Programming Research Group

FORMALIZATION AND EXECUTION OF STE IN HOL (EXTENDED VERSION)

Ashish Darbari

PRG-RR-03-17

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

We present an implementation of STE model checking in the higher-order logic theorem prover HOL. The motivation for such an implementation is to have an open source framework in which we can exploit the strengths of STE model checking and HOL theorem proving. We intend to use this framework for future experiments in the design and implementation of abstraction and reduction techniques to extend the reach of STE model checking.

Contents

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. Our aim is have an open source framework in which we can experiment with new abstraction and reduction techniques for extending the reach of STE model checking for large scale verification.

Symbolic trajectory evaluation [1] or STE in short, is a highly effective model checking technique for datapath verification [2]. It has been combined with theorem proving to verify complex industrial designs [3, 4].

Aagaard *et al.* [5] outlined the theoretical foundation for linking the general logic of STE with higher order logic. They identified 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 [6] 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 report, we explain our implementation work on integrating STE model checking with higher order theorem proving based on the results presented in [5, 6]. We present the 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 *etal*. [5, 6]. 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 [5, 6], we discovered that there is a discrepancy in the proof of one of the lemmas in [6]. We will elaborate more on it when we show the mechanized version of that lemma.

The accompanying source code of the implementation is available online at `http:`

[1] HOL here denotes the higher order logic theorem prover HOL 4 Kananaskis 1

//users.comlab.ox.ac.uk/ashish.darbari/Research/STE-HOL/.

1.1 Related Work

Combination of model checking with theorem proving has attracted significant attention in the past decade. Some of the early work in this area was done on embedding modal mu-calculus in HOL [7] and in PVS [8]. Schneider *et al.* [9] worked on integrating LTL in HOL. PROSPER is an open source framework which examines linking of many different formal verification tools [10] including the model checker SMV and the LTL model checking with HOL98. Tomas Uribe presents a good survey on combining model checking with theorem proving [11].

Some of the the pioneering work done on combining STE model checking with HOL theorem proving was done by Joyce and Seger [12]. They combined the Voss [13] system with the HOL theorem prover to design a system called HOL-Voss. They focussed on Voss specific implementation of STE. Here the STE model checker was used as a decision procedure, and the only way to interact with it was to use it as a tactic. The user could not interact with the model checker directly. HOL-Voss was soon followed by VossProver another effort in the combination of STE with theorem proving. VossProver was a lightweight theorem prover built on top of Voss. An interesting account of the evolution of combining STE with theorem proving appeared in a paper published in 1999 by Aagaard *et al.* [14]. In this paper Aagaard *et al.* [14] proposed a solution of combining STE with theorem proving using fl. They however lifted the language to make it reflective similar to Lisp. This gave 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 was established in their approach by evaluating the lifted fl expressions, significant contribution was the fact that Boolean expressions were represented in the system by Binary Decision Diagrams (BDDs) [15]. The theorem proving support they offered in the lifted-fl language was an LCF-style implementation. However the core of the theorem prover was a set of trusted tactics and was not not fully expansive. Tactics worked backwards and did not allow forward proofs.

In 1999, Aagaard *et al.* [5] published a paper outlining the general concept of embedding STE in a higher order logic. This paper identified the theoretical details for linking the general logic of STE (as opposed to Voss) with the Boolean logic of HOL. Our work is an implementation of the theory presented in [5, 6].

1.2 Organization of the Paper

In this section we outline the road map of our paper. We start in the next section by describing our philosophy of integration. In the subsequent section we present an outline of the basic STE theory and show fragments of our implementation of the definitions of the basic concepts of the STE theory in HOL. In Section 4 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 5 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 Philosophy of Integration

Integrating the language of the model checker (object language) inside a theorem prover (meta language) entails the semantic embedding [16] of the logic of the model checker within the logic of the theorem prover. In the context of embedding hardware description languages in HOL, Boulton *et al.* [17] describe the two approaches of semantic embedding as deep and shallow.

