Simulation Coverage Enhancement Using Test Stimulus Transformation

C. Norris Ip
ip@cadence.com, Cadence Berkeley Labs, California

Abstract

This paper introduces the concept of abstract state exploration histories to a simulation environment, and presents a test stimulus transformation (TST) technique to improve simulation coverage. State exploration histories are adapted from reachability analysis in Formal Verification. In TST, an aggressively abstracted state exploration history is maintained during simulation. While this history is being collected, test stimuli from an existing test bench are transformed on-the-fly to explore new scenarios that are not in the history. The results showed that a 3-fold increase in transition coverage for a cache coherence controller, and 10 times faster coverage convergence for a MPEG2 decoder can be achieved.

1. Introduction

Simulation is traditionally the main tool to perform design validation and logic verification. However, as the complexity of a design increases, it is difficult to write test stimuli that would simulate every aspect of the design. The simulation may only cover a small portion of all possible scenarios. This is known as the coverage problem.

Another approach in validation is formal methods using reachability analysis and model checking [12,2]. However, in spite of the advances in formal verification techniques, the complexity of a typical design still exceeds the capacity of existing formal verification tools. Therefore, a designer can only use it to verify small units within a design or an abstract view of the design, which leads to many difficulties in practice.

This paper proposes a technique called Test Stimulus Transformation (TST), which adapts idea from formal verification into a simulation environment. The main innovations behind TST are the concept of abstract state exploration histories and the strategy of performing transformation of test stimuli during simulation (on-the-fly).

Other semi-formal techniques have often concentrated on generating tests from scratch before simulation. For example, in [7,5,9,3], high-quality abstract tests are extracted from a reduced finite-state-machine (FSM) model of a design. These tests are then converted to concrete tests in an ad hoc manner so that they can be applied to the implementation. Our strategy of performing transformation of test stimuli on-the-fly leads to a technique that is easier to use, and avoids the problem of translating abstract tests from the reduced FSM model to concrete tests for the simulation of the actual design.

The concept of state exploration history is adapted from reachability analysis in formal verification. In reachability analysis, the set of examined states and the set of states whose successors have not been examined are maintained. This avoids redundant work and guarantees termination. In traditional simulation, no history is maintained, and a simulator may repeatedly exercise the same scenarios without going to other corner cases. Therefore, we propose the use of state exploration history to provide information for avoiding redundant work in simulation.

Because a complete history is expensive to maintain, TST is designed to allow aggressive abstraction and approximation. Compared to abstractions in formal verification (e.g. [1]), this abstraction process is flexible and easy to use, because the simulation is performed on the actual design, not on the abstracted states.

Apart from the concept of state exploration history, the development of TST focuses on integration and usage. The main objective is to provide good simulation coverage enhancement while minimizing changes to existing simulation flows. While it can generate tests from scratch, it is designed to transform existing test suites to increase coverage and to utilize the knowledge and the effort that have been put into these tests.

The current implementation of TST is simple to use; it does not require constraints to be specified at the inputs, but it does not require a declarative language to describe the test benches, nor their symbolic manipulation. As shown in Section 4, simple annotations on existing test suites have generated more than 3-fold increase in transition coverage and 10 times faster coverage convergence. We are currently designing a test bench authoring tool to automate the annotation process.

In the next section, the definition of state exploration history and how it can be used in a simulation are presented. In Section 3, TST is presented with a comparison to related work. A few case studies and a conclusion are presented at the end of the paper.

2. State Exploration History

The main innovation in the Test Stimulus Transformation (TST) approach is the use of abstract state exploration history
in a simulation environment. This concept provides the basis from which the transformation is performed.

A state exploration history is a summary of what stimuli have been used to exercise the design, and what design states have been exercised. Traditionally this concept has been used in formal verification algorithms. For example, consider the simple reachability analysis algorithm shown in Figure 1.

```plaintext
Reached = Unexpanded = Initial State;
while (Unexpanded ≠ ∅) {
    remove a state s from Unexpanded;
    for (each action a) {
        s' = a(s);
        if (s' ∉ Reached)
            put s' in Reached and Unexpanded;
    }
}
```

**Figure 1: reachability analysis**

In this algorithm, the states previously examined are stored in the set Reached. If a state s is in Unexpanded, no stimulus has been applied to the state s. If a state s is in Reached but not in Unexpanded, all stimuli have been applied to the state s, and the corresponding next states have been included in Reached. The two sets Reached and Unexpanded together summarize a state exploration history.

Because the algorithm maintains a complete state exploration history, it achieves 100% coverage when the algorithm terminates. However, for complex designs, the state space may be so huge that it cannot complete the analysis in the limited amount of available time and memory. This is known as the state explosion problem.

