTrForm (P ⟶_T A) = (λt m. let v = (defSqOfTrForm A t m) in
                                  (P ⟶ (fst v), P ⟶ (snd v))
defSqOfTrForm (Next A) = (λt m. let v=(defSqOfTrForm A (t - 1) m) in
                            if (t≠0) then v else X)
defSqOfTrForm chaos= λt m. X
```

In the above definition of defSqOfTrForm, $\longrightarrow$ 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. defSqOfTrForm $A \models A$

Furthermore, for any sequence σ that satisfies A, the defining sequence is the weakest of all.

Lemma 9.

(1) defSqOfTrForm $A \sqsubseteq_{sq} sq \Longrightarrow sq \models A$.
(2) $sq \models A \Longrightarrow$ defSqOfTrForm $A \sqsubseteq_{sq} sq$

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* defTrajOfCirc A nl, *is defined as follows:*
defTrajOfCirc A nl $\equiv$ fSeq nl (defSqOfTrForm A)

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

Lemma 11. (defTrajOfCirc A nl) $\in$ trajOfCirc $nl \wedge$ (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) $[\![\tau \in$ trajOfCirc $nl;\ \tau \models A]\!] \Longrightarrow$ (defTrajOfCirc A nl)$\sqsubseteq_{sq} \tau$

(2) $[\![(\mathsf{defTrajOfCirc}\ A\ nl\ ) \sqsubseteq_{sq} \tau]\!] \Longrightarrow \tau \models A$

The following lemma is the most fundamental result of STE theory, which states that $(\mathsf{defSqOfTrForm}\ C) \sqsubseteq_{sq} (\mathsf{defTrajOfCirc}\ A\ nl)$ if and only if $\mathsf{cktSat}\ nl$ $(A \rightsquigarrow C)$ 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 $\mathsf{cktSat}\ nl\ (A \rightsquigarrow C)$, we only need consider whether $(\mathsf{defSqOfTrForm}\ C) \sqsubseteq_{sq} (\mathsf{defTrajOfCirc}\ A\ nl)$ holds.

Lemma 13.

(1) $[\![(\mathsf{defSqOfTrForm}\ C) \sqsubseteq_{sq} (\mathsf{defTrajOfCirc}\ A\ nl) \Longrightarrow \mathsf{cktSat}\ nl\ (A \rightsquigarrow C)$

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

(2) $[\![\mathsf{cktSat}\ nl\ (A \rightsquigarrow C)]\!] \Longrightarrow (\mathsf{defSqOfTrForm}\ C) \sqsubseteq_{sq} (\mathsf{defTrajOfCirc}\ A\ nl)$

Proof. By lemma 11, we have $(\mathsf{defTrajOfCirc}\ A\ nl) \in \mathsf{trajOfCirc}\ nl$ and $(\mathsf{defTrajOfCirc}\ A\ nl) \models A$. From this, by the definition of $\mathsf{cktSat}\ nl\ (A \rightsquigarrow C)$, we have (a) $(\mathsf{defTrajOfCirc}\ A\ nl) \models C$. By lemma 9 (2), we easily show $\mathsf{efSqOfTrForm}\ C \sqsubseteq_{sq} (\mathsf{defTrajOfCirc}\ A\ nl)$. ∎

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* $\mathsf{subNet}\ nl\ nl'$*, which is an inductively defined set by the following rules:*