A deep embedding is done by representing the abstract syntax of the object language in the theorem prover, and then define within the meta-logic of the theorem prover semantic functions that assign meanings to the syntax of the object language. A shallow embedding relies on a special interface to parse the syntax of the object language directly into the semantic structures within the logic of the theorem prover.

In the setting where HOL is the theorem prover of choice, a deep embedding of an object language would mean assigning HOL types to the syntax of the object language, and a shallow embedding would entail having ML types for the object language syntax.

Both deep and shallow embeddings have their advantages and disadvantages. It is usually the case that we do a shallow embedding of a fragment of the object language, and a deep embedding of the rest. Angelo *et al.* [18] present a detailed coverage of degrees of shallowness in embedding hardware description languages.

In the next section we will elaborate in more detail the theory of STE, but here we shall explain the role of embedding in our work. We present a deep embedding of STE logic in HOL. The basic entity in STE is a trajectory formula. A trajectory formula can have Boolean expressions in it. We do a shallow embedding of the Boolean expressions appearing in the trajectory formula, and a deep embedding of other syntactic entities which make a trajectory formula. The idea of using deep embedding for most of the trajectory formula is to be able to reason about them in the HOL logic, and prove properties about the STE logic in HOL. We prefer to adopt a shallow embedding of the Boolean expressions so that we can use the theory of Booleans in HOL, and we don't have to reinvent it.

The trade-off between deep and shallow embedding will become more clear in the light of our work on embedding STE theory in HOL, that we will present in the subsequent sections.

3 Symbolic Trajectory Evaluation

Symbolic trajectory evaluation [1], 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 replaced 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.

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 [19]. Inspite of having Xs, in the absence of symbolic variables, with ternary modelling one has to still verify a large number of cases for a memory like circuit.

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 be used to represent interdependencies among node values.

In the subsequent sections, we shall present an implementation of STE in HOL. For more details see [5, 1].

3.1 The Four Valued Lattice

The logic of Symbolic trajectory evaluation is three valued. The three values are the usual binary values 0 and 1 and a third value called X. The value X stands for the case when we do not know for sure if the value is 0 or 1 but it could be any of the two. Since the value X stands for 0 or 1 but we don't know which, it conveys less information than for example the value 0 where we know for sure that it is 0. Similarly X conveys less information than 1. Mathematically we can introduce an ordering relation ($\sqsubseteq$), with $X \sqsubseteq 0$ and $X \sqsubseteq 1$. The three valued set will be denoted by $\mathcal{D}$.

To develop a smooth mathematical theory for STE, we add a further value $\top$ (called 'top') and extend the ordering relation on $\mathcal{D}$ to include $\top$, to make a complete lattice. We use the idea of dual-rail encoding [13], to define the four lattice values in the STE logic. Below we show the definition of the dual-rail encoding in HOL.

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

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. We have used the dual-rail encoding from [13].

```
(* 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 we have the typing

```
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 we have the typing

```
sigma: num→string→(bool∗bool)
```

We can extend the notion of ordering on lattice values to get orderings on lattice states and lattice sequences.

```
(* Ordering lifted over states *)

⊢ ∀state1 state2. leq_state state1 state2 =
                               ∀node:string. leq (state1 node)(state2 node)‘;

(* Ordering lifted over sequences *)

⊢ ∀sigma1 sigma2. leq_seq sigma1 sigma2 =
                               ∀node:string t:num.
                                 leq (sigma1 t node) (sigma2 t node)‘;
```

A useful operation on sequences is calculating the Suffix of a sequence. It is formalized below.

```
(* Definition of suffix *)

⊢ ∀i sigma. Suffix i (sigma: num→string→(bool ∗ bool))
                               = λt node.(sigma (t+i) node)‘;
```