On the other hand, a simulation environment exercises a design description by feeding stimuli from a test bench. The coverage information is obtained by a coverage tool, typically for code coverage, signal coverage, event coverage, and state machine coverage. A manual process is used to write new test benches to exercise missing coverage.

In order to improve the stimulation quality in a more automatic fashion, the idea of state exploration history is adapted to simulation. The goals of this adaptation are:

- to improve stimulation coverage in general, but not necessarily to achieve 100% coverage,
- to be applicable for any design complexity, with minimal computation overhead and a fixed amount of memory.

A typical simulation relies on assertions, golden models, and event monitors to determine the correctness of the design. Because the history is not used for detection of errors and the abstraction is performed on the history but not on the design, aggressive abstraction and approximation of the history can be performed. There is no restriction in how the abstractions are performed, and the user is not required to understand formal verification to use them properly.

The remainder of this section defines state exploration history and discusses how it is collected during simulation.

### 2.1 Definition

A design can be captured using a labeled transition system.

**Definition 1:** A labeled transition system (LTS) is a tuple

\[ T = (Q, A, Δ) \]

where \( Q \) is a set of states, \( A \) is a set of actions, and \( Δ \) is a function \( Q \times A \rightarrow Q \).

In a typical design, the set of states \( Q \) is determined by the set of state-holding elements, and the set of actions \( A \) is defined by the set of inputs. There may also be constraints on the inputs, which can be specified as a function \( Q \rightarrow 2^A \), mapping each state to a set of legal actions. Sometimes we use the notation \( a(s) \) to specify \( Δ(s, a) \) when \( Δ \) is clear from the context.

Using a LTS to represent a design, we can define a complete state exploration history as:

**Definition 2:** A complete state exploration history w.r.t. a LTS \( T = (Q, A, Δ) \) is defined as a tuple \( H_T = (R, Ω) \)

where \( R \) is a subset of \( Q \) and \( Ω \) is a function \( R \rightarrow 2^A \).

In the algorithm shown in Figure 1, the two variables Reached and Unexpanded represent a complete state exploration history. Reached corresponds directly to \( R \) in the history; a state \( s \) in Unexpanded corresponds to a mapping of \( s \rightarrow ∅ \) in \( Ω \); and a state \( s \) in Reached but not in Unexpanded corresponds to a mapping of \( s \rightarrow A \) in \( Ω \).

In a simulator, each simulation step takes a stimulus \( a \) from the test bench, and applies it to the current state \( s \) to obtain a successor \( a(s) \). If we were to maintain a complete state exploration history during simulation, each simulation step changes the current history \( H_T = (R, Ω) \) to

\[ \langle R', Ω' \rangle = \langle R \cup a(s), \ Ω \cdot (s \rightarrow (Ω(s) \cup a)) \rangle \]

The formula \( Ω \cdot (s \rightarrow (Ω(s) \cup a)) \) is the same as \( Ω \) except it maps the state \( s \) to the original set of actions plus \( a \). The resulting history has higher coverage if \( a \) is not in \( Ω(s) \).

However, the use of a complete state exploration history is limited by the state explosion problem. So we need abstraction. Instead of performing abstraction on the LTS, we perform abstraction on the state exploration history:

**Definition 3:** An abstraction function w.r.t. an abstract state set \( Q' \) and an abstract action set \( A' \) is a tuple
\[ \zeta = \langle \xi_Q, \xi_A \rangle, \text{ where } \xi_Q \text{ is a function } Q \to Q \text{ and } \xi_A \text{ is a function } A \to A'. \]

In subsequent discussion, we use \( \zeta \) instead of \( \xi_Q \) or \( \xi_A \) when it is clear from the context.

**Definition 4:** An abstract state exploration history w.r.t. a history \( H_T = \langle R, \Omega \rangle \) and an abstraction function \( \zeta \) is defined as a tuple \( H_{T, \zeta} = \langle R', \Omega' \rangle \), where \( R' \) is a smallest subset of \( Q \) such that \( \forall (q \in Q) (q \in R \Rightarrow \zeta(q) \in R') \) and \( \Omega' \) is the smallest subset of \( 2^A \) such that
\[ \forall (s \in Q, a \in A) (a \in \Omega(s) \Rightarrow \zeta(a) \in \Omega'(\zeta(s))) \]

Using an abstraction function \( \zeta \), a simulation step can update the abstract history from
\[ H_{T, \zeta} = \langle R', \Omega' \rangle \]
to
\[ \langle R' \cup \zeta(a(s)), \Omega \cdot (\zeta(s) \to (\Omega(\zeta(s)) \cup \zeta(a))) \rangle. \]

### 2.2 Implementation

A state exploration history can be captured in a hash table, with keys being states in \( R \) and the data being subsets of \( A \). Three kinds of abstractions are used in TST.

