Property-Specific Testbench Generation for Guided Simulation

Aarti Gupta, Albert E. Casavant, and Pranav Ashar NEC CCRL, Princeton, NJ, USA {agupta, aec, ashar}@nec-lab.com X. G. (Sean) Liu University of California, Santa Barbara, CA, USA xgliu@ece.ucsb.edu

Akira Mukaiyama, and Kazutoshi Wakabayashi NEC CRL, Kawasaki, Japan {mukai, wakaba}@ccm.cl.nec.co.jp

Abstract

Simulation continues to be the primary technique for functional validation of designs. It is important that simulation vectors be effective in targeting the types of bugs designers expect to find rather than some generic coverage metrics. The overall focus of our work is to generate a property-specific testbench for guided simulation, that is targeted either at proving the correctness of a property or at finding a bug. This is facilitated by generation of a property-specific model, called a "Witness Graph", which captures interesting paths in the design. Starting from an initial abstract model of the design, symbolic model checking, pruning, and refinement steps are applied in an iterative manner, until either a conclusive result is obtained or computing resources are exhausted. This paper describes the theoretical underpinnings of generating and using a Witness Graph for CTL correctness properties, practical issues related to the generation of a testbench, and experiences with an industrial example. We have been able to demonstrate on a real in-house design that such an approach can lead to significant reduction in the time required to analyze the design for a CTL property and find a witness.

1. Introduction

Functional validation is one of the key problems hindering successful design of large and complex hardware or hardwaresoftware combinations. The technology for formal verification, in which the correctness criteria (properties) are specified formally, and a tool exhaustively and automatically exercises the functionality of the design to prove the properties, has improved significantly in the recent past. In particular, the use of Computation Tree Logic (CTL) as a way of specifying properties and model checking as a method of proving the properties has shown the potential to become accepted in industry [3]. Unfortunately, formal verification technology, including CTL-based model checking, is not robust enough yet to be relied upon as the sole validation technology. The primary hurdle is the inability of model checking tools to handle large state spaces in current designs using reasonable amounts of resources. On the other hand, simulation is inherently slow, requiring the simulation of billions of vectors for complex hardware. Furthermore, the coverage of design functionality provided by these vectors remains largely unknown.

A practical alternative is semi-formal verification, where the specification of correctness criteria is done formally, as in model checking, but checking is done using simulation, which is *guided* by directed vector sequences derived from knowledge of

the design and/or the property being checked. A typical validation framework consists of a language specifying correctness criteria and vector generation constraints, where the constraints are derived manually according to the property of interest, e.g. [14]. As shown in Figure 1, the focus of our work, called *intelligent testbench generation*, is to *automatically* determine the appropriate vector generation constraints, based on analysis of *both* the design and the property being checked.

We use CTL for formal specification of correctness properties; our ideas can be applied similarly to other forms of specifications such as Linear Temporal Logic (LTL), ω -regular automata etc. Furthermore, the properties for which targeted vector generation is performed could either be provided manually by the user, or be derived automatically from the HDL design description, based on generic notions of correctness, e.g. through use of assertions.

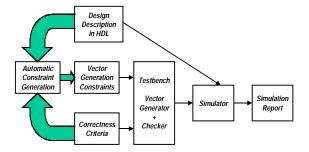


Figure 1: Intelligent Testbench Generation

The testbench integrates a test vector generator, and a checker module (monitor) that checks for violation or satisfaction of the property. The goal for the vector generator is to increase the likelihood that either a witness to the property or a counterexample is generated. This task is facilitated by embedding in constraints that are derived from a *Witness Graph*. Intuitively, a Witness Graph represents a collection of states/transitions/paths in the design that are useful for enumerating witnesses or counter-examples for the required property. In this paper, we describe our methods for generating a Witness Graph, and its use in searching for witnesses or counter-examples during simulation.

1.1 Related Work

Our work is broadly related to other efforts that have combined formal verification techniques with simulation for functional validation. In particular, we also extract an abstract model of the design for the purpose of functional validation [7]. However, we focus on correctness properties, rather than simple coverage measures such as state/transition/pair-arc coverage, which may not always correlate with error coverage. It should also be noted that we do not claim to have solved the problem of concretizing abstract simulation vectors, which is the primary hindrance in the practical application of such techniques. We circumvent the concretization problem by focusing not on generation of simulation vectors, but on automatic generation of the testbench itself. The testbench is organized as a backtracking search procedure, where embedded constraints on transitions/paths between abstract states can be used to filter (pseudo-) randomly generated inputs during simulation. Naturally, the effectiveness of our technique depends critically on the practical efficiency of this search. Our approach is to use a combination of known methods including static analysis of the abstract model, e.g. [9, 16], hints from the user, trace data from previous simulations runs etc. to improve the search.