```
consts subNet :: entity set ⇒ entity set ⇒ entity set
inductive subNet nl nl′
intros
subAddself :
⟦enttr ∈ nl′; enttr ∈ nl ⟧ ⟹ enttr ∈ subNet nl nl′
subAddLink :
⟦enttr₀∈ subNet nl nl′; enttr₁∈ nl;
 (fanout (enttr₁)) ∈ set (fanins (enttr₀))⟧ ⟹ enttr₁∈ subNet nl nl′
```

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

Example 15. *In Example 2, let* $nl' = \{xnorG_0\}$*,* $\mathsf{subNet}\ nl\ nl' = \{\mathsf{Input}\ a_0,$ $\mathsf{Input}\ b_0,\ xnorG_0\}$.

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)

$$[\![nl' \subseteq nl; \mathsf{isDefinedIn}\ \ n \in nl']\!] \Longrightarrow (n,v) \in (\mathsf{rclosure}\ nl'\ \ s) = (n,v) \in (\mathsf{rclosure}\ nl\ \ s)$$

(2)

$$[\![nl' \subseteq nl; \mathsf{isDefinedIn}\ \ n \in nl']\!] \Longrightarrow \mathsf{fclosure}\ nl\ \ s\ \ n = \mathsf{fclosure}\ nl'\ \ s\ \ n$$

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.

$$[\![nl' \subseteq nl; \mathsf{isDefinedIn}\ \ n \in nl']\!] \Longrightarrow \mathsf{fSeq}\ nl\ \ s\ \ n = \mathsf{fSeq}\ nl'\ \ s\ \ n$$

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

Lemma 18.

$$[\![nl' \subseteq nl; \mathsf{isDefinedIn}\ \ n \in nl']\!] \Longrightarrow \mathsf{defTrajOfCirc}\ B\ nl\ \ t\ \ n = \mathsf{defTrajOfCirc}\ B\ nl'\ \ t\ \ n$$

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 $\mathsf{cktSat}\ nl\ A \leadsto C$ iff $\mathsf{cktSat}\ nl'\ A \leadsto C$.

Lemma 19 (subsetI).

$$\begin{aligned}&[\![nl' \subseteq nl;\ \forall n. n \in (\mathsf{onNodes}\ C) \longrightarrow \mathsf{isDefinedIn}\ n\ nl']\!]\\ &\Longrightarrow \mathsf{cktSat}\ nl'\ \ A \leadsto C = \mathsf{cktSat}\ nl\ \ A \leadsto C\end{aligned}$$

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 $n \in (\mathsf{onNodes}\ C)$*, we have that* $\mathsf{defTrajOfCirc}\ A\ nl\ \ t\ \ n = \mathsf{defTrajOfCirc}\ A\ nl'\ \ t\ \ n$*. Therefore,* $(\mathsf{defSqOfTrForm}\ C)\ t\ n \sqsubseteq (\mathsf{defTrajOfCirc}\ A\ nl)\ t\ n$ *iff* $(\mathsf{defSqOfTrForm}\ C)\ \ t\ \ n \sqsubseteq_{sq} (\mathsf{defTrajOfCirc}\ A\ nl')\ \ t\ \ n$ *for any* t*, any node* $n \in (\mathsf{onNodes}\ C)$*. We are only interested in the evaluation of nodes* $n \in (\mathsf{onNodes}\ C)$ *because* $(\mathsf{defSqOfTrForm}\ C)\ \ t\ \ n = \mathsf{X}$ *for any node* $n \notin (\mathsf{onNodes}\ C)$ *and* $\mathsf{X} \sqsubseteq v$ *for any value* v*.*

We need two preliminary definitions before we continue.

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

$$\begin{array}{l}
\texttt{onNodes} :: \texttt{trajForm} \Rightarrow \texttt{node set} \\
\texttt{primrec} \\
\quad \texttt{onNodes}\ (\texttt{Is1 n}) = \{\texttt{n}\} \\
\quad \texttt{onNodes}\ (\texttt{Is0 n}) = \{\texttt{n}\} \\
\quad \texttt{onNodes}\ (\texttt{A and}_{\texttt{T}}\ \texttt{B}) = (\texttt{onNodes A}) \cup (\texttt{onNodes B}) \\
\quad \texttt{onNodes}\ (\texttt{P} \longrightarrow_{\texttt{T}} \texttt{A}) = \texttt{onNodes A} \\
\quad \texttt{onNodes}\ (\texttt{Next A}) = \texttt{onNodes A} \\
\quad \texttt{onNodes chaos} = \varnothing
\end{array}$$

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

Definition 21.

$$\begin{array}{l}
\texttt{InducedNet} :: \texttt{entity set} \Rightarrow \texttt{node set} \Rightarrow \texttt{entity set} \\
\texttt{InducedNet nl ns} \equiv \texttt{subNet nl}\ \{\texttt{g}.\exists \texttt{n}.\texttt{isDefinedIn n nl} \wedge \texttt{n} \in \texttt{ns} \wedge \texttt{g} = \texttt{lookUp nl n}\}
\end{array}$$

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, $(\mathsf{onNodes}\ B) \cap \mathsf{defAsOuts}\ (\mathsf{InducedNet}\ nl\ (\mathsf{onNodes}\ C)) = \varnothing$, then B has nothing with the truth of this assertion.

Lemma 22 (steEqAnt).

$$\begin{array}{l}
[\![(\mathsf{onNodes}\ B) \cap \mathsf{defAsOuts}\ (\mathsf{InducedNet}\ nl\ (\mathsf{onNodes}\ C)) = \varnothing; \\
\forall n.n \in (\mathsf{onNodes}\ C) \longrightarrow \mathsf{isDefinedIn}\ n\ nl]\!] \\
\Longrightarrow \mathsf{cktSat}\ nl\ A \leadsto C = \mathsf{cktSat}\ nl\ (A\ \mathsf{and}\ B) \leadsto C
\end{array}$$

For instance, let $A = (\mathsf{Is1}\ a_0)\ \texttt{and}_{\texttt{T}}\ (\mathsf{Is1}\ b_0)$, $B = (\mathsf{Isb}\ a_1\ Ba_1)\ \texttt{and}_{\texttt{T}}\ (\mathsf{Isb}\ b_1\ Bb_1)$, $C = \mathsf{Is1}\ c_0$, nl be the netlist as shown Fig. 1, $nl' = \mathsf{InducedNet}\ nl\ (\mathsf{onNodes}\ C)$, then we have $\mathsf{onNodes}\ B = \{a_1, b_1\}$, $\mathsf{onNodes}\ C = \{c_0\}$, $nl' = \{\mathsf{Input}\ a_0, \mathsf{Input}\ b_0, xnorG_0\}$, because $(\mathsf{onNodes}\ B) \cap \mathsf{defAsOuts}\ nl' = \varnothing$, $\mathsf{cktSat}\ nl\ (A\ \mathsf{and}\ B) \leadsto C$ is equivalent to $\mathsf{cktSat}\ nl\ A \leadsto C$. Usually the $(A\ \mathsf{and}\ B) \leadsto C$ has more symbolic variables than $A \leadsto C$ does, so we often use the following law which tells us the heuristics to simplify an assertion by eliminating unnecessary antecedents.

Lemma 23 (steDelAnt).

$$\begin{array}{l}
[\![(\mathsf{onNodes}\ B) \cap \mathsf{defAsOuts}\ (\mathsf{InducedNet}\ nl\ (\mathsf{onNodes}\ C)) = \varnothing; \\
\mathsf{cktSat}\ nl\ A \leadsto C]\!] \\
\Longrightarrow \mathsf{cktSat}\ nl\ (A\ \mathsf{and}\ B) \leadsto C
\end{array}$$

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* sym *nl nl' f, which is defined as follows:*

```
sym :: (node => node) ⇒ entity set ⇒ entity set ⇒ bool
sym f M N ≡ bij f ∧ f‘(defAsOuts M) = (defAsOuts N)∧
(∀m.isDefinedIn m M ⟶ isDefinedIn (f m) N∧
(let lx = (lookUp M m) in
 let ly = (lookUp N (f m)) in
 (case (lx) of
  Input x ⇒ ly = Input (f x)|
  Delay out data ⇒ ly = Delay (f out) (f data)|
  Gate out inps tab ⇒
  ly = Gate (f out) (map f inps) tab)))
```

Roughly speaking, 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 nodeSetSym f M N nl specifies that the subnetlists induced from node sets M and N in a 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.

