Formalization and Execution of STE in HOL

Ashish Darbari

Computing Laboratory
University of Oxford
Oxford, England, OX1 3QD

Abstract. We present an early implementation of STE model checking in the higher-order logic theorem prover HOL. Our results are based on an earlier work done by [1, 2] in combining STE with theorem proving. By way of formalizing the results presented in [1], we have an initial platform for executing STE semantics directly in HOL. We can relate correctness results of the STE logic to the Boolean logic of HOL. We show how any trajectory assertion that is validated to true in STE, can be translated to an equivalent theorem in HOL. To this end we have extended the work presented in [1, 2] by implementing not only the theoretical results of [1, 2] but also incorporating the core STE implementation presented in [3]. As a useful benefit of proving the lemmas and theorems on machine, we discovered a flaw in the proof of one of the lemmas presented in [2].

1 Introduction

One of the main challenges posed to verification engineers today is to manage the size of the verification problem. Classic verification techniques like symbolic model checking typically suffer from the state explosion problem. However the degree to which they allow automation, and the expressivity of the language of the model checker, makes them very useful for verifying complex temporal properties.

Deductive verification techniques like theorem proving can handle a verification task of any size, but at the cost of manual intervention. Even the very best state-of-the art theorem provers require substantial manual guidance throughout the proof. To overcome the limitations of each of the above verification approaches, the idea is to blend them together to exploit the strength of each one of them, and alleviate the weaknesses. The work presented in this paper falls in the general area of combining model checking with theorem proving. Specifically we are investigating the combination of symbolic trajectory evaluation based model checking with higher order logic theorem proving.

Symbolic trajectory evaluation [3] or STE in short, is a highly effective model checking technique for datapath verification [4]. It has been combined with theorem proving to verify complex industrial designs [5, 6].

Aagaard et.al. in [1] outlined the theoretical foundation for linking the general logic of STE with higher order logic. They outlined the issues involved in making such a combination, and then presented a cohesive theory of integration, proving lemmas and theorems which justify a sound semantic link between STE and theorem proving. However, in that paper they did not provide formal proofs of the lemmas and the theorems. In the extended technical report [2] they provided proofs of some of the lemmas and theorems on paper. They claimed in their paper and the report that using the lemmas and theorems we can in principle verify properties using STE model checking and deduce the equivalent theorems in higher order logic. They however did not give any implementation details to show how this can be achieved in practice.

In this paper we provide the implementation machinery for integrating STE model checking with higher order theorem proving based on the results presented in [1, 2]. We show in this paper the implementation details of embedding the STE logic in a higher order theorem prover `HOL`.[1] By having an implementation of the theory of STE, and the semantic link to higher order logic, we are able to execute STE directly in `HOL`. Our implementation allows us to check properties using STE, and if the property is valid, we get an equivalent theorem in `HOL`, thereby achieving exactly what was envisioned by Aagaard et.al. in [1, 2]. We have tested a few examples using our implementation and the initial results are encouraging.

As a side benefit of mechanizing the theory presented in [1, 2], we discovered that there is a discrepancy in the proof of one of the lemmas in [2]. We shall talk more about it when we show the mechanized version of that lemma.

We are not able to present in this paper, the machine proofs and the example because of lack of space. The script file with all the details is available online.[2]

1.1 Related Work

Many researchers in the past have considered this problem of integrating the model checking with theorem proving [5, 7–9].

Rajan et.al. [7] have presented an integration of a BDD based model checker for propositional μ-calculus with the PVS theorem prover. They argued that μ-calculus serves as a good basis for combining model checking with theorem proving.

The long term goal of our research is to combine STE with theorem proving for verifying large circuit designs. We have been greatly inspired by the work done by Aagaard et.al. [5]. They claim that their verification effort resulted in the discovery of eight previously unknown bugs, four of which were high quality bugs – meaning they would not have been diagnosed with traditional validation techniques.

The work we are presenting in this paper however comes closest to [1, 2, 8, 10].

[1] `HOL` here stands for the theorem prover `HOL` 4, Kananaskis 1

[2] http://users.comlab.ox.ac.uk/ashish.darbari/Research/TPHOLS03/

Joyce and Seger in the past have worked on combining the Voss system with the HOL theorem prover [8]. They focused on Voss specific implementation of STE. We took the approach advocated by Aagaard et.al. [1, 2] in actually integrating the general logic of STE with the HOL theorem prover.