Another line of work is based on using symbolic methods within simulation to make it more effective [5, 17]. However, this has so far been targeted at obtaining better coverage for reachability and invariant checking, rather than handling more general correctness properties. There have been many efforts based on constraint solving for testbench generation, e.g. [8]. These can potentially be combined with our techniques based on model checking to derive and/or solve embedded constraints in the testbench. Finally, the details of our analysis technique are similar to other efforts in the area of abstraction, approximate model checking, and refinement [4, 10, 11, 12, 13]. A discussion of these is deferred to Section 2, where our techniques are described in detail.

2. Witness Graph Generation

Given a set of atomic propositions *A*, the set of CTL formulas is recursively defined as follows [3]:

$$\begin{array}{l} \text{CTL formulas} = p \in \ \mathsf{A} \ | \ !f \ | \ f^*g \ | \ f+g \ | \\ \text{EX } f \ | \ \text{EF } f \ | \ \text{EG } f \ | \ \text{E} (f \ \text{U} \ g) \ | \\ \text{AX } f \ | \ \text{AF } f \ | \ \text{AG } f \ | \ \text{A} (f \ \text{U} \ g), \end{array}$$

where p denotes an atomic proposition, f and g are CTL formulas, and !/*/+ denote the standard Boolean negation/conjunction/disjunction operators, respectively. The CTL modalities consist of a path quantifier A (all paths) or E (exists a path), followed by a temporal operator – X (next time), F (eventually), G (globally), U (until). The nesting of these modalities can express many correctness properties such as safety, liveness, precedence etc. For example, a formula AG f expresses that f is true globally in all states on all paths, i.e. f is an invariant.

The intended purpose of a Witness Graph is to serve as a property-specific abstract model of the design, which captures witnesses or counter-examples for the property. Note that for full CTL, a witness or counter-example need not be a simple path, but may be a general graph. For practical reasons, we focus on generation of a small Witness Graph that is also *complete*, i.e. it should include *all* witnesses or counter-examples. We follow an iterative flow for generation of a Witness Graph, as shown within the dashed box in Figure 2.

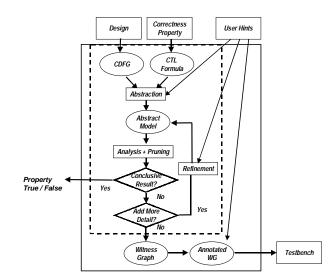


Figure 2: Flow for Witness Graph Generation

Starting from a given design and property, we first obtain an abstract model. Next, we perform analysis by model checking and pruning, and refine the model to perform analysis again. The iterative process is repeated until either a conclusive result is obtained, or resource limitations are reached. In the latter case, the current abstract model constitutes the Witness Graph. It can be represented in any of the standard FSM forms, including a control data flow graph (CDFG), an RTL description, or an implicit symbolic representation using BDDs [2]. The details of this flow are described in the rest of this section. As also shown in Figure 2, the Witness Graph is subsequently annotated with priorities etc., which is then used for automatic generation of the testbench – this is described in Section 3.

2.1 Initial Abstract Model

First, we use the cone-of-influence abstraction [1, 10], whereby any part of the design that does not affect the property is removed. Since the number of control states in a CDFG design representation is typically small, we perform explicit traversal on the control states to identify irrelevant datapath operations. This provides better abstraction than a purely syntactic analysis on the next state logic of a standard RTL description. Next, we identify datapath variables that do not directly appear as atomic propositions in the CTL property, and are therefore potentially suitable for abstraction as pseudo-primary inputs. Again, we use explicit traversal over the control states to identify datapath dependencies for ranking these candidates and abstracting them. The resulting model constitutes an upper bound approximation of the underlying Kripke structure [10, 11, 12, 13].

Example: As a running example for our techniques, consider the CDFG design description shown in Figure 3. It consists of 9 control states, labeled ST0 through ST8, with initial state ST0. The variables i, j, A, B, C, and F are primary inputs, and the rest are datapath variables. The light bordered boxes indicate the datapath operations performed in each control state, while the labels on the edges between control states identify the conditions under which those transitions take place. Note that while the number of control states is small, the total state space including the datapath is actually large. Suppose the correctness property is EF (M \geq = 6), i.e. we want to check the existence of a path starting from ST0 on which eventually some state satisfies M >= 6. We use cone-of-influence analysis to determine that state ST3 does not contain any relevant datapath operations. Next, since M is the only datapath variable referred to in the atomic proposition, we include M and its immediate dependency H as state variables. All other data variables are abstracted away as pseudo-primary inputs.