3.2 Circuit Models in STE

In STE, the formal model of a circuit is given by a next-state function that maps lattice states to lattice states. If we denote the next-state function by `Y` then

```
Y: (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` is constructed incrementally and piecemeal by ternary symbolic simulation of an HDL or a netlist source for the circuit. To run a typical example using our STE formalization requires one to define the model `Y` completely.[2]

[2] We will show in Section 5, how we do this for a specific circuit.

One crucial property the next-state function needs to satisfy 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`) on `s` is also less than or equal to ($\sqsubseteq$) the state reached by applying `Y` to `s'`.

Since sequences return lattice values for each node at a given time point, a sequence encodes a set of behaviours that a circuit can exhibit. The next-state function or the lattice model of the circuit provides the meaning of circuit behaviour. 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 behaviours that the sequence encodes is a subset of the behaviours that the circuit can actually exhibit. Below we show the formalization in HOL.

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

3.3 Syntax of STE

In STE, the basic syntactic entity used in specification 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 [16, 17, 18] of all the operators `Is_0`, `Is_1`, `AND`, `WHEN` and `NEXT`. However we have chosen to represent the guard [5, 6] shallowly by actually using the HOL type `bool` to represent the guard.[3] This is to allow us to inherit the theory of booleans (`bool`) in HOL. The HolBdd [20] package is interfaced to the type `bool` in HOL so 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).

3.4 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.

[3]Angelo *et al.* in [18] and Boulton *et al.* in [17] present interesting case studies of different kinds of embeddings of hardware description languages.

```
(* 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)))
```

3.5 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.

```
(* Datatype of Assertions *)

val _ = Hol_datatype `Assertion = ==>> of TF => TF`;
val _ = set_fixity "==>>" (Infixl 470);
```

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

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

Seger and Bryant in [1] proposed an algorithm for STE model checking. 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 up to the depth of `Cons`.

We have defined the functions to calculate the defining sequence (`DefSeq`) and the defining trajectory (`DefTraj`) in HOL. A defining sequence for a trajectory formula returns the lattice value at a given time point for a given node. It is the weakest sequencec that satisfies a given formula. There exists a defining sequence for each formula.

```
(* Defining Sequence of a trajectory formula *)

⊢ (DefSeq (Is_0 m) = λt n. (if ((n = m) ∧ (t = 0))
                              then Zero else
∧ (DefSeq (Is_1 m) = λt n. (if ((n = m) ∧ (t = 0))
                              then One else X
∧ (DefSeq (tf1 AND tf2) = λt n. (lub (DefSeq tf1 t n)
                                     (DefSeq tf2 t n)))
∧ (DefSeq (tf WHEN (P:bool)) = λt n.
                         (P ==> first(DefSeq tf t n),
                                P ==> second(DefSeq tf t n)))
∧ (DefSeq (NEXT tf) = λt n. (if ~(t = 0) then
                              (DefSeq tf (t-1) n)
                                 else X))`;
```

Defining trajectory behaves like a defining sequence if the time point for which the value is requested is zero. In other words Defining trajectory for a trajectory formula at time point 0 is just the defining sequence, and for all higher time points, defining trajectory includes the information arising out of circuit model. For a given circuit model, the defining trajectory for a given formula at a time point higher than zero, expresses the constrained state the circuit is going to take at that time point given that it will be driven into that state by its circuit model (which operates on the previous state), and the values that have been asserted in the trajectory formula at that time point which is expressed by the defining sequence at that time point.