Aagaard et.al. in [10] proposed a solution of combining STE with theorem proving using a strongly typed functional language in the ML family, called fl. They lifted the language to make it reflective similar to Lisp. This gives them a possibility of executing fl functions and also to reason about the behavior of fl functions. The link from theorem proving to model checking is established in their approach by evaluating the lifted fl expressions.

The theorem proving support they offer in the lifted-fl language is an LCF-style implementation. However the core of the theorem prover is a set of trusted tactics and is not not fully expansive. Tactics work backwards and do not allow forward proofs.

1.2 Organization of the Paper

In this section we outline the road map of our paper. We start in the next section by presenting an outline of the basic STE theory. We show fragments of our implementation of the definitions of the basic concepts of the STE theory in `HOL`. In Section 3 we present the formalization of the link between STE and the two valued Boolean logic, providing relevant details together with the lemmas and theorems which justify the link.

In Section 4 we show how to use the combined formalization of STE theory and the linkage with `HOL`, to execute STE on an example design. In the last section we present conclusions and point to future directions.

2 Symbolic Trajectory Evaluation

Symbolic trajectory evaluation [3], combines the ideas of *ternary modelling* with *symbolic simulation*. In ternary modelling the binary set of values {0, 1} is extended with a third value X which indicates an unknown logic value. By assuming a monotonicity property of the simulation algorithm one can ensure that any binary value resulting when simulating patterns containing X's would also result when X's are replaceed by 0's and 1's. Thus the number of patterns that must be simulated to verify a circuit are reduced dramatically by representing many different operating conditions by patterns containing X's. With ternary simulation, a state with some nodes set to X covers those circuit states obtained by replacing the X values with either a 0 or a 1. The state with all nodes set to X thus covers all possible actual circuit states.

Although ternary modelling allows us to cover many conditions with a single simulation run, it lacks the power required for complete verification, except for a small class of circuits such as memories [11].

Symbolic trajectory evaluation extends the idea of ternary modelling by including the notion of time and the usage of symbolic Boolean variables. Using

STE one can specify and verify system behavior over time. By using symbolic Boolean variables and propositional logic expressions over these, we can represent whole classes of data values on circuit nodes. The Boolean expressions or BDDs respresenting values at different circuit nodes can have variables in common. These variables can record complex interdependencies among node values.

In the subsequent sections, we shall present an implementation of STE in HOL. Readers unfamiliar with the detailed theory of STE and the syntax of HOL are referred to [1, 3].

2.1 The four valued lattice

Symbolic trajectory evaluation employs a ternary circuit state model, in which the usual binary values 0 and 1 are augmented with a third value X that stands for an unknown. To represent this mathematically, we introduce a partial order relation $\sqsubseteq$, with $X \sqsubseteq 0$ and $X \sqsubseteq 1$. The relation orders values by information content: X stands for a value about which we know nothing, and so is ordered below the specific values 0 and 1.

To develop a smooth mathematical theory for STE, we add a further value $\top$ (called 'top') to get the set of circuit node values $\mathcal{D} = \{0, 1, X\} \cup \top$. We extend the ordering relation to make $(\mathcal{D}, \sqsubseteq)$ a complete lattice.
We use the idea of dual-rail encoding [12], to define the four lattice values in the STE logic.

```
(* Four lattice values *)
 |- Top = (F,F)
 |- One = (T,F)
 |- Zero = (F,T)
 |- X = (T,T)
```

Please note that the choice of `Top`, being defined as (`F`, `F`) and other values like `X` being (`T`, `T`) is in principle arbitrary, any possible permutation on the two Boolean values `T`, and `F` can be chosen to denote the four values. However efficient definition of least upper bound, and the ordering $\sqsubseteq$, does depend crucially on a particular permutation of `T` and `F` that we choose for representing the lattice values.

```
(* Least upper bound (lub) *)
 |- ∀a b c d. lub (a,b) (c,d) = (a ∧ c,b ∧ d)

(* Information ordering *)
 |- ∀a b. leq a b = (b = lub a b)
```

Observable points in circuits are nodes. Nodes can be defined by the HOL type `string`. A lattice state is then defined as an instantaneous snapshot of circuit behavior given by an assignment of lattice values to nodes. If we denote the lattice state by `s` then

```
s: string->(bool # bool)
```

A lattice sequence assigns a lattice value to each node at each point in time. Time is just the set of natural numbers (`num` in `HOL`). If we denote the lattice sequence by `sigma` then

```
sigma: num->string->(bool # bool)
```

Because of lack of space here, we do not show all the functions that we have implemented in `HOL`. However, we do think its important to mention them because we use them in other definitions. Some of these functions we wrote in `HOL` are the `Suffix` and the extension of the information ordering on lattice states (`leq_state`) and sequences (`leq_seq`).