```
nodeSetSym :: (node => node) ⇒ node set ⇒ node set ⇒ entity set => bool
nodeSetSym f M N nl ≡ sym f (InducedNet nl M) (InducedNet nl N)
```

Example 26. *Let $nl_0 = \{\mathsf{Input}\ a_0, \mathsf{Input}\ b_0, xnorG_0\}$, $nl_1 = \{\mathsf{Input}\ a_1, \mathsf{Input}\ b_1, xnorG_1\}$, $N_0 = \{c_0\}$, $N_1 = \{c_1\}$, and $f = \lambda x.$*(if $x = a_0$ then a_1 else if $x = a_1$ then a_0 else if $x = b_0$ then b_1 else if $x = b_1$ then b_0 else *if* $x = c_0$ then c_1 else if $x = c_1$ *then* c_0 *else* x). InducedNet nl $N_0 = nl_0$, InducedNet nl $N_1 = nl_1$. *We have that* sym nl_0 nl_1 f *and* nodeSetSym f N_0 N_1 nl.

Next we define permutations on states, sequences, and formulas. These are similar to their conterparts in [17].

Definition 27. *Permutation on states.*

```
appSym2State :: (node ⇒ node) ⇒ state ⇒ state
  appSym2State f s = λ n.s (f n))
```

Definition 28. *Permutation on sequences.*

```
appSym2Seq :: (node ⇒ node) ⇒ stateSeq ⇒ stateSeq
  appSym2Seq f sq ≡ λ t.appSym2State f (sq t)
```

Definition 29. *Permutation on formulas.*

```
applySym2Form :: (node ⇒ node) ⇒ trajForm ⇒
trajForm
primrec
  appSym2Form f (Is0 n) = Is0 (f n)
  appSym2Form f (Is0 n) = Is1 (f n)
  appSym2Form f (A and_T B) =  (appSym2Form f A) and_T (appSym2Form f B)
  appSym2Form f (P ⟶_T A) =  P ⟶_T (appSym2Form f A)
  appSym2Form f  (Next A) = Next (appSym2Form f A)
  appSym2Form f chaos = 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: isSwap $f \equiv \forall a\ b.f\ a = b \longrightarrow f\ b = a$.

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.

$$\text{isSwap } f \Longrightarrow$$
$$\text{appSym2Seq } f \ (\text{defSqOfTrForm } A) = \text{defSqOfTrForm } \ (\text{appSym2Form } f \ A)$$

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 appSym2Form f A w.r.t. nl'.

Lemma 31.

$$[\![nl \in \text{netlists};\ nl' \in \text{netlists};\ \text{sym } f\ nl\ nl'; \text{isSwap } f]\!]$$
$$\Longrightarrow \text{appSym2Seq } f\ (\text{trajOfCirc } A\ nl) = \text{trajOfCirc } (\text{appSym2Form f } A)\ nl'$$

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.

$$[\![\text{sym } f\ nl\ nl';\ \text{isSwap } f;\ ]\!] \Longrightarrow$$
$$\text{cktSat } nl\ \ (A \leadsto C) = \text{cktSat } nl'\ \ (\text{appSym2Form } f\ A \leadsto \text{appSym2Form } f\ C)$$

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 $(A \leadsto C)$ implies cktSat nl (appSym2Form f $A \leadsto$ 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).

$$\begin{array}{l}[\![\mathsf{isSwap}\ f; \forall n.n \in (\mathsf{onNodes}\ C) \rightarrow \mathsf{isDefinedIn}\ n\ nl;\\ \forall n.n \in (\mathsf{onNodes}\ (\mathsf{appSym2Form}\ f\ C)) \rightarrow \mathsf{isDefinedIn}\ n\ nl;\\ \mathsf{nodeSetSym}\ f\ (\mathsf{onNodes}\ C)\ (\mathsf{onNodes}\ (\mathsf{appSym2Form}\ f\ C))\ nl]\!]\\ \Longrightarrow \mathsf{cktSat}\ nl\ (A \leadsto C) = \mathsf{cktSat}\ nl\ (\mathsf{appSym2Form}\ f\ A \leadsto \mathsf{appSym2Form}\ f\ C)\end{array}$$

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, $[\![nl \in \mathsf{netlists}; \mathsf{isClosed}\ nl;\ \mathsf{cktSat}\ nl\ (A \leadsto B)$; cktSat nl $(A \leadsto C)]\!] \Longrightarrow$ cktSat nl $A \leadsto (B\ \mathsf{and_T}\ C)$, 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:

$$\begin{array}{l}\mathtt{andLists}\ [] = \mathtt{chaos}\\ \mathtt{andLists}\ (\mathtt{A\#listA}) = \mathtt{A}\ \mathtt{and_T}\ (\mathtt{andLists\ listA})\end{array}$$

Two predicates isFullAndLine :: LINE ⇒ bool and isAndTab :: PLA ⇒ bool are introduced to define a truth table of an AND-gate:

$$\begin{array}{l}\mathtt{isFullAndLine\ line} \equiv \forall \mathtt{l.l\ mem\ line} \longrightarrow \mathtt{l} = \mathtt{ONE}\\ \mathtt{isAndTab\ tab} \equiv \mathtt{length\ tab} = 1 \wedge \mathtt{isFullAndLine\ (hd\ tab)}\end{array}$$

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).

$$\begin{array}{l}[\![\mathsf{isAndTab}\ tab; \mathsf{Gate}\ out\ inps\ tab \in nl;\\ \forall l.(l\ \mathsf{mem}\ tab) \longrightarrow \mathsf{length}\ l = \mathsf{length}\ inps]\!] \Longrightarrow\\ \mathsf{cktSat}\ nl\ ((\mathsf{andLists}\ (map\ (\lambda n.\mathsf{Is1}\ n)\ inps)) \leadsto (\mathsf{Is1}\ out))\end{array}$$

The second lemma says that if one input node of an and-gate are set low, then its out turns low.