```
(* Defining Trajectory *)

⊢ (DefTraj tf Model 0 node = DefSeq tf 0 node)
∧ (DefTraj tf Model (SUC t) node  = lub (DefSeq tf (SUC t) node)
                               (Model (DefTraj tf Model t) node))`;
```

We now state the STE implementation (`STE_Impl`) algorithm [1], which takes an assertion and a circuit model `Y` and computes a symbolic constraint[4] (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 are infinite. The depth of the segment is computed from the depth of the consequent in the assertion. The function `Depth` calculates the number of `NEXT` operators in a trajectory formula. It is defined inductively as

[4]It is also known as a residual

```
(* Depth of a trajectory formula *)

⊢ (Depth (Is_0 n) = 0)
∧ (Depth (Is_1 n) = 0)
∧ (Depth (tf1 AND tf2) = MAX (Depth tf1)(Depth tf2)))
∧ (Depth (tf WHEN P) = Depth tf)
∧ (Depth (NEXT tf) = 1 + (Depth tf))`;
```

```
 (* STE Implementation *)
  ⊢ ∀Ant Cons Y.
           STE_Impl (Ant ==>> Cons) Y =
               ∀t. t <= Depth Cons
                    ==> ∀n. MEMBER n (Nodes Ant Append Nodes Cons)
                    ==> leq (DefSeq Cons t n) (DefTraj Ant Y t n)
```

The function `Nodes`, calculates the list of nodes in a given formula, and the function `MEMBER` 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 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` if and only if the STE implementation guarantees that the trajectory assertion is valid for the model `Y`.

```
  (* Theorem 1: Correctness of STE algorithm *)
  ⊢ ∀Ant Cons Y.
        Monotonic Y ==>
         SAT_CKT (Ant ==>> Cons) Y
                     = STE_Impl (Ant ==>> Cons) Y
```

`Theorem 1` above is a succint representation for `AuxTheorem1`.

```
(* AuxTheorem 1 *)
  ⊢ ∀Ant Cons Y.
         Monotonic Y ==>
          (SAT_CKT (Ant ==>> Cons) Y =
          ∀t.
            t <= Depth Cons ==>
            ∀n.
              MEMBER n (Nodes Ant Append Nodes Cons) ==>
              leq (DefSeq Cons t n) (DefTraj Ant Y t n)) : thm
```

where we have unfolded the definition of `STE_Impl`. If we prove `AuxTheorem1` then the proof of `Theorem 1` is trivial, and can be obtained by a single call to `PROVE_TAC` — the first order decision procedure in HOL. This theorem gives us the equivalence between the concept of validity of an STE assertion and the mechanical procedure of the STE model checking algorithm. It therefore enables us to use the STE model checking procedure to practically model check an STE assertion against a given circuit model and infer that the STE assertion is valid against the same circuit model.

Getting the proof of `AuxTheorem1` involves doing two proofs, one is the if-part and the other the only-if part. The reader is referred to the proof script that is available online for the details. The idea behind the proof is to actually prove a theorem that doesn't limit the time up to the depth of the consequent. We therefore prove the following theorem (`SAT_CKT_IFF_STE`) first.

```
(* SAT_CKT_IFF_STE *)

  ⊢ ∀Ant Cons Y.
        Monotonic Y ==>
        (SAT_CKT (Ant ==>> Cons) Y =
         ∀t. leq_state (DefSeq Cons t) (DefTraj Ant Y t)) : thm
```

The above theorem states the property that a given circuit model that is monotonic, satisfies an STE assertion if and only if for all time points, it is the case that the defining sequence of the consequent returns a lattice value which is less than or equal to the lattice value obtained by calculating the defining trajectory for the antecedent with respect to the given circuit model.

Clearly if we can prove the above theorem, it will follow directly that for all time points less than the depth of the consequent in the STE assertion, the property is true - which is exactly what is expressed in `AuxTheorem1`. Together with the theorem `SAT_CKT_IFF_STE` and the facts[5] that the defining sequence of of a trajectory formula at all time points greater than the depth of the formula, is simply X; and all those nodes that are not named in antecdent or consequent take on a value X, one can obtain `AuxTheorem1`. The details are worked out in the proof script available online.

4 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;

[5] These are actually lemmas that we have proved

- relating correctness results in the STE world to correctness results in the Boolean world.

We shall discuss these issues in subsequent sections.

4.1 Semantics of Trajectory Formulas in Boolean Logic

States in the Boolean world are functions from the set of $\mathcal{N}$ to the set of Boolean values. We refer to the states in the Boolean world as *Boolean states*. We denote the Boolean set in HOL by `bool`. Using the type `string` to denote the set of nodes $\mathcal{N}$ the Boolean state will have the typing