2.2 Circuit Models in STE

In STE, the formal model of a circuit is given by a next-state function `Y_ckt` that maps lattice states to lattice states:

```
Y_ckt: (string->(bool # bool))->(string->(bool#bool))
```

Intuitively, the next-state function expresses a constraint on the set of possible states into which the circuit may go for any given state. In implementations of STE, the circuit model `Y_ckt` is constructed incrementally and piecemeal by ternary symbolic simulation of an HDL or a netlist source for the circuit. In our presentation here the circuit is uninterpreted. To run a typical example using our STE formalization requires one to define the model `Y_ckt` completely.[3]

One crucial property the next-state function needs to preserve is the property of monotonicity. Any next-state funcntion is monotonic if for all lattice states `s` and `s'`, if `s` $\sqsubseteq$ `s'` then state obtained by applying the next-state function (`Y_ckt`) on `s` is also less than or equal to ($\sqsubseteq$) the state reached by applying `Y_ckt` to `s'`.

Since sequences return lattice values for each node at a given time point, a sequence encodes a set of behaviors that a circuit can exhibit. The next-state function or the lattice model of the circuit provides the meaning of circuit behavior. Now we shall define what it means for a sequence to be in the lattice model of a circuit. A sequence is in the language of a lattice model of a circuit if the set of behaviors that the sequence encodes is a subset of the beahviors that the circuit can actually exhibit. Below we show the formalization in `HOL`.

```
(* Lattice sequence in the language of a circuit *)
 |- ∀sigma Y_ckt.
        in_STE_lang sigma Y_ckt =
        ∀t. leq_state (Y_ckt (sigma t))(sigma (t + 1))
```

[3] We will show in a later section, how we do this for a specific circuit.

2.3 Syntax of STE

In STE, the basic syntactic entity used in specifcation is a symbolic trajectory formula. In formalizing the definition of STE syntax, we define a new type `TF`, of trajectory formulas in `HOL`.

```
(* Syntax of trajectory formula *)
 val _ = Hol_datatype
   `TF =
   Is_0 of string
 | Is_1 of string
 | AND of TF => TF
 | WHEN of TF => bool
 | NEXT of TF`;
```

We have a deep embedding [13–15] of all the operators `Is_0`, `Is_1`, `AND`, `WHEN` and `NEXT`. However we have chosen to represent the guard [1, 2] shallowly by actually using the `HOL` type `bool` to represent the guard.[4] This is to allow us to inherit the theory of booleans (`bool`) in `HOL`. The HolBdd [16] package is also interfaced to the type `bool` in `HOL`, possibly later we can use the HolBdd package, without having to invest too much effort (we won't have to reinvent the theory of Booleans and link them with BDDs).

2.4 Semantics of STE

We now define the semantics of the trajectory formula in `HOL`. We formalize in `HOL`, the function `SAT_STE`, that defines when a trajectory formula is satisfied by the lattice sequence.

```
(* Sequence satisfying a formula *)
 |- (∀n. SAT_STE (Is_0 n) = (λsigma. leq Zero (sigma 0 n)))
 ∧ (∀n. SAT_STE (Is_1 n) = (λsigma. leq One (sigma 0 n)))
 ∧  (∀tf1 tf2. SAT_STE (tf1 AND tf2) = (λsigma. SAT_STE tf1 sigma
                                      ∧ SAT_STE tf2 sigma))
 ∧ (∀tf P.SAT_STE (tf WHEN P) = (λsigma. P ==> SAT_STE tf sigma))
 ∧ (∀tf. SAT_STE (NEXT tf) = (λsigma. SAT_STE tf (Suffix 1 sigma)))
```

[4] Angelo et.al. in [15] and Boulton et.al. in [14] present interesting case studies of different kinds of embeddings of hardware description languages.

2.5 STE Verification Engine

Verification in STE takes place by testing the validity of an assertion of the form

```
Ant ==>> Cons
```

where `Ant` and `Cons` are trajectory formulas having the abstract type `TF` in `HOL` formalization, and `==>>` is a constructor that takes two elements of type `TF` and returns an element of an abstract type `Assertion` in `HOL`.