Lemma 35 (andTabPropF).

$$\llbracket \mathsf{isAndTab}\ tab; \mathsf{Gate}\ out\ inps\ tab \in nl; inps_i\ \mathsf{mem}\ inps;$$
$$\forall l.(l\ \mathsf{mem}\ tab) \longrightarrow \mathsf{length}\ l = \mathsf{length}\ inps\rrbracket \Longrightarrow$$
$$\mathsf{cktSat}\ nl\ (\mathsf{Is0}\ inps_i)) \rightsquigarrow (\mathsf{Is0}\ out))$$

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 posAssertOfLine *inps lits* returns a list of trajectory formulas, each of which specifies a special value of each node $inps_i$ according to the literal $lits_i$. If $lits_i$ is ZERO, then $inps_i$ is specified as ff by an Is0 formula, else if $lits_i$ is ONE, then $inps_i$ is specified as tt by an Is1 formula, otherwise it is set as X by chaos. Let $inps = [i_1, i_2]$, $line = [\mathsf{ONE}, \mathsf{ONE}]$, then `posAssertOfLine` *inps line*=[Is1 i_1, Is1 i_2].

```
posAssertOfLine :: node list ⇒ Literal list ⇒  trajForm list
primrec
 posAssertOfLine inps [] = []
 posAssertOfLine inps (l#line) =
 let otherAss = posAssertOfLine (tl inps) line in
 (case l of ZERO ⇒ (Is0 (hd inps))#otherAss|
    ONE ⇒ (Is1 (hd inps))#otherAss|
    DONTCARE ⇒ chaos#otherAss)
```

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

Lemma 36 (orTabPropT).

$$\llbracket \mathsf{Gate}\ out\ inps\ tab \in nl; l\ \mathsf{mem}\ tab;$$
$$\forall l.(l\ \mathsf{mem}\ tab) \longrightarrow \mathsf{length}\ l = \mathsf{length}\ inps\rrbracket \Longrightarrow$$
$$\mathsf{cktSat}\ nl\ (\mathsf{andLists}\ (\mathsf{posAssertOfLine}\ inps\ l)) \rightsquigarrow (\mathsf{Is1}\ out))$$

Next we introduce a function isNegAssOfLine *A line inps*, the function returns true if a formula A specifies a proper value for some node $inps_i$ according to the literal $lits_i$, if A is Is1 n, then the literal is ZERO, else if A is Is1 n, then the literal is ONE. For simplicity, 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 i_1, Is0 i_2], we have $\exists A.\,(A \ \mathsf{mem}\ asList) \wedge$isNegAssOfLine A *inps* l for any l such that l mem *tab*.

Lemma 37 (orTabPropF).

$$\llbracket \mathsf{Gate}\ out\ inps\ tab \in nl; \forall l.(l\ \mathsf{mem}\ tab) \longrightarrow \mathsf{length}\ l = \mathsf{length}\ inps;\\ \forall l.(l\ \mathsf{mem}\ tab) \longrightarrow (\exists A.\,(A\ \mathsf{mem}\ asList) \wedge is\mathsf{NegAssOfLine}\ A\ \ inps\ l\ )\rrbracket \Longrightarrow\\ \mathsf{cktSat}\ nl\ (\mathsf{andLists}\ asList) \leadsto (\mathsf{Is0}\ out))$$

For convenience, we define a syntactical abbreviation: Isb n $a \equiv (a \longrightarrow_{\mathsf{T}} \mathsf{Is1}\ n)$ $\mathtt{and}_{\mathsf{T}}$ $(\neg a \longrightarrow_{\mathsf{T}} \mathsf{Is0}\ n)$. Roughly speaking, Isb 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.

$$\llbracket \mathsf{Delay}\ out\ data \in nl; nl \in \mathsf{netlists}; \mathsf{isClosed}\ nl\rrbracket \Longrightarrow\\ \mathsf{cktSat}\ nl\ (\mathsf{Isb}\ n\ a) \leadsto \mathsf{Next}\ (\mathsf{Isb}\ out\ a))$$

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 techniques 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_{ij} for $f\ i\ j$.

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

Let $bvOfAs$ and $bvOfBs$ be two vectors of boolean variables to model symbolic values of nodes, $bvOfAs_i$ is a boolean variable. $antOfAs = [\mathsf{Isb}\ a_i\ bvOfAs_i.\ i<N]$, $antOfBs = [\mathsf{Isb}\ b_i\ bvOfBs_i.\ i<N]$, $Gp_0 = \exists i. i<N \wedge bvOfAs_i \neq bvOfBs_i$, $Gp_1 = \forall i. i<N \longrightarrow bvOfAs_i = bvOfBs_i$. Let $ant = \mathsf{andLists}\ (antOfAs@antOfBs)$, $cons_0 = Gp_0 \longrightarrow_{\mathtt{T}} \mathsf{Is0}\ out$, $cons_1 = Gp_1 \longrightarrow_{\mathtt{T}} \mathsf{Is1}\ out$, $cons = cons_0\ \mathtt{and}_{\mathtt{T}}\ cons_1$. Here we want to prove an assertion $\mathsf{cktSat}\ nl\ (ant \leadsto cons)$. Intuitively, ant specifies the symbolic values of the nodes to be compared, $cons_0$ says that out is low when a and b do not agree on a bit i, and $cons_1$ says 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) $[\![i<N; \neg bvOfAs_i \wedge bvOfBs_i]\!] \Longrightarrow \mathsf{cktSat}\ nl\ ant \leadsto \mathsf{andLists}\ [\mathsf{Is0}\ a_i, \mathsf{Is1}\ b_i]$