2.2 Analysis of the Abstract Model

The next step is to perform analysis on the abstract model to identify states that contribute to any witness/counter-example for the property. For formulas starting with an E-type operator, we look for all witnesses; while for formulas starting with an A-type operator, we look for all counter-examples. For rest of this discussion, assume that we are interested in finding witnesses – the same discussion holds for finding counter-examples.

The pseudo-code for our algorithm, called mc_for_sim (model checking for simulation), is shown in Figure 4. Its inputs are an abstract model m, which is an upper-bound approximation of the concrete design d, and a CTL formula f in negation normal form, i.e. where all negations appear only at the atomic level.

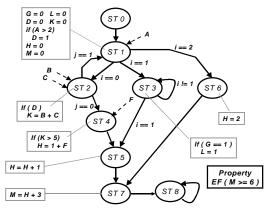


Figure 3: Example CDFG and Property

The main idea is to use model checking over m to pre-compute a set of abstract states which are likely to constitute witnesses, and to use this set for guidance during simulation over d, in order to demonstrate a concrete witness. In particular, we target *over*-approximate sets of satisfying states during model checking, so that we can search through an over-approximate set of witnesses during simulation.

The mc_for_sim algorithm works recursively in the standard bottom-up manner over the CTL formula f (represented in the form of a parse tree, where left / right subformulas of f are denoted leftChild(f)/rightChild(f), respectively). It associates sets of abstract states called upper/negativewith subformulas of f (and their negations when needed).

```
mc_for_sim(model m, ctlFormula f) {
  ctlFormula f1, f2, negf;
  states upper,upper1,upper2=NULL,negative=NULL;
  // handle subformulas recursively
  if (f1 = leftChild(f)) {
    mc_for_sim(m,f1);
    upper1 = get_upper(f1);
  }
}
```

```
if (f2 = rightChild(f)) {
  mc_for_sim(m,f2));
  upper2 = get_upper(f2);
// case analysis on operator at this level
switch(type(f)) {
case TRUE: upper = ALL; break;
case FALSE: upper = NULL; break;
case ATOMIC: upper = mc_atomic(m,f); break;
case NOT: upper = complement(upper1); break;
case AND: upper = and(upper1,upper2); break;
case OR: upper = or(upper1,upper2); break;
case EX: case EF: case EU: case EG:
  upper = mc_etype(upper1,upper2); break;
default: // A-type operators left
  switch(type(f)) {
  case AX: upper = mc_ex(upper1); break;
  case AF: upper = mc_ef(upper1); break;
  case AU: upper = mc_eu(upper1,upper2); break;
  case AG: upper = mc_eg(upper1); break;
  // compute negative sets also
  negf = negate(f);
  mc_for_sim(m,negf);
  negative = and(upper,get_upper(negf)); break;
// associate the sets with f
associate(f, upper, negative);
```

Figure 4: Pseudo-code for mc_for_sim Algorithm

Theorem 1: The set of abstract states upper associated with each subformula by the mc_for_sim algorithm corresponds to an over-approximation of the set of concrete states that satisfy the subformula.

Proof: The proof is by induction on the structure of the formula. Note that atomic propositions (and constants) are computed exactly, in the standard manner, providing the basis of the induction. Furthermore, since only atomic-level negations are allowed in a negation normal form, they too are computed exactly. Since other Boolean operators are monotonic, they preserve over-approximations of the subformulas. For subformulas beginning with an E-type operator (EX, EF, EU, EG), standard model checking over *m* (function *mc_etype*) ensures that the result is an over-approximation over d, since mhas more paths than d. However, for subformulas beginning with an A-type operator (AX, AF, AU, AG), the situation is somewhat different. Since m may have many false paths with respect to d, standard model checking over m may result in an under-approximation over d. Therefore, we compute upper by considering the corresponding E-type operator, which is guaranteed to result in an over-approximation.

The over-approximation for the A-type operators is rather coarse. To mitigate this effect, we also compute a set of abstract states called *negative* as shown in Figure 4. It corresponds to the intersection of set *upper* with a set which is recursively computed for the *negation* of the A-type subformula. Though not shown in the pseudo-code, an actual implementation of the above algorithm keeps track of the visited nodes in the parse trees of various CTL subformulas, such that each node is explored at most once. Therefore, its overall complexity is the same as that of standard symbolic model checking.

2.2.1 Conclusive Proof Due to Model Checking

It is possible that model checking on m itself provides a conclusive result for d in some cases. Pseudo-code for performing this check is shown in Figure 5, where the mc_for_sim algorithm is used to compute the sets upper / negative for the top-level formula.