The function that checks the validity of such an assertion is formalized in `HOL` as `SAT_CKT`

```
(* Validity of a trajectory assertion *)
 |- ∀Ant Cons Y_ckt.
        SAT_CKT (Ant ==>> Cons) Y_ckt =
          ∀sigma. in_STE_lang sigma Y_ckt ==>
                    ∀t. SAT_STE Ant  (Suffix t sigma)  ==>
                                  SAT_STE Cons (Suffix t sigma)
```

Seger and Bryant in [3] proposed an implementation algorithm of STE. They introduced the idea of a defining sequence and a defining trajectory. They then argued that any trajectory assertion of the form `Ant ==>> Cons` can be verified by the STE implementation if and only if the defining sequence of `Cons` is less than or equal to ($\sqsubseteq$) the defining trajectory of `Ant`, for all nodes mentioned in the assertion and for all time points upto the depth of `Cons`.

We have defined the functions to calculate the defining sequence (`DefSeq`) and the defining trajectory (`DefTraj`) in `HOL`, unfortunately we cannot show their formalized definition here due to lack of space.

We now state the STE implementation (`STE_Impl`) algorithm [3], that takes an assertion and a circuit model `Y_ckt` and computes a symbolic constraint (over the free variables appearing in the guard of the trajectory formulas in the assertion) under which the assertion will be valid. The strength of this implementation algorithm lies in the fact that it is sufficient to compute finite segments of the defining sequence and the defining trajectory, to completely verify the assertion even though in theory both the defining sequence and the defining trajectory is infinite. The depth of the segment is computed from the depth of the consequent in the assertion.

```
(* STE implementation *)
 |- ∀Ant Cons Y_ckt.
        STE_Impl (Ant ==>> Cons) Y_ckt = ∀t. t <= Depth Cons ==>
              ∀n. MEMBER n (Nodes Ant Append Nodes Cons) ==>
                        leq (DefSeq Cons t n) (DefTraj Ant Y_ckt t n)
```

The function `Depth` calculates the number of `NEXT` operators in a trajectory formula. The function `Nodes`, calculates the list of nodes in a given formula, and

the function MEMEBR checks for the occurence of a node in a given list of nodes. Append is the usual append on lists. We have defined all these functions in HOL.

We now present the theorem[5] that makes an assertion about the correctness of the STE algorithm.

The theorem states that the trajectory assertion is valid for a circuit with model Y_ckt if and only if the STE implementation guarantees that the trajectory assertion is valid for the model Y_ckt.

```
(* Theorem 1: Correctness of STE algorithm *)
|- ∀Ant Cons Y_ckt.
       SAT_CKT (Ant ==>> Cons) Y_ckt
                      = STE_Impl (Ant ==>> Cons) Y_ckt
```

3 From lattice world to the relational world

The language of the theorem prover HOL is based on a Boolean logic. Hence in order to make a connection between STE and HOL, we have to address the key problem of connecting the four valued STE logic to a two-valued Boolean logic. This entails addressing the following issues

- defining when the embedded STE trajectory formulas are satisfied by a Boolean valued sequence
- defining a connection between the lattice values and the Boolean values
- identifying a connection between the circuit model in STE world and the circuit model in the Boolean world
- relating correctness results in the STE world to correctness results in the Boolean world

We shall discuss these issues in subsequent sections.

3.1 Semantics of Trajectory Formulas in Boolean Logic

States in the Boolean world are functions from the set of nodes $\mathcal{N}$ to the Boolean set $\mathcal{B}$, where $\mathcal{B} = \{\mathtt{T}, \mathtt{F}\}$. We refer to the states in the Boolean world as *Boolean states*. The set $\mathcal{B}$ is the set bool in HOL. Using the type string to denote the set of nodes $\mathcal{N}$ we shall represent a Boolean state by subscripting the letter s with b.

```
s_b: string->bool
```

A *Boolean sequence* is a function which returns a Boolean state, at given point of time. We denote the Boolean sequence by sigma_b in HOL and time is denoted by the type num.

```
sigma_b: num->string->bool
```

[5] At present we have used this theorem as an axiom in HOL, since we are not finished with the proof yet.

We shall now define the function SAT_BOOL that defines when a trajectory formula is satisfied by a Boolean sequence sigma_b.

```
(* Boolean sequence satisfies a trajectory formula *)
 |- (∀n. SAT_BOOL (Is_0 n) = (λsigma_b. sigma_b 0 n = F))
 ∧ (∀n. SAT_BOOL (Is_1 n) = (λsigma_b. sigma_b 0 n = T))
 ∧ (∀tf1 tf2.
       SAT_BOOL (tf1 AND tf2) =
          (λsigma_b. SAT_BOOL tf1 sigma_b ∧ SAT_BOOL tf2 sigma_b))
 ∧ (∀tf P.
       SAT_BOOL (tf WHEN P) = (λsigma_b. P ==> SAT_BOOL tf sigma_b))
 ∧ (∀tf.
       SAT_BOOL (NEXT tf) =
         (λsigma_b. SAT_BOOL tf (Suffix_b 1 sigma_b)))
```

The function Suffix_b is defined in a way similar to the function Suffix. Suffix_b returns the i^{th} suffix of a Boolean sequence.