**State abstraction:**

A subset of state holding elements in the design are specified as the representatives, so that the abstraction \( \zeta \) maps two states with the same values in these representatives to the same abstract state.

Typically, latches with important architectural information should be selected as representatives, so that the coverage will measure important architectural events, and ignore irrelevant differences between two states.

In the current TST implementation for Verilog simulators, a simple PLI procedure\(^1\), called `$tstObserver`, takes Verilog signals as arguments, and generates a hash table for the abstract state exploration history. While it may be difficult to automate the process, it is possible in some restricted cases; for example, some commercial tools can automatically extract FSM states of a design, which can be used as the representatives.

**Stimulus partition:**

The set of possible stimuli (actions) are partitioned into equivalence classes, so that the abstraction \( \zeta \) maps two stimuli in the same equivalence class to the same abstract stimulus.

For example, `read` instructions for a processor can be classified into two equivalence classes, one that causes a page-fault and one that does not cause a page-fault.

In the current TST implementation for Verilog simulators, a simple PLI procedure\(^1\), called `$tstTransformer`, takes a set of arguments as the range of possible values, the constraints, and the partitions at each input.

**Approximation to a fixed memory usage:**

In order to avoid excessive memory usage, a bit-state hashing technique is used in the hash table. Bit-state hashing was originally introduced in reachability analysis to reduce memory usage [6]. It uses the hash function as an abstraction, so that two states with the same hash values are mapped to the same abstract state.

In the current TST implementation, the user can specify a specific memory size, and TST will automatically select a hash table with bit-state hashing using an appropriate number of buckets, to fit into the specified memory size.

With these abstractions and approximation, the memory requirement for maintaining an abstract state exploration history can be bounded to a fixed amount regardless of the complexity of a design.

### 3. Test Stimulus Transformation

Apart from the concept of state exploration history, the development of TST focuses on integration and usage:

- use an abstract state exploration history as the target for coverage enhancement,
- use the information in an abstract state exploration history to synthesize test stimuli,
- perform synthesis of test stimulus during simulation (on-the-fly instead of preprocessing),
- utilize the knowledge embedded in an existing test suite.

In TST, as shown in Figure 2, the simulator is running in parallel with another tool executing a set of `transformers` to improve the stimuli from an existing test suite. The description of the design and a test suite are taken from the usual simulation flow. During simulation, the transformers collect an

---

1. PLI (Programming Language Interface) is a standard interface in Verilog Simulators to allow Verilog descriptions to call C procedures in a user’s library [13].
abstract state exploration history from the simulator, examine
the stimulus from the test suite for the current simulation step,
and transform them before passing them as inputs to the simu-
lator.

Figure 2: test stimulus transformation

3.1 Coverage Enhancement Strategy

The concept of abstract state exploration history provides a
concrete coverage metric upon which the coverage is to be
improved. The job of the transformers in TST is:

Given a state $s$ from a simulator and an abstracted state
exploration history $H_s$, determine a stimulus $a$ such that
the abstracted state exploration histories in subsequent
simulation steps will have higher coverage.

The current TST implementation attempts to select a stimulus
$a$ such that $\zeta(a)$ is not in the set $\Omega(\zeta(s))$ from the current
history. If such a stimulus exists, applying it to the simulation
will definitely generate an abstract history with better cover-
age in the next simulation step. If such a stimulus does not
exist, the transformer selects one using a weighted randomiza-
tion strategy to increase the change of finding such stimuli in
future simulation steps.

3.2 Transformation Strategy and Algorithms

Three transformation algorithms have been implemented
using the same strategy discussed in the previous subsection.
However, they differ in how they use the stimuli from the test
bench and how they respond when the basic enhancement
strategy cannot find a stimulus $a$ such that $\zeta(a)$ is not in the
set $\Omega(\zeta(s))$.

The knowledge in a test bench could be very important in
the test stimulus transformation process. With a complex
design, it is impractical to exercise every possible scenario. A
test bench is usually written with a specific aspect of the
design in mind, and the sequence of stimuli from a test bench
may drive the simulation to the right direction. This informa-
tion can be used in several ways:

- start the simulation without transformation, and then turn
  on transformation when the coverage measurement
  indicates a slow down in coverage improvement.
- when the basic enhancement strategy indicates more than
  one good stimulus, choose one that is the same or similar
to the stimulus indicated by the test bench.
- when the basic enhancement strategy cannot find any good
  stimulus, select one using a weighted randomization
  algorithm with a high weight on the stimulus from the test
  bench.