(2) $[\![i<N; bvOfAs_i \wedge \neg bvOfBs_i]\!] \Longrightarrow \mathsf{cktSat}\ nl\ ant \leadsto \mathsf{andLists}\ [\mathsf{Is1}\ a_i, \mathsf{Is0}\ b_i]$

(3) $[\![i<N]\!] \Longrightarrow \mathsf{cktSat}\ nl\ \ (\mathsf{andLists}\ [\mathsf{Is0}\ a_i, \mathsf{Is1}\ b_i]) \leadsto \mathsf{Is0}\ c_i$

(4) $[\![i<N]\!] \Longrightarrow \mathsf{cktSat}\ nl\ \ (\mathsf{andLists}\ [\mathsf{Is1}\ a_i, \mathsf{Is0}\ b_i]) \leadsto \mathsf{Is0}\ c_i$

(5) $[\![i<N; bvOfAs_i \neq bvOfBs_i]\!] \Longrightarrow \mathsf{cktSat}\ nl\ ant \leadsto \mathsf{Is0}\ c_i$

(6) $[\![i<N]\!] \Longrightarrow \mathsf{cktSat}\ nl\ \mathsf{Is0}\ c_i \leadsto \mathsf{Is0}\ out$

(7) $[\![i<N; bvOfAs_i \wedge bvOfBs_i]\!] \Longrightarrow$
$\mathsf{cktSat}\ nl\ ant \leadsto (\mathsf{andLists}\ (\mathsf{posAssertOfLine}\ [a_i,b_i]\ [\mathsf{ONE},\mathsf{ONE}]))$

(8) $[\![i < N]\!] \Longrightarrow \mathsf{cktSat}\ nl$ $(\mathsf{andLists}\ (\mathsf{posAssertOfLine}\ [a_i, b_i]\ [\mathsf{ONE}, \mathsf{ONE}])) \leadsto \mathsf{Is1}\ c_i$

(9) $[\![i < N; \neg bvOfAs_i \wedge \neg bvOfBs_i]\!] \Longrightarrow \mathsf{cktSat}\ nl\ ant \leadsto$ $(\mathsf{andLists}\ (\mathsf{posAssertOfLine}\ [a_i, b_i]\ [\mathsf{ZERO}, \mathsf{ZERO}]))$

(10) $[\![i < N]\!] \Longrightarrow \mathsf{cktSat}\ nl$ $(\mathsf{andLists}\ (\mathsf{posAssertOfLine}\ [a_i, b_i]\ [\mathsf{ZERO}, \mathsf{ZERO}])) \leadsto \mathsf{Is1}\ c_i$

(11) $[\![i < N; bvOfAs_i = bvOfBs_i]\!] \Longrightarrow \mathsf{cktSat}\ nl\ ant \leadsto \mathsf{Is1}\ c_i$

(12) $[\![Gp_1]\!] \Longrightarrow \mathsf{cktSat}\ nl\ ant \leadsto (\mathsf{andLists}\ [\mathsf{Is1}\ c_i.\ i < N])$

(13) $\mathsf{cktSat}\ nl\ (\mathsf{andLists}\ [\mathsf{Is1}\ c_i.\ i < N]) \leadsto \mathsf{Is1}\ out$

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 orTabPropF is the main rule used to prove these results. (6) says that once c_i is set low, then the output out is set low. (6) is proved by law 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 orTabPropT is the main rule used to prove these results. From these, (12) can be easily proved. (13) can be proved by law andTabPropT.

Lemma 40. $\mathsf{cktSat}\ nl\ (ant \leadsto cons)$.

Proof. For the main goal, we use rule steconjI to decompose it two subgoals: (a) $\mathsf{cktSat}\ nl\ ant \leadsto cons_0$ and (b) $\mathsf{cktSat}\ nl\ ant \leadsto cons_1$.

In order to prove (a), by rule steImpl, we assume that (c) Gp_0, and need show $\mathsf{cktSat}\ nl\ ant \leadsto \mathsf{Is0}\ out$. From (c), we obtain i where $i < N$ and (d) $bvOfAs_i \neq bvOfBs_i$. From this and Lemma 39 (5), we have (e) $\mathsf{cktSat}\ nl\ ant \leadsto \mathsf{Is0}\ c_i$. With Lemma 39 (6), by rule steTrans, we show $\mathsf{cktSat}\ nl\ ant \leadsto \mathsf{Is0}\ out$.

In order to prove (b), by rule steImpl, we assume that (f) Gp_1, and need show $\mathsf{cktSat}\ nl\ ant \leadsto \mathsf{Is1}\ out$.From (f) and Lemma 39 (12), we have (g) $\mathsf{cktSat}\ nl\ ant \leadsto (\mathsf{andLists}\ [\mathsf{Is1}\ c_i.\ i < N])$. With Lemma 39 (13), by rule steTrans, we can show $\mathsf{cktSat}\ nl\ ant \leadsto \mathsf{Is1}\ out$. ■

10.2. $M - N - CAM$

Figure 5 shows a part of a $M - N-$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, Tag and $match$ be a vector of nodes. Let $M > 1$, $N > 1$, $xnorTab$ and $andLine$ be defined as in subsection 10.1, $css = [[c_{ij}.\ j < N].\ i < M]$, $xnorGs = \{\mathsf{Gate}\ c_{ij}\ [T_{ij}, Tag_j]\ xnorTab.\ j < N, i < M\}$, $matches = [match_i.\ i < M]$, $andGs = \{\mathsf{Gate}\ match_i\ css_i\ [andLine].\ i < M\}$, $orLine = \lambda i.[(\lambda j.\mathsf{if}\ (j = i)\ \mathsf{then\ ONE\ else\ DontCare})\ j.\ j < M]$, $orTab = [orLine\ i.i < M]$, $orG = \mathsf{Gate}\ hit\ matches\ orTab$. Let nl be a closed netlist such that $xnorGs \cup andGs \cup \{orG\} \subseteq nl$.