```
sb: 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
```

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.

4.2 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))
```

4.3 Relational Circuit Model

A circuit in the Boolean world, is modeled by a next-state relation, which for a given circuit gives a relation between present and next Boolean state. 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: (string→bool)→(string→bool)→bool`

4.4 Boolean Sequence in the Language of the Circuit

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

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

4.5 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` and `Yb`) describe the same behaviour.

Intuitively, for a given circuit, with a relational model `Yb` and the lattice model `Y`, 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`, then the lattice model `Y` 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 Yb.
       Okay (Y,Yb) =
       ∀s_b s_b'.
         Yb s_b s_b' ==>
         leq_state (Y (extended_drop_state s_b))
                   (extended_drop_state s_b')
```

4.6 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 Yb.
       Okay (Y, Yb) ==>
           ∀sigma_b. in_BOOL_lang sigma_b Yb ==>
                     in_STE_lang (extended_drop_seq sigma_b) Y
```

The lemma states a fact that whenever the two circuit models `Y` and `Yb` 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 [6] is incorrect. The lemma as stated in [6], says that whenever two circuits satisfy the property `Okay` (Axiom 2 in [6]), 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 [6]), which is stated as an implication. Just by using the implication in Axiom 2, we cannot prove the equivalence property of the lemma [6].

Our claim here is that either we state both Axiom 2 and Lemma 1 (in [6]) 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 [6]) as an implication. Interestingly, in the paper [5] 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` and the Boolean model `Yb`, if `Y` and `Yb` 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`, for all time points `t` greater than zero, if the antecedent of the trajectory assertion (`Ant`) is satisfied by the t^{th} suffix of the Boolean valued sequence, then the consequent of the trajectory assertion `Cons` is also satisfied by the t^{th} suffix of the Boolean valued sequence.

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

⊢ ∀Ant Cons Y Yb.
       Okay (Y,Yb) ==>
       SAT_CKT (Ant ==>> Cons) Y ==>
       ∀sigma_b.
         in_BOOL_lang sigma_b Yb ==>
         ∀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.

5 Executing STE in HOL

The tool which allows us to execute STE model checking to verify an asserion, and infer a corresponding theorem in HOL is a theorem which we call `BridgeTheorem`. This theorem acts as a bridge between STE model checking and the notion of correctness in the Boolean world. Once we state the theorem we will outline the proof sketch and explain how we use it for executing STE model checking on an example circuit, and get a theorem in HOL which actually is a theorem we intuitively expect to hold if we do not do STE model checking anyway [5, 6].

[6] We have taken the example from [5, 6]

```
(* BridgeTheorem *)

⊢ ∀Ant Cons Y Yb.
       Okay (Y, Yb) ==>
           Monotonic Y ==>
               ((∀t.t <= Depth Cons ==>
                      ∀n.
                          MEMBER n (Nodes Ant Append Nodes Cons) ==>
                             leq (DefSeq Cons t n) (DefTraj Ant Y t n))
           ==>
               (∀sigma_b.
                       in_BOOL_lang sigma_b Yb ==>
                       ∀t.
                          SAT_BOOL Ant (Suffix_b t sigma_b) ==>
                          SAT_BOOL Cons (Suffix_b t sigma_b)))
```

`BridgeTheorem` states that if the lattice model and the Boolean model for a given circuit are equivalent *i.e.* satisfy the predicate `Okay`, and if the lattice model `Y` is monotonic then we get the correctness theorem about an STE assertion in the Boolean logic of HOL whenever the STE model checking run guarantees that in the three valued logic of STE.