3.2 Relating Lattice values to Boolean values

We define an operation called drop which drops the values from the Boolean world to the values in the STE world.

```
(* Dropping from Boolean to lattice Values *)
 |- (drop T = One) ∧ (drop F = Zero)
```

We shall need the point wise extension of the drop operation on states and sequences, in order to define some useful lemmas later. Lifting the drop operation pointwise, we can relate the lattice valued states and sequences to the Boolean valued states and sequences as

```
(* Drop operation lifted over states *)
 |- ∀s_b. extended_drop_state s_b =  (λnode. drop (s_b node))
(* Definition : Drop operation lifted over sequences *)
 |- ∀sigma_b.
        extended_drop_seq sigma_b =
                       (λt. extended_drop_state (sigma_b t))
```

3.3 Relational Circuit Model

A circuit in the Boolean world, is modelled by a next-state relation, which for a given circuit gives a relation between present and next Boolean state. The circuit is uninterpreted here similar to the way it was in the definition of the lattice model. While running example circuits, we define the relational model of a circuit in HOL. We will show in a later section how we accomplish this for a concrete example.

```
Yb_ckt: (string->bool)->(string->bool)->bool
```

3.4 Boolean Sequence in the language of the circuit

A Boolean valued sequence is in the language of the circuit, with the relational model `Yb_ckt`, iff the consecutive Boolean valued states are included in the next-state relation `Yb_ckt`.

```
(* Boolean sequence is in the language of a circuit *)
 |- ∀sigma_b Yb_ckt.
        in_BOOL_lang sigma_b Yb_ckt =
        ∀t. Yb_ckt (sigma_b t) (sigma_b (t + 1))
```

3.5 Relating circuit models in STE and Boolean World

We have made connections between Boolean and lattice valued states and sequences. In order to make a sound connection between the functional circuit model in the STE world with the relational circuit model of the Boolean world, we need to make sure that the two models of the circuit (`Y_ckt` and `Yb_ckt`) describe the same behavior.

Intuitively, for a given circuit, with a relational model `Yb_ckt` and the lattice model `Y_ckt`, the two circuit models describe the same circuit, if and only if for any two Boolean states `s_b` and `s_b'` (where `s_b` is the present state and `s_b'` is the state at next point of time) if `s_b` and s_b' are related by the relational model `Yb_ckt`, then the lattice model `Y_ckt` when applied to the drop of the present Boolean state `s_b` should return a lattice value, that conveys information less than or equal ($\sqsubseteq$) to, the information conveyed by the lattice value returned by the drop of the next Boolean state `s_b'`.

We define the predicate `Okay` in `HOL`, that asserts when the two circuit models describe the same circuit.

```
(* Linking Boolean and lattice models *)
 |- ∀Y_ckt Yb_ckt.
        Okay (Y_ckt,Yb_ckt) =
        ∀s_b s_b'.
          Yb_ckt s_b s_b' ==>
          leq_state (Y_ckt (extended_drop_state s_b))
                    (extended_drop_state s_b')
```

3.6 Relating Correctness Results

In this section we shall relate the correctness results from the STE world to the Boolean world. The intuition is that any trajectory assertion that is satisfied by lattice valued sequence should be satisfied by the Boolean valued sequence.

Before we state the theorem that relates the correctness results between the two worlds, we shall state two lemmas which we have used in the proof of the theorem.

```
(* Lemma 1: Relating Boolean and lattice valued sequences *)
 ∀Y_ckt Yb_ckt.
          Okay (Y_ckt, Yb_ckt) ==>
            ∀sigma_b. in_BOOL_lang sigma_b Yb_ckt ==>
                          in_STE_lang (extended_drop_seq sigma_b) Y_ckt
```

The lemma states a fact that whenever the two circuit models `Y_ckt` and `Yb_ckt` talk about the same circuit (i.e. satisfy the property `Okay`) then for every Boolean sequence which is in the relational model of the circuit, the drop of the Boolean sequence is in the lattice model of the circuit.