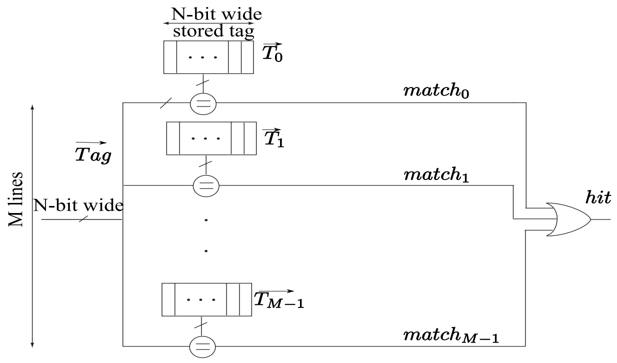

Figure 5: A $M - N-$ CAM

Let $bvOfTs$ be a vector of vectors of of boolean variables to model symbolic values of stored tags, $bvOfTag$ is a vector of boolean variables to model the symbolic value of input tag. $antOfTag = [\mathsf{lsb}\ Tag_j\ bvOfTag_j.\ j < N]$, $antOfTs = [[\mathsf{lsb}\ T_{ij}\ bvOfBs_{ij}.\ j < N].\ i < M]$, $GpOfUnHitI = \lambda i.(\exists j.j < N \wedge bvOfTag_j \neq bvOfT_{ij})$, $GpOfHitI = \lambda i.(\forall j.j < N \longrightarrow bvOfTag_j = bvOfT_{ij})$, $GpOfUnHit = \forall i.i < M \longrightarrow GpOfUnHitI\ i$, $GpOfHit = \exists i.i < M \wedge GpOfHitI\ i$. Let $ant = \mathsf{andLists}\ (antOfTag@(\mathsf{flat}\ antOfTs))$, $cons_0 = GpOfUnHit \longrightarrow_{\mathrm{T}} \mathsf{Is0}\ hit$, $cons_1 = GpOfHit \longrightarrow_{\mathrm{T}} \mathsf{Is1}\ hit$, $cons = cons_0\ \mathtt{and}_{\mathrm{T}}\ cons_1$. Here we want to prove an assertion $\mathsf{cktSat}\ nl\ (ant \leadsto cons)$. In this assertion, ant still specifies that the symbolic values of the nodes of the input tag and the stored tags, $cons_0$ says that the node hit is set low if no line matches the input tag, and $cons_1$ says that the node hit is set high if there exists one line which matches the input tag.

Lemma 41.

(1) $[\![i < M; GpOfUnHitI\ i]\!] \Longrightarrow \mathsf{cktSat}\ nl\ ant \leadsto \mathsf{Is0}\ match_i$

(2) $[\![GpOfUnHit]\!] \Longrightarrow \mathsf{cktSat}\ nl\ ant \leadsto (\mathsf{andLists}\ [\mathsf{Is0}\ match_i.\ i < M])$

(3) $\mathsf{cktSat}\ nl\ (\mathsf{andLists}\ [\mathsf{Is0}\ match_i.\ i < M]) \leadsto \mathsf{Is0}\ hit$

(4) $[\![GpOfUnHit]\!] \Longrightarrow \mathsf{cktSat}\ nl\ ant \leadsto \mathsf{Is0}\ hit$

(5) $[\![i < M; GpOfHitI\ i]\!] \Longrightarrow \mathsf{cktSat}\ nl\ ant \leadsto \mathsf{Is1}\ match_i$

(6) $[\![i < M\ ]\!] \Longrightarrow \mathsf{cktSat}\ nl\ (\mathsf{Is1}\ match_i) \leadsto$ $(\mathsf{andLists}\ (\mathsf{posAssertOfLine}\ matches\ (orLine\ i)))$

(7) $[\![i < M]\!] \Longrightarrow \mathsf{cktSat}\ nl$ $(\mathsf{andLists}\ (\mathsf{posAssertOfLine}\ matches\ (orLine\ i)) \leadsto \mathsf{Is1}\ hit)$

(8) $[\![i < M; GpOfHitI\ i]\!] \Longrightarrow \mathsf{cktSat}\ nl\ ant \leadsto \mathsf{Is1}\ hit$

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 $GpOfUnHitI\ i$ 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 $GpOfHitI\ i$ 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 combing (5), (6) and (7). From these results, it is rather easy to derive the following result by using rules steImpl, steConjI, and steTrans.

Lemma 42. cktSat $nl\ ant \leadsto cons$

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

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

In order to prove (b), by rule steImpI, we assume that (f) $GpOfHit$, and need show cktSat $nl\ ant \leadsto$ Is1 hit. From (f), we can obtain a i such that $i < M$ and $GpOfHitI\ i$. From this, by Lemma 41 (8), we can easily prove that cktSat $nl\ ant \leadsto$ 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 rclosure accurately capture the semantics of the information propagation process. Furthermore, we can formally derive function fclosure and fSeq. Here the function 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 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 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 rclosure $nl\ s$ is single-valued, thus the function 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 steTrans and 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

[1] C.-J. H. Seger, R. E. Bryant, Formal verification by symbolic evaluation of partially-ordered trajectories, Formal Methods in System Design 6 (2) (1995) 147–189. doi:http://dx.doi.org/10.1007/BF01383966.