Figure 5: Pseudo-code for check_mc Algorithm

Theorem 2: The result computed by the check_mc algorithm is correct in all cases.

Proof: From Theorem 1, set *upper* corresponds to an overapproximate set of satisfying states. Therefore, if the initial state does not belong to this set, clearly the property is false. Now, assume that the initial state does belong to set *upper*. Recall that for an A-type operator, we also compute the set *negative*. If the initial state does not belong to set *negative*, then there does not exist any path in *m* starting from the initial state that shows negation of the property. Therefore, it is guaranteed that no such concrete path exists in *d*, i.e. the property is true. In all other cases, the result from model checking is inconclusive. \blacksquare

2.2.2 Partial Proof Due to Model Checking

When the result due to model checking is inconclusive, we fall back upon simulation for generating witnesses/counter-examples for the property. For full CTL, we need to handle the alternation between E and A quantifiers. In general, handling of "all" paths is natural for model checking, but is unsuitable for simulation. The purpose of computing negative sets for A-type subformulas is to avoid a proof by simulation where possible. Note that an abstract state *s* which belongs to *upper*, but not to negative, is a very desirable state to target as a witness for the A-type subformula. This is because the proof of the A-type subformula is complete for state *s* due to model checking itself (as described in the proof of Theorem 2). Therefore, as soon as state s is reached during simulation, there is no further proof obligation. On the other hand, if a state *t* belongs to *negative* also, our task during simulation is to check whether an abstract counter-example for the A-type subformula actually corresponds to a concrete path. If a concrete counter-example is found, state t is not a true witness state, and can be eliminated from further consideration. This observation is used in the witness generation algorithm described in Section 3.

2.2.3 Related Work

Our abstraction technique and *mc_for_sim* algorithm are similar to other works in the area of abstraction and approximate

model checking [4, 11, 12, 13]. Like many of these efforts, we also use an "existential" abstraction which preserves the atomic propositions, and also allows us to compute overapproximations of satisfying states (sets upper). Furthermore, our computation of the negative sets for the A-type subformulas is similar to computing under-approximations. (In principle, we can compute these for all subformulas.) However, our purpose for computing these approximations is not only to use these sets for conservative verification for CTL (or its existential/universal fragments), or even for iterative refinement. Ultimately, these sets are used to provide guidance during simulation for designs where it may not be possible to perform any symbolic analysis at all. Therefore, unlike existing techniques, our *mc_for_sim* algorithm specifically avoids employing existential/universal quantification over the state space of concrete variables. Instead, we use much coarser approximations - the E-type operators in place of the A-type operators. Indeed, it would be appropriate to use any known technique for obtaining the tightest approximations. Our additional contribution is also in showing how these sets can be used to demonstrate concrete witnesses in the context of simulation.

2.3 Pruning of the Abstract Model

The next step is to prune the abstract model by removing states that do not contribute to any witness or counter-example. We first mark the required states, and remove any states that are left unmarked, by replacing them with a special "sink" state. (In order to allow repeated use of model checking on the pruned model, every transition out of "sink" state leads back to itself, and all atomic propositions in the CTL property are assumed to be false in the "sink" state.) The pseudo-code for our state marking algorithm is shown in Figure 6.

```
mark_witness_top(model m, ctlFormula f)
 reachable = compute_reachable(m,initState(m));
 switch(type(f)){
  case AX: case AF: case AU: case AG:
    witness top= and(get negative(f),reachable);
    break;
  default:
    witness_top= and(get_upper(f),reachable);
 mark_states(witness_top);
 mark_witness_rec(m, f, witness_top);
mark_witness rec(model m, ctlFormula f, states
careSet)
states witness, negWitness, subWitness;
// associate witness set for f
witness = and(get_upper(f), careSet);
associate_witness(f,witness);
// recursive calls with modified careSets
switch(type(f)) {
case TRUE:case FALSE:case ATOMIC:case NOT:
  break;
case AND: case OR:
case EF: case EU: case EG:
   mark_witness _rec(m,leftChild(f),witness);
   if (rightChild(f) != NULL)
     mark_witness_rec(m,rightChild(f),witness);
   break;
```

```
case EX:
   subWitness = compute_image(m, witness);
   //mark additional states
   mark_states(subWitness);
   mark_witness_rec(m,leftChild(f),subWitness);
   break;
case AX: case AF: case AU: case AG:
   negWitness = and(get_negative(f),careSet);
   associate_neg_witness(f,negWitness);
   mark_witness_rec(m,negate(f),negWitness);
   break;
}
```

Figure 6: Pseudo-code for State Marking Algorithm

Theorem 3: The proposed state marking algorithm is conservative, i.e. it will not miss any state needed to demonstrate a witness or counter-example for the property of interest.