The idea of the proof for the `BridgeTheorem` is rooted in `AuxTheorem 1` (Section 3.5) and `Theorem 2` (Section 4.6). Let us briefly review the intuition behind these theorems. `AuxTheorem 1` states the equivalence between the theoretical notion of validity of an STE assertion (`SAT_CKT`) and the practical STE model checking technique (`STE_Impl`). `Theorem 2` tells us that if a given STE assertion is satisfied by a given lattice circuit model, then it is appropriate to deduce a theorem in HOL, stating that the same STE assertion is satisfiable by a Boolean model for the same circuit. `Theorem 2` therefore provides us the link between the correctness result in STE (`SAT_CKT`) to the notion of correctness in the Boolean world (`SAT_BOOL`). What we have in the `BridgeTheorem` above is a link between STE model checking and the correctness in the Boolean world.

The proof of `BridgeTheorem` relies on unfolding the definition of `SAT_CKT` in `AuxTheorem 1` and then using the resulting theorem and `Theorem 2` to resolve the goal. For details refer to the proof script online.

Once we have proved the `BridgeTheorem` we can now use it for deriving theorems in HOL, for specific circuits and STE assertions.

We have written functions in ML, that allow the user to enter information about a specific circuit, and both its lattice and Boolean models and write STE assertions that need to be verified against the supplied circuit models. The top level function that interacts with the user is called `STE_TO_BOOL`. This function accepts the user supplied values and eventually generates a specific theorem in HOL. If the residual from the STE model checking run evaluates to true, then we just get the theorem in HOL that we intuitively expect to have when we do not perform any STE model checking, and restrict ourselves to purely theorem proving approach. If the STE model checking run comes up with a residual then the residual becomes a part of the resulting theorem in HOL.

Our implementation has two basic components. One is a *firmware* part and the other is the *dynamic* part. The firmware part consists essentially of all the definitions that we have presented above, supporting lemmas and all the theorems that eventually lead us to the `BridgeTheorem`. This part is called firmware because the user never really needs to modify any of this, for executing STE model checking on different circuits. The dynamic component has a hardwired part, but dynamic component is the part where the user adds more information about circuit models, and writes STE assertions. The hardwired part of the dynamic component comprises those functions which we need for general simplification and reduction no matter what the given ciruit model is and what specific STE assertions have been written. The part that gets changed is the one where circuit specific reductions and simplifications need to be done for each given circuit, and the user needs to code specific ML functions for each circuit in question. These are the functions which determine for every given circuit that the given circuit models — lattice and Boolean represent the same circuit *i.e.* prove `Okay` and also ensure that the lattice model is monotonic.

Now we outline the execution strategy used by the toplevel function `STE_TO_BOOL`. This function accepts an STE assertion from the user, lattice and Boolean models (these are ML functions) for a given circuit, and a string identifier corresponding to the requested circuit. The string identifier is used for loading the circuit models for the specific circuit whose lattice and Boolean models have been supplied. Below we outline the tasks `STE_TO_BOOL` performs before it eventually generates the theorem in HOL.

- After loading the circuit library, the next task is to start STE model checking on the given circuit model. This is undertaken by a function that the user writes specifically for the requested circuit. This function takes as an argument the antecedent, and consequent of the STE assertion, lattice and Boolean models and a list of theorems that are used in simplification. The list of theorems consists of a base list which is the list of all hardwired definitions, lemmas and theorems we have proved once, and some definitions related to the specific circuit model.

  This function is responsible for invoking the routine for STE model checking (which we call `SPECL_AND_EXECUTE`), ensuring whether or not the lattice and Boolean models represent the same circuit (prove `Okay`), and then uses a bunch of hardwired functions for general simplification. The output of this function is a theorem which is closer to the theorem we want in HOL, but there few more simplifications need to be done in the last step.