[2] J. O'Leary, X. Zhao, R. Gerth, C.-J. H. Seger, Formally verifying IEEE compliance of floating-point hardware, Intel Technology Journal Q1 (1999) 147–190.

[3] M. D. Aagaard, R. B. Jones, C.-J. H. Seger, Combining theorem proving and trajectory evaluation in an industrial environment, in: DAC '98: Proceedings of the 35th annual conference on Design automation, ACM, New York, NY, USA, 1998, pp. 538–541. doi:http://doi.acm.org/10.1145/277044.277189.

[4] Technical Publications and Training, Intel Corporation, Forte/fl user guide, 2003rd Edition.

[5] C.-T. Chou, The mathematical foundation for symbolic trajectory evaluation, in: CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification, Springer-Verlag, London, UK, 1999, pp. 196–207.

[6] Altera corporation, Quartus II quick start guide, `http://www.altera.com/literature/manual/mnl_qts_quick_start.pdf`.

[7] J.-W. Roorda, K. Claessen, Explaining symbolic trajectory evaluation by giving it a faithful semantics, in: D. Grigoriev, J. Harrison, E. A. Hirsch (Eds.), Computer Science - Theory and Applications, First International Computer Science Symposium in Russia, CSR 2006, St. Petersburg, Russia, June 8-12, 2006, Proceedings, Vol. 3967 of Lecture Notes in Computer Science, Springer, 2006, pp. 555–566.

[8] J.-W. Roorda, Symbolic trajectory evaluation using a satisability solver, Ph.D. thesis, Department of Computer Science and Engineering Chalmers University of Technology and Goteborg University (2005).

[9] A. Darbari, Symmetry reduction for STE model checking, in: FMCAD, IEEE Computer Society, 2006, pp. 97–105.

[10] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL - a proof assistant for higher-order logic, LNCS 2283, Springer, 2002.

[11] Y. Li, Formalization of symbolic trajectory semantics, `http://lcs.ios.ac.cn/~lyj238/steSymmetry.html` (2009).

[12] M. J. C. Gordon, Why higher-order logic is a good formalism for specifying and verifying hardware, in: G. Milne, P. Subrahmanyam (Eds.), Formal Aspects of VLSI Design, Elsevier Science Publishers, 1986.

[13] T. F. Melham, Formalizing abstraction mechanisms for hardware verification in higher order logic, Ph.D. thesis, University of Cambridge (August 1989).

[14] T. Melham, Higher order logic and hardware verification, Vol. 31 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1993.
URL `http://www.comlab.ox.ac.uk/tom.melham/pub/Melham-1993-HOL.html`

[15] M. Aagaard, T. F. Melham, J. W. O'Leary, Xs are for trajectory evaluation, booleans are for theorem proving, in: CHARME '99: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Springer-Verlag, London, UK, 1999, pp. 202–218.

[16] A. Darbari, Formalization and execution of STE in HOL (extended version), Tech. Rep. RR-03-17, Oxford University Computing Laboratory (March 2003).

[17] A. Darbari, Symmetry reduction for STE model checking using structured models, Ph.D. thesis, University of Oxford (2006).

[18] M. Sheeran, μFP, a language for VLSI design, in: LISP and Functional Programming, 1984, pp. 104–112.

[19] J. T. O'Donnell, Hydra: Hardware description in a functional language using recursion equations and high order combining forms, in: G. J. Milner (Ed.), The Fusion of Hardware Design and Verification, North-Holland, 1988, pp. 309–328.

[20] J. T. O'Donnell, Generating netlists from executable circuit specifications, in: J. Launchbury, P. M. Sansom (Eds.), Functional Programming, Workshops in Computing, Springer, 1992, pp. 178–194.

[21] P. Bjesse, K. Claessen, M. Sheeran, S. Singh, Lava: hardware design in Haskell, in: ICFP, 1998, pp. 174–184.

[22] J. Grundy, T. Melham, J. O'Leary, A reflective functional language for hardware design and theorem proving, Journal of Functional Programming 16 (2) (2006) 157–196. doi:10.1017/S0956796805005757.
URL `http://www.comlab.ox.ac.uk/tom.melham/pub/Grundy-2006-RFL.pdf`

[23] T. Nipkow, Winskel is (almost) right: Towards a mechanized semantics textbook, in: Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag, London, UK, 1996, pp. 180–192.

[24] T. Nipkow, L. C. Paulson, Proof pearl: defining functions over finite sets, in: J. Hurd (Ed.), Theorem Proving in Higher Order Logics (TPHOLs 2005), Vol. 3603 of LNCS, Springer, 2005, pp. 385–396.

[25] G. Winskel, Formal semantics of programming languages, MIT Press, Cambridge, Massachusetts, 1993.

[26] L. C. Paulson, ML for the working programmer, University of Cambridge Press, 1996.

[27] University of California, Berkeley, Berkeley logic interchange format (BLIF), `http://www.cs.uic.edu/~jlillis/courses/cs594/spring05/blif.pdf` (February 22 2005).

[28] M. Pandey, R. Raimi, R. E. Bryant, M. S. Abadir, Formal verification of content addressable memories using symbolic trajectory evaluation, in: DAC '97: Proceedings of the 34th annual Design Automation Conference, ACM, New York, NY, USA, 1997, pp. 167–172. doi:http://doi.acm.org/10.1145/266021.266056.

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 Flase; nat, the type of natural numbers. Standard boolean operators $\wedge$ and $\vee$ and $\rightarrow$ are defined as usual. Function types are denoted by $\Rightarrow$, and product types by $\times$. 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 $\tau_1 \Rightarrow \tau_2$, and t is a term of type τ_1.