Proof: Note that we are interested in states that not only *start* a witness/counter-example, but demonstrate it *fully*. The crucial observation is that for any CTL formula f, except of type EX/AX, such states also satisfy f. For atomic propositions and Boolean operators, this is trivially true since there are no paths to consider. For type EF/EU/EG, the witnesses are paths where each state satisfies f. Similarly, for type AF/AU/AG, counter-examples are paths where each state satisfies *l*f. Indeed, it is only for EX/AX, that we need to mark additional states, i.e. those that satisfy the subformula of f. Therefore, it would be enough to mark satisfying states once at the top, followed by additional marking only in the EX/AX case.

In our method, sets upper and negative correspond to over-approximations of concrete satisfying states. Furthermore, for A-type subformulas, we need to focus only on states that belong to both sets, in order to search for a concrete counterexample during simulation. Recall that for states that belong to upper but not to negative, the proof is complete due to model checking itself. Therefore, our marking algorithm uses the sets upper/negative to associate sets called witness/neg_witness with each required CTL subformula. As an additional optimization, since the former sets are computed bottom-up, we use the latter sets top-down, as care-sets for the subformulas. At the topmost level, the care-set consists of the set of states reachable from the initial state. Note that the special handling of EX-type subformulas requires an extra image computation to exploit the care-set. Since the sets upper/negative are over-approximations, and the care-sets preserve reachability from the initial state, our state marking algorithm is conservative.

Returning to our example, for the abstract model of Figure 4, the states ST3 and ST6 remain unmarked after performing the above analysis. This is because there is no path through these states that can demonstrate a witness for the property EF (M>=6). Therefore they are pruned, and replaced by the special "sink" state.

2.4 Refinement of the Abstract Model

The amount of detail that can be allowed in the abstract model depends on the level of complexity that can be handled by the model checker. However, once pruning is done resulting in a

smaller model, it may be possible to refine the model and perform the analysis again. Recall that our initial abstract model was obtained by abstracting away many of the datapath variables as pseudo-primary inputs. We perform refinement by selectively bringing back some of these datapath variables into the state space. Note that pruning reduces the size of the model, while refinement increases it. The iterative pruning and refinement allows us to model much more of the state space in the final Witness Graph than would be possible otherwise.

For our example, we may choose to add datapath variables D and K as state, after which model checking and pruning are performed again. If we don't wish to add any more datapath state at this point, we obtain the final Witness Graph as shown in Figure 7.

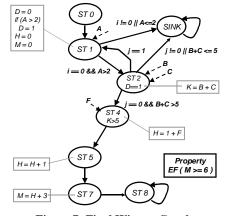


Figure 7: Final Witness Graph

Again, our techniques for iterative refinement are similar to those used by other researchers, where lack of a conclusive result from the abstract model [11, 12], or some counterexamples on the abstract model [4, 10] are used to guide further refinement. In contrast, we focus on all witnesses/counterexamples during model checking. Furthermore, we use the associated sets for marking states in order to prune the abstract model before attempting further refinement. To the best of our knowledge, existing techniques do not perform such model pruning. This is largely because pruning of states does not necessarily lead to compact BDD representations used for symbolic manipulation. However, our goal is not only to obtain a conclusive result by model checking where possible, but also to reduce the gap between the abstraction levels of the final Witness Graph and the concrete design to be simulated. Since the final simulation is performed on explicit states, rather than symbolic sets, such pruning may be very useful in practice.

2.5 Witness Graph as a Coverage Metric

Apart from using a Witness Graph for generating a testbench, it can also be used as a coverage metric for evaluating the effectiveness of a given set of simulation vectors. Most available metrics are based either on code (line/branch/toggle) coverage of the design description, or on extraction of FSM models with the associated state/transition coverage [7]. In contrast, our metric is obtained by analysis of the design with respect to the given property. The better the coverage of a given set of simulation vectors over the states/transitions/paths of a complete Witness Graph, the more likely it is that simulation will succeed in proving/disproving the property. Note that a high coverage still does not guarantee correctness in the design – it only provides a metric to assess the quality of simulation. Recently, there has also been work on specification coverage metrics, which focus on how much of the design space is covered by multiple properties [6]. We can potentially use these techniques to extend our per-property analysis to coverage of overall correctness.

3. Testbench Design for Witness Generation

The Witness Graph is used to guide the testbench in searching for witnesses or counter-examples during simulation of the concrete design. The underlying skeleton of the testbench is a backtracking search algorithm, described in detail next.