As mentioned earlier in the introduction, the proof of Lemma 1 presented in [2] is incorrect. The lemma as stated in [2], says that whenever two circuits satisfy the property `Okay` (Axiom 2 in [2]), then every Boolean sequence is in the language of the relational model of the circuit if and only if the drop of the Boolean sequence is in the lattice model of the circuit. The proof of the lemma relies on Axiom 2 (in [2]), which is stated as an implication. Just by using the implication in Axiom 2, we cannot prove the equivalence property of the lemma [2].

Our claim here is that either we state both Axiom 2 and Lemma 1 (in [2]) as an implication or state them both as an equivalence. We chose to keep an implication in the definition of `Okay` (the counterpart of Axiom 2), since it gives us enough power to say what we wanted to say, and we also state `Lemma 1` (the counterpart of Lemma 1 in [2]) as an implication. Interestingly, in the paper [1] the authors have stated the Axiom and the Lemma both as an implication.

We now state a lemma below which captures the fact that a trajectory formula is satisfiable by a Boolean sequence if and only if it is satisfiable by the drop of the Boolean sequence.

```
(* Lemma 2: Relating satisfaction over Boolean and lattice valued
            sequences *)
 ∀tf seq_b. SAT_BOOL tf seq_b = SAT_STE tf (extended_drop_seq seq_b)
```

Now we are in a position to state `Theorem 2`. `Theorem 2` states that for a given circuit with lattice model `Y_ckt` and the Boolean model `Yb_ckt`, if `Y_ckt` and `Yb_ckt` satisfy the property `Okay`, then if a given trajectory assertion is satisfied by the lattice model, then for all Boolean valued sequences which are in the language of the Boolean model `Yb_ckt`, for all time points `t` greater than zero, if the antecedent of the trajectory assertion (`Ant`) is satisfied by the $\texttt{t}^{th}$ suffix of the Boolean valued sequence, then the consequent of the trajectory assertion `Cons` is also satisfied by the $\texttt{t}^{th}$ suffix of the Boolean valued sequence.

```
(* Theorem 2: Correctness in STE world implies correctness
              in the Boolean world *)

|- ∀Ant Cons Y_ckt Yb_ckt.
       Okay (Y_ckt,Yb_ckt) ==>
       SAT_CKT (Ant ==>> Cons) Y_ckt ==>
       ∀sigma_b.
         in_BOOL_lang sigma_b Yb_ckt ==>
         ∀t.
           SAT_BOOL Ant (Suffix_b t sigma_b) ==>
           SAT_BOOL Cons (Suffix_b t sigma_b)
```

`Theorem 2` forms the crux of the connection between the lattice world and the relational world. It gives us the power to link the correctness statements in STE world to the notion of correctness in the relational world. This means we can use the STE verification engine to compute a symbolic constraint under which a trajectory assertion would be valid, and then infer a corresponding Theorem in the relational world. These theorems in the Boolean world are the theorems we intuitively expect to hold when the property stated in the trajectory assertion is verified independently in the theorem prover.

Of course, the validity of `Theorem 2` and `Lemma 1` relies on the fact that it is possible to translate the correctness statements from the STE world to the Boolean world only if the circuit models in the two worlds satisfy the property `Okay`.

In the next section we show how we combine `Theorem 1` and `Theorem 2` to prove for the unit-delay Nand gate[6] that if an implementation of STE algorithm (`STE_Impl`) returns the value `T`, then we can get an equivalent theorem in `HOL`.

4 Executing STE in HOL

In this section we illustrate the example of a two input unit-delay Nand gate, whose output is tied to one of its inputs (see [1, 2]). We define the lattice and the Boolean models for such a circuit. Below is an example we wrote in `HOL`.

[6] We have taken the example from [1, 2]

```
(* Definition of Not, And and Nand using dual-rail encoding *)
|- ∀a b. Not (a, b) = (b, a)
|- ∀a b c d. And (a, b) (c, d) = (a ∧ c, b ∨ d)
|- ∀a b. Nand a b = Not (And a b)

(* Definition of lattice model for the unit delay Nand gate *)
|- ∀s node.
       Nand_lattice s node =
       (if node = "in" then
          X
        else
          (if node = "out" then Nand (s "in") (s node) else X))

(* Definition of the relational model for the unit delay Nand gate *)
|- ∀s_b s_b'.
       Nand_bool s_b s_b' =
       ∀node.
         ((node = "out") ==> (s_b' node = ~(s_b "in" ∧ s_b node))) ∧
         ((node = "in") ==> s_b' node ∨ ~s_b' node)
```