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

$$\mathsf{datatype}\ (\alpha_1, \ldots, \alpha_n) = C_1\ \tau_{11} \ldots \tau_{1k_1} \mid \ldots \mid C_m\ \tau_{m1} \ldots \tau_{mk_m}$$

where α_i are distinct type variables (the parameters), C_i are distinct constructor names and τ_{ij} are types. Note that n can be 0, i.e., there is no type parameters in datatype declaration.

Definition commands. consts command declares a function's name and type. defs gives the definition of a declared function. constdefs combines the effect of consts and defs. For instance, the following commands define a square function on nat.

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

consts S::τ set inductive S intros

$rule_1$: $[|a_{11} \in S; ...; a_{1k_1} \in S; A_{11}, ..., A_{1i_1}|] \Longrightarrow a_1 \in S$ ... $rule_n$: $[|a_{n1} \in S; ...; a_{nk_n} \in S; A_{n1}, ..., A_{ni_n}|] \Longrightarrow a_n \in S$

Lemmas. Lemmas are presented by the notation $[\![A_1; A_2; ...; A_n]\!] \Longrightarrow B$, which means that with assumptions $A_1, \ldots, A_n$, 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).

$$\mathsf{cktSat}\ nl\ (A \leadsto A)$$

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

Lemma 44 (steTrans).

$$[\![\mathsf{cktSat}\ nl\ (A \leadsto B); \mathsf{cktSat}\ nl\ (B \leadsto C)]\!] \Longrightarrow \mathsf{cktSat}\ nl\ (A \leadsto C)$$

Next rule steconjI splits the consequent of an STE assertion into individual conjuncts, which can be verified separately.

Lemma 45 (steconjI).

$$[\![\mathsf{cktSat}\ nl\ (A \leadsto B); \mathsf{cktSat}\ nl\ (A \leadsto C)]\!] \Longrightarrow \mathsf{cktSat}\ nl\ (A \leadsto B\ \mathtt{and}_{\mathtt{T}}\ C)$$

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

Lemma 46 (steImpI).

$$[\![g \Longrightarrow \mathsf{cktSat}\ nl\ (A \leadsto B)]\!] \Longrightarrow \mathsf{cktSat}\ nl\ (A \leadsto g \longrightarrow_{\mathtt{T}} C)$$

Rule steEnStrenAnt says that if $\mathsf{defSqOfTrForm}\ A' \sqsubseteq_{sq} \mathsf{defSqOfTrForm}\ A$, then assertions $A' \leadsto B$ imples $A \leadsto B$ because the antecedent A is stronger than A'.

Lemma 47 (steEnStrenAnt).

$[\![\mathsf{cktSat}\ nl\ (A' \leadsto B); \mathsf{defSqOfTrForm}\ A' \sqsubseteq_{sq} \mathsf{defSqOfTrForm}\ A]\!] \Longrightarrow \mathsf{cktSat}\ nl\ (A \leadsto B)$

Rule steWeakenCons says that if defSqOfTrForm $B \sqsubseteq_{sq}$ defSqOfTrForm B', then assertions $A \leadsto B'$ implies $A \leadsto B$ because the consequent B is weaker than A'. .

Lemma 48 (steWeakenCons).

$[\![\mathsf{cktSat}\ nl\ (A \leadsto B'); \mathsf{defSqOfTrForm}\ B \sqsubseteq_{sq} \mathsf{defSqOfTrForm}\ B']\!] \Longrightarrow \mathsf{cktSat}\ nl\ (A \leadsto B)$

Lemma steAndComm and steAndAssoc say that operator and_T satisfies commutative and associative laws.

Lemma 49 (steAndComm). $\mathsf{defSqOfTrForm}\ (A\ \mathsf{and}_\mathsf{T}\ B) = \mathsf{defSqOfTrForm}\ (B\ \mathsf{and}_\mathsf{T}\ A)$

Lemma 50 (steAndAssoc). $\mathsf{defSqOfTrForm}\ ((A\ \mathsf{and}_\mathsf{T}\ B)\ \mathsf{and}_\mathsf{T}\ C) =$ $\mathsf{defSqOfTrForm}\ (A\ \mathsf{and}_\mathsf{T}\ (B\ \mathsf{and}_\mathsf{T}\ C))$

A conjunct $(\mathsf{False} \longrightarrow_\mathsf{T} B)$ can be safely eliminated from a trjectory formula.

Lemma 51 (elimFalseGuard). $\mathsf{defSqOfTrForm}\ (A\ \mathsf{and}_\mathsf{T}\ (\mathsf{False} \longrightarrow_\mathsf{T} B)) =$ $\mathsf{defSqOfTrForm}\ \mathsf{A}$

A trajectory formula $\mathsf{True} \longrightarrow_\mathsf{T} A$ is equivalent to A.

Lemma 52 (simpTrueGuard). $\mathsf{defSqOfTrForm}\ (\mathsf{True} \longrightarrow_\mathsf{T} A) = \mathsf{defSqOfTrForm}\ \mathsf{A}$

chaos is the unit of the operator and_T.

Lemma 53 (andChaosId). $\mathsf{defSqOfTrForm}\ (A\ \mathsf{and}_\mathsf{T}\ \mathsf{chaos}) = \mathsf{defSqOfTrForm}\ A$

defSqOfTrForm is congruent for operator and_T.

Lemma 54 (steAndCong). $\mathsf{defSqOfTrForm}\ (A\ \mathsf{and}_\mathsf{T}\ B) = \mathsf{defSqOfTrForm}$ $(A\ \mathsf{and}_\mathsf{T}\ B')$ *if* $\mathsf{defSqOfTrForm}\ B = \mathsf{defSqOfTrForm}\ B'$.