3.1 Backtracking Search Algorithm

The pseudo-code for our search algorithm, called *search_witness*, is shown in Figure 8. It returns *SUCCESS* if it succeeds in finding a concrete witness starting from a given state *s*, in a given concrete design *d*, for a given CTL formula *f*; and *FAILURE* otherwise. It uses the *witness/neg_witness* sets which implicitly constitute the Witness Graph. In the pseudo-code, *abs(s)* denotes the abstract state corresponding to a concrete state *s*.

```
search witness(design d, ctlFormula f, state s)
ł
 states w. wl;
 int result, neg_result;
 w = get_witness(f);
 w1 = get_witness(leftChild(f));
 // case analysis on operator at this level
 switch(type(f)) {
 case TRUE: result= SUCCESS; break;
 case FALSE: result= FAILURE; break;
 case ATOMIC: result= satisfies(s,f); break;
 case NOT: result=satisfies(s,negate(f));break;
 case AND:
   result= search witness(d,leftChild(f),s);
   if (result==SUCCESS)
     result= search witness(d,rightChild(f),s);
   break;
 case OR:
   result= search_witness(d,leftChild(f),s);
   if (result==FAILURE)
      result= search_witness(d,rightChild(f),s);
   break;
 case EX:
    foreach state t, abs(t) \in w1, {
      if (exists_transition(s,t)){
       result=search_witness(d,leftChild(f),t);
       if (result==SUCCESS) break;}}
   break;
  case EF:
    foreach state t, abs(t) \in w1, {
      if (path = find_a_path(s,t)){
       result= search_witness(d,leftChild(f),t);
       if (result==SUCCESS) break; }}
   break;
  case EU:
    result= search_witness(d,rightChild(f),s);
    if (result==FAILURE) {
      mark(s,f);
```

```
result= search witness(d,leftChild(f),s);
    if (result==SUCCESS)
      foreach unmarked state t, abs(t) \in w {
        if (exists_transition(s,t)){
          result= search witness(d,f,t);
          if (result==SUCCESS) break;}}
  break;
case EG:
  result= search witness(d,leftChild(f),s);
  if (result==SUCCESS) {
    mark(s,f);
    if (exists_transition_to_marked(s,f))
      result= SUCCESS;
    else
      foreach unmarked state t, abs(t) \in w {
        if (exists_transition(s,t)){
          result= search witness(d,f,t);
          if (result==SUCCESS) break; }}}
 break;
case AX: case AF: case AU: case AG:
  if (abs(s) \notin get_neg_witness(f))
    result= SUCCESS;
  else {
   // generate counter-example for !f
    neg_result=search_witness(d,negate(f),s);
    result = (neg_result == SUCCESS) ?
      FAILURE : SUCCESS; }
}
return result;
```

Figure 8: Pseudo-code for search_witness Algorithm

Theorem 4: The search_witness algorithm is complete, i.e. it will find a concrete witness or counter-example if it exists.

Proof: Given the association of witness/neg_witness sets with various CTL subformulas, the handling of atomic propositions, Boolean operators, and the E-type temporal operators is according to their standard characterizations. The handling of the A-type operators reflects our earlier remarks – if abs(s) does not belong to set negative, the proof of the A-type subformula is complete due to model checking itself, and we can return with SUCCESS. Otherwise, we look for a counter-example for f starting from s. If such a counter-example is found, i.e. $neg_result=SUCCESS$, then we return FAILURE; and vice versa. Recall from the proof of Theorem 3 that the witness/neg_witness sets correspond to over-approximations of concrete witness states reachable from the initial state. Therefore, a search based on these sets is guaranteed to be complete.

3.2 Prioritizing Search for Witnesses

Though the *search_witness* algorithm is complete in principle, it is impractical to search through all possible concrete states in the *foreach* loops of the pseudo-code. Typically, the search would be limited by available resources such as space and time. In practice, any prior information about the existence of transitions/paths between two given concrete states can be used to prioritize the search. The designer may help in assigning priorities by providing hints, i.e. specifying particular control states or transitions as intermediate targets. Trace data from previous simulation runs which may identify "easy to reach" states can also be incorporated. Other schemes, e.g. [9, 16],

based on combination of distance to targets, transition probabilities, static analysis, hints etc. can also be used.

Basically, two questions must be answered before applying the next simulation vector:

- 1. What should be the next state?
- 2. What inputs are required to reach that state?

To choose the next state in our approach, the fixpoint approximations in the symbolic computation of a CTL subformula, called *onion rings*, are used to determine the shortest distance to target. The Witness Graph for each sub formula is now stored as a collection of BDDs, where each BDD represents the abstract state set with a certain distance tag for that subformula. These onion rings are restricted to the reachable state sets in the abstract model. In choosing the next abstract state, the testbench prioritizes states with lower distance tags.