We then write the STE assertions that we need to verify, using the STE implementation `STE_Impl`. Intuitively, if we assert the Boolean variables `v1` and `v2` on one of the input nodes and the output node respectively, then after one unit of time we can expect to observe the value $\neg$(`v1` $\wedge$ `v2`) at the output. Infact this is exactly what we assert in the STE assertion as shown below.

```
(* input node has a value v1 *)
val ant1 = ``(Is_1 "in" WHEN v1) AND (Is_0 "in" WHEN ~v1)``

(* output node has a value v2 *)
val ant2 = ``((Is_1 "out") WHEN v2) AND ((Is_0 "out") WHEN ~v2)``

(* Antecedent: "in" is v1 and "out" is v2 *)
val Ant = Term `^ant1 AND ^ant2`;;

(* Consequent: N("out" is ~(v1∧v2)) *)
val Cons = ``NEXT
      ((Is_1 "out" WHEN ~(v1 ∧ v2)) AND (Is_0 "out" WHEN (v1 ∧ v2)))``
```

We have written ML functions and (conversions[7] in `HOL`) to develop automated proof strategies which perform computation. Here we will present informally an outline.

[7] conversions have the type `term -> thm`

`Theorem 1` states the equivalence of `SAT_CKT` and the `STE_Impl`. We substitute `STE_Impl` for `SAT_CKT` in `Theorem 2` and we get an auxiliary theorem, that relates the `STE_Impl` and the satisfaction of trajectory assertion over Boolean sequence. We then take this auxiliary theorem and apply some of our hard-wired conversions on it, to eventually get the desired theorem in `HOL`.

We wrote the top-level function `STE_TO_BOOL` that takes an antecedent, a consequent, the lattice model of the circuit (‘‘`Nand_lattice`‘‘), the relational model (‘‘`Nand_bool`‘‘) and the string `"Nand"` and it computes a theorem in `HOL`. The string `"Nand"` actually tells the function `STE_TO_BOOL` to use the conversion written specifically for the Nand gate circuit.

```
- STE_TO_BOOL Ant Cons ‘‘Nand_lattice‘‘ ‘‘Nand_bool‘‘ "Nand";
  runtime: 14.100s,    gctime: 2.070s,     systime: 0.050s.
  Meson search level: ................................
> val it =
|- ∀v2 v1 sigma_b.
    in_BOOL_lang sigma_b Nand_bool ==>
    ∀t.
     (sigma_b t "in" = v1) ∧ (sigma_b t "out" = v2) ==>
     (sigma_b (t + 1) "out" = ~(sigma_b t "in" ∧ sigma_b t "out"))
```

In the above example the trajectroy assertion is true, for any assignment of Boolean values to the variables `v1` and `v2`. So the STE implementation in this case returns the value `T`.

If the STE implementation doesn't return the value `T` but instead returns a symbolic Boolean expression (residual), even then we can get an equivalent theorem in `HOL`, however that theorem will have the residual as an assumption. We are at present working on developing functions that will take the residual and come up with a counter examples or sets of satisfying valuations that will make the residual `T`. One possibility we are considering is to use the HolSatLib [17] package. At the moment we have not completely investigated this, but this definitely something for future work.

The functions and conversions that we have written have two components, one is a fairly general component that can take any circuit model and do some pre-processing; the other specific component is tailored to handle the proof of the `Okay` property for each specific circuit in question. Since the proof for each circuit in question depends on the model definitions, it seems impractical to have one general purpose proof routine for every circuit.

5 Conclusion and Future Work

In this paper we presented the formalization of the STE logic in `HOL`. We formalized the results presented in Aagaard et.al's work [1] on linking trajectory evaluation to higher-order logic. We also extended their idea by writing functions that implement the core STE algorithm known from [3] and show that one

can execute the semantics of STE directly in a theorem prover like HOL. In this process, we wrote special purpose proof strategies (conversions) that we used to advance the computation of the STE Implementation and reach to a point where we get theorem in HOL.

Since our work is in a preliminary stage, we cannot yet compare our implementation with Aagaard et.al. [10]. It seems it will be very useful to compare and also draw on their experience of doing a similar task, specially because they use a language (lifted-fl) specially tailored for this kind of task. The language allows representation of Boolean expressions as BDDs. This gives them a seamless integration of model checking and theorem proving. In our case, we don't model guards in STE by BDDS.

In HOL the Booleans and the BDDS are two different types. We will need to stitch them together possibly using the HolBdd package [16]. At the moment we have not completely investigated the usage of HolBdd and the ramifications it will have on our work. This is one of the goals we have set for immediate future work in this area. Together with BDDs we intend to experiment with other abstraction ideas which can help us reduce the verification effort of large circuit designs.

We are also working on making the function STE_Impl more efficient, and optimizing other functions and conversions that we have.