- In the final few steps of simplification we ensure that the lattice circuit model is monotonic (prove `Monotonic`) and via some simplifying reductions obtain the desired theorem in HOL.

At the heart of the function `SPECL_AND_EXECUTE` is the `BridgeTheorem`. The function specializes the `BridgeTheorem` with the user supplied values for antecedent, consequent, lattice and Boolean models and initiates the execution of the specialized theorem. We coded special conversions and functions which do the execution. Conversion [21, 22] is a special kind of function in HOL, which does term rewriting. Conversions have the

HOL type `term` → `thm`. A conversion takes a term and if that term is equal to some other term then a new theorem is created stating the equality of the two terms. We can then invoke standard simplifiers in HOL with the new theorem and perform further simplification. Although HOL has some standard conversions, we have written our own conversions for the STE model checking routine. Another function in HOL which we used and is extremely useful in theorem derivations is `CONV_RULE` [22]. It takes a theorem (also known as a rule in HOL) and uses conversions to rewrite the terms appearing in the supplied theorem to derive a new theorem. Conversions together with `CONV_RULE` give us significant power to write new set of simplifications which help us in deriving new theorems.

Now we present an example to illustrate our implementation. The simple example is of a two input unit-delay Nand gate whose one of the outputs is tied to the input (see Figure 1)

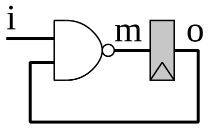

Figure 1: Unit Delay Nand Gate

We define the lattice and the Boolean models for such a circuit. Below is an example we wrote in HOL.

```
(* 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 the input node **i** and the output node **o** respectively, then after one unit of time we can expect to observe the value $\neg$(v1 $\wedge$ v2) at the output **o**. In fact 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 now show what happens when we call the function `STE_TO_BOOL`. The function takes `Ant`, `Cons`, 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 trajectory 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`. Other examples where the residual does not evaluate to true, are available online with the proof scripts.

6 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 [5] on linking trajectory evaluation to higher-order logic. We also extended their idea by writing functions that implement the core STE algorithm known from [1] 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 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.* [14]. 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 and therefore it appears to us that our approach may not turn out to be as good as [14]. However one of the immediate goals we have set out for ourselves is to investigate how to integrate BDDs in our implementation. We are investigating using the HolBdd package [20] to see how well it suits our design goals.

We are investigating the use of abstraction and reduction techniques with STE, and we look forward to combine these in our implmentation. 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 [6]. Although the discrepancy isn't a major bug in the report, we believe our effort in uncovering it is worthwhile.

7 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] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] S. Agerholm and H. Skjϕdt, "Automating a model checker for recursive modal assertions in hol," Tech. Rep. DAIMI IR-92, Computer Science Department, Aarhus University, 1990.

[8] 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.

[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] Thomas F. Melham, "PROSPER - an investigation into software architecture for embedded proof engines," in *Frontiers of Combining Systems*, 2002, pp. 193–206.

[11] Tomas E. Uribe, "Combinations of model checking and theorem proving," in *Frontiers of Combining Systems*, 2000, pp. 151–170.

[12] J. Joyce and C.-J. Seger, "Linking BDD based symbolic evaluation to interactive theorem proving," in *ACM/IEEE Design Automation Conference*, June 1993.

[13] Carl-Johan Seger, "VOSS - A formal hardware verification system user's guide," Tech. Rep. TR-93-45, Dept. of Computing Science, Univeristy of British Columbia, Vancouver, Canada, 1993.

[14] 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.

[15] Randal E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Mike Gordon, *HolBddLib, Version 2*, Computer Laboratory, University of Cambridge, March 2002.

[21] Lawrence C. Paulson, “A higher-order implementation of rewriting,” *Science of Computer Programming*, vol. 3, pp. 119–149, 1983.

[22] *The HOL System Description, Kananaskis 1*, June 2002.