Once the desired next abstract state is determined, the input must be chosen to enable such a transition from the current concrete state. Our approach works by generating a candidate input vector and simulating it on the entire design. If the abstract part of the new concrete state matches the desired next abstract state, this candidate is chosen, else a new candidate is attempted. A candidate input vector may be chosen randomly or by solving circuit constraints using BDD / SAT / LP techniques.

In our approach, candidate input vectors are generated using BDDs to solve circuit constraints in the abstract model. Since a symbolic analysis of the abstract model was performed during Witness Graph generation, the required BDDs are tractable. We use BDDs to combine into one step the choice of the next state and the input vector, by solving for an input vector that takes the abstract model from the current state to any state in the next desired onion ring. It is possible that no input vector can be obtained in this manner even after many candidates have been tried. In such a case, a new desired next state must be chosen. It is also possible that a transition to the desired onion ring with a smaller tag cannot be obtained after many next states have been attempted as targets. In this case, backtracking must be performed to return to an older state along the path.

3.3 Testbench and Simulation Setup

An outline of the simulation setup is shown in Figure 9. The conventional simulator in the setup could be based on Verilog, VHDL or a C-based HDL. The testbench code is generated automatically after completing symbolic analysis on the abstract model which generates the annotated Witness Graph. The Main Loop repeatedly queries the Vector Generator, which uses the Witness Graph and circuit constraint BDDs to identify the desired next states and input vector candidates to apply.

The entire testbench generation process is completely automatic and transparent to the user. The Main Loop is generated in the HDL native to the simulator while the Vector Generator is written in C. The two communicate through the PLI or other API provided by the simulator. The Witness Graph and circuit constraint BDDs are stored in compact binary file format to be read by the Vector Generator. We currently have implementations of this setup for a standard Verilog HDL simulator, and also for an in-house C-based-HDL simulator.

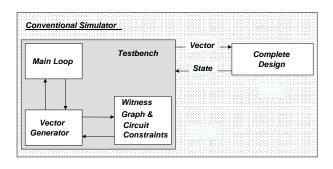


Figure 9: Simulation Setup

During simulation, the testbench outputs either a viewable VCD (Value Change Dump) file with the waveform corresponding to the generated witness, or a report that the witness could not be generated and corresponding partial waveforms up to the states that were reached.

4. Case Study: Memory Interface Design

4.1 Design Details

We have experimented with the proposed automatic propertyspecific testbench generation approach on an in-house memory interface design (*MIF*). The *MIF* design implements a complex arbitration scheme between multiple clients wishing to access the memory resource. The algorithm starts with fixed priorities for the clients, but dynamically adjusts them based on the state of the client, DMA requests, and interrupts. The *MIF* design is fairly complex, with a Verilog implementation consisting of 516 flip-flops, 1035 primary inputs, and 15989 literals. A reachability analysis of the entire *MIF* design appears to be infeasible with current BDD-based or any other technology.

As a result of the complex arbitration, it is very hard to manually, or through explicit enumeration of the state space, visualize the various scenarios under which a client may or may not be granted access. Arising out of this difficulty, the user highlighted to us a property he wished to check for this design in which he desired to determine if one of the clients, the CPU, would ever be required to wait for more than 5 cycles to be granted access to the memory. This behavior is described by the CTL property $EF(cpu_wait_counter > 5)$, where the signal $cpu_wait_counter$ counts the number of cycles that the CPU has to wait after making a request. Basically, we would like to find a witness input sequence in which the CPU has not been granted access within 5 cycles after requesting it.

4.2 Experimental Setup

Our prototype implementation is based on the publicly available symbolic model checker called VIS [1]. Since the Verilog description of the *MIF* design could not be directly read into VIS (due to limitations of its Verilog front-end), we started with a high-level description of the design written in a C-like HDL, which was used as input to an in-house high-level synthesis tool called Cyber [14]. An intermediate representation during this flow provides a CDFG representation of the design. This representation is used to automatically generate an FSM model (in VIS's *blif-mv* format), which is used for subsequent model

checking. We also automatically translated an extended CTL syntax which allows arithmetic operators in atomic predicates, to VIS's CTL syntax without arithmetic operators.

4.3 Results

Verification of the given correctness property with the VIS model checker was successful, but took 41 hours of CPU time on a 150 MHz Sun UltraSparc workstation with 512 MB of RAM. To the credit of our property-specific testbench generation approach, consisting of analysis of an abstract model followed by intelligent testbench generation, we were able to perform the same check in about 1 hour, by generating and simulating an input sequence which showed that the CPU had to wait for 6 cycles after making a request.