In the process of formalizing the theory of STE we have uncovered a bug in the proof of one of the lemmas stated in the technical report [2]. Although the discrepancy isn't a major bug in the report, we believe our effort in uncovering it is well worth it.

6 Acknowledgment

Thanks are due to Mike Gordon for arranging my visit to the Computing Laboratory at Cambridge and motivating me to work on this. Michael Norrish at Cambridge provided me some useful hints on how to get started with HOL. John O' Leary, Jim Grundy and Robert Jones at Intel, gave me useful feedback in early stages of my work. Myra VanInwegen gave useful feedback on an early draft of the paper.

Special thanks to Tom Melham who really made a difference with his expertise in the subject and has provided continuous help and inspiration.

This work has been supported in part by a research grant from Intel Corp. USA.

References

1. M. D. Aagaard, T. F. Melham, and J. W. O'Leary, "Xs are for trajectory evaluation, Booleans are for theorem proving," in *Correct Hardware Design and Verification Methods: 10th IFIP WG10.5 Advanced Research Working Conference: Bad Herrenalb, September 1999: Proceedings*, L. Pierre and T. Kropf, Eds. 1999, vol. 1703 of *Lecture Notes in Computer Science*, pp. 202–218, Springer-Verlag.

2. M. D. Aagaard, T. F. Melham, and J. W. O'Leary, "Xs are for trajectory evaluation, Booleans are for theorem proving (extended version)," Tech. Rep. TR-2000-52, Department of Computing Science, University of Glasgow, January 2000.
3. C.-J. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, March 1995.
4. M. D. Aagaard, R. B. Jones, T. F. Melham, J. W. O'Leary, and C.-J. H. Seger, "A methodology for large-scale hardware verification," in *Formal Methods in Computer-Aided Design: Third International Conference, FMCAD 2000: Austin, November 2000: Proceedings*, Jr. W. A. Hunt and S. D. Johnson, Eds. 2000, vol. 1954 of *Lecture Notes in Computer Science*, pp. 263–282, Springer-Verlag.
5. Mark Aagaard, Robert B. Jones, and Carl-Johan H. Seger, "Combining theorem proving and trajectory evaluation in an industrial environment," in *Design Automation Conference*, 1998, pp. 538–541.
6. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of Boolean constraints," in *ACM/IEEE Design Automation Conference*, July 1998.
7. S. Rajan, N. Shankar, and M. K. Srivas, "An integration of model checking with automated proof checking," in *Proceedings of the 7th International Conference On Computer Aided Verification*, P. Wolper, Ed., Liege, Belgium, 1995, vol. 939, pp. 84–97, Springer Verlag.
8. J. Joyce and C.-J. Seger, "Linking BDD based symbolic evaluation to interactive theorem proving," in *ACM/IEEE Design Automation Conference*, June 1993.
9. Klaus Schneider and Dirk W. Hoffmann, "A HOL conversion for translating linear time temporal logic to omega-automata," in *Theorem Proving in Higher Order Logics*, 1999, pp. 255–272.
10. Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger, "Lifted-fl: A pragmatic implementation of combined model checking and theorem proving," in *Theorem Proving in Higher-Order Logics*. September 1999, Springer-Verlag.
11. R. Bryant, "Formal verification of memory circuits by switch-level simulation," *IEEE Transactions on ComputerAided Design of Integrated Circuits and Systems*, January 1991, Vol.10, no.1, pp. 94-102.
12. C.-J. H. Seger, "Voss — a formal hardware verification system: User's guide," Tech. Rep. TR-93-45, University of Brtish Columbia Department of Computer Science, December 1993.
13. M.J.C. Gordon, "Mechanizing programming logics in higher-order logic," in *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, G.M. Birtwistle and P.A. Subrahmanyam, Eds., Banff, Canada, 1988, pp. 387–439, Springer-Verlag, Berlin.
14. R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel, "Experience with embedding hardware description languages in HOL," in *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, Nijmegen, 1992, pp. 129–156, North-Holland.
15. C. M. Angelo, L. Claesen, and H. De Man, "Degrees of formality in shallow embedding hardware description languages in hol," in *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop (HUG'93)*, J. J. Joyce and C.-J. H. Seger, Eds., pp. 89–100. Springer, Berlin, Heidelberg, 1994.
16. Mike Gordon, *HolBddLib, Version 2*, Computer Laboratory, University of Cambridge, March 2002.
17. Mike Gordon, *HolSatLib Documentation, Version 1.0*, Computer Laboratory, University of Cambridge, October 2001.