The details of our abstract model analysis are as follows. We first used explicit traversal over the control states to determine the cone of influence, and to automatically derive an abstract model. For this design, flip-flops at levels greater than a dependency level of 3 from atomic propositions were abstracted by making them free inputs. This led to a reduction in the number of flip-flops to 167. The entire abstraction process is very fast, and took less than 1 minute. Next, we performed model checking on this abstract model including reachability analysis. Combined with generation of the Witness Graph information in BDD form, it took 3779 seconds.

For testbench generation, we experimented with both the Verilog simulator (*Modeltech*) and the in-house C-based HDL simulator. The entire testbench source code (written in Verilog+C for the former, and C++ for the latter) was automatically generated from the Witness Graph, and took negligible time. The final simulation, which showed the actual witness on the complete *MIF* design, was also very fast, and took less than 1 minute in each case.

A comparison between the performance of VIS on the original model, which required 41 hours, and our testbench generation prototype on the abstract model, which required about 1 hour, highlights the speed advantage to be gained by our approach. We hope to demonstrate in the near future that even coarser abstractions would have sufficed for this example.

We believe that this experience with the verification of the CTL property on a real-life example highlights the viability of our approach, and the potential for its incorporation in testbench generation flows used in the industry.

5. Conclusions

We have presented algorithms for generating a Witness Graph, which captures all witnesses or counter-examples in an abstract model of a design with respect to a given correctness property. These algorithms iteratively employ abstraction, approximate model checking, pruning, and refinement, with many novel features in comparison to existing techniques. We have also presented a backtracking search algorithm that uses the Witness Graph for finding a concrete witness or counter-example during simulation. Based on these algorithms, we have developed an automatic *intelligent testbench generation* framework compatible with generic HDL-based simulation environments. We have been able to demonstrate on a real in-house LSI design that such an approach can lead to significant reduction in the time required to analyze the design for a CTL property and find a witness.

References

- R. K. Brayton et al. "VIS: A system for verification and synthesis", In *Proceedings of CAV*, volume 1102 of *LNCS*, pages 428-432, June 1996.
- [2] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation", *IEEE Transactions on Computers*, C-35(8):677-691, Aug. 1986.
- [3] E. M. Clarke, O. Grumberg, and D. Peled, "Model Checking", The MIT Press, 1999.
- [4] E. M. Clarke, O. Grumberg, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement", In *Proceedings of CAV*, volume 1855 of *LNCS*, pages 154-169, 2000.
- [5] M. Ganai, A. Aziz, and A. Kuehlmann, "Augmenting simulation with symbolic algorithms", In *Proceedings of DAC*, June 1999.
- [6] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, "Coverage estimation for symbolic model checking", In *Proceedings* of DAC, pages 300-305, June 1999.
- [7] Y. Hoskote, D. Moundanos, and J. A. Abraham, "Automatic extraction of the control flow machine and application to evaluating coverage of verification vectors", In *Proceedings of ICCD*, pages 532-537, Oct. 1995.
- [8] C.-Y. Huang and K.-T. Cheng, "Assertion checking by combined word-level ATPG and modular arithmetic constraint-solving techniques", In *Proceedings of DAC*, pages 118-123, 2000.
- [9] A. Kuehlmann, K. McMillan, and R. K. Brayton, "Probabilistic state space search", In *Proceedings of ICCAD*, 1999.
- [10] R. P. Kurshan, Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach, Princeton University Press, 1995.
- [11] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi, "Tearing based abstraction for CTL model checking", In *Proceedings of ICCAD*, pages 76-81, Nov. 1996.
- [12] J. Lind-Nielsen and H. R. Anderson, "Stepwise CTL model checking of state/event systems", In *Proceedings of CAV*, volume 1633 of *LNCS*, pages 316-327, 1999.
- [13] D. E. Long, Model Checking, Abstraction and Modular Verification, PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1993.
- [14] Verisity Design, Inc. Specman Elite, http://www.verisity.com/html/specmanelite.html.
- [15] K. Wakabayashi, "C-based Synthesis Experiences with a Behavior Synthesizer "Cyber" ", In *Proceedings of DATE*, pages 390-393, 1999.
- [16] C. Han Yang and David L. Dill, "Validation with guided search of the state space", In *Proceedings of DAC*, June 1998.
- [17] J. Yuan, J. Shen, J. Abraham, and A. Aziz, "On combining formal and informal verification", In *Proceedings of CAV*, volume 1254 of *LNCS*, pages 376-387, June 1997.