The current implementation uses three different transformation
algorithms for three different cases:

The timing of spontaneous requests:
The test bench may spontaneously generate requests. For
example, agents on a bus may issue requests non-deter-
mministically. In these cases, the quality of a simulation
depends on the exact time at which the requests occur.
When the test bench generates a request, the transformer
may suppress the request. When the test bench can gener-
ate a request but refrain from doing so, the transformer
may initiate a request. Special checks are included in the
transformer so that the transformer may not suppress or
inject too many requests.

The timing of delayed responses:
The test bench may need to provide responses to a request
from a design with some delay. For example, upon receiv-
ing a request, an agent on a bus may need to respond with
data after an indefinite delay.
When the test bench sends a response that could be
delayed, the transformer may delay it until the delivery
will improve the coverage. Special checks are included in
the transformer so that it will not delay a response forever.

The values of inputs:
The value of inputs, regardless of whether it is a request, a
response, or an instruction within a stream, are usually
taken from a set of possible values. The transformer may
change it to another value that increases the coverage.
Special checks are included to make sure the user-speci-
fied constraints are satisfied.

3.3 Using Multiple Histories and Transformers

Sometime, it is more efficient to use different abstracted state
exploration histories for different transformers, as shown in
3.4 Comparison to Related Work

There are two basic functional verification strategies: simulation and formal verification, with two distinct limitations:

- **Redundancy in simulation**: A simulation run may explore the same part of the state space over and over again, without going to the corner cases.

- **Excessive memory usage in formal verification**: A reachability analysis algorithm needs to store a complete history of everything that it has examined to avoid redundant work and to guarantee completion.

The TST technique proposed in this paper is designed to achieve a good balance between these two extremes.

There are other semi-formal methods that combine techniques in formal verification and simulation. Most of them have concentrated on automatic test generation: generate a set of good tests by exhaustively analyzing a reduced FSM model before simulation. In [7,5,9,3], reduced FSM models were obtained by abstracting out the data path, partitioning the state variables, or using temporal abstraction. Abstract tests are then extracted from these models using techniques such as state transition tour. Finally this set of tests is converted to concrete tests so that they can be applied to the simulation.

The TST technique described in this paper takes a different approach: instead of performing state exploration on a reduced FSM and generating tests from scratch and before running the simulation, the transformers in TST are constantly interacting with the simulator to fine-tune the stimuli. It also utilizes the knowledge captured in existing test suites to target specific aspects of the design.

Some semi-formal methods are variants of the basic formal verification algorithms. Both [10] and [11] have proposed techniques to change the search priority within a formal verification algorithm so that errors can be detected early in the process. [4] have proposed using automatic test pattern generation technique (ATPG) to analyze the design and to generate test stimulus. They rely on the ability to save and restore states easily. On the other hand, TST is designed for a simulation environment, which is typically more expensive to save and restart from a different state when compare to a formal verification tool. Therefore, it is designed to avoid save and restore, with minimal changes to an existing simulation flows and existing test suites.

Furthermore, in these variants of formal verification techniques, detection of errors are performed directly on the LTS represented by the history, which corresponds either to the description of the design, or a conservative approximation of it. In TST, the history can be arbitrarily approximated and abstracted, and the detection of errors are performed on the actual simulation without any abstraction or approximation.

4. Case Studies

Simulations on two practical designs are used to evaluate the test stimulus transformation technique, and their results are summarized in this section.

4.1 DASH Cache Coherence Protocol

TST was evaluated on the DASH cache coherence protocol [8]. Using a discrete event simulator for system level design, a 4 processor system was simulated using three sets of test stimuli: 1) typical workload, 2) heavy workload (10 times more requests from the processors than typical workload), and 3) transformed typical workload.

Stimuli from 1) and 2) are obtained by a test bench that generates requests at random time with random values for the request types, addresses, and data.

The TST algorithms are implemented as four generic library blocks: one for collection of state exploration history and three kinds of transformers for the three algorithms discussed in Section 3.2. Stimuli from 3) are obtained by instantiating a history collection block, with the FSM states in the four cache controllers as representatives, and instantiating several transformers to change the request types and the delay in the communication network.

The results shown in Figures 4, 5, and 6 are obtained using 1MByte of memory to store the state exploration history. The state coverage measurement is the size of Ω in the history, and the transition coverage is the number of distinct current-state/successor pair, which roughly corresponds to the effect of improving Ω in the history.

These coverage measurements are presented in the graphs in Figure 4 and Figure 5. At the end of 10^8 cycles, about 8% increase in state coverage and about 3 times more transition coverage are obtained with TST. Without transformation, it
takes about 2 times more simulation cycles to reach 30,000 abstract states, and about 10 times more simulation cycles to reach 150,000 abstract transitions. Furthermore, these simulations have shown that increasing the workload or the simulation time may not improve the coverage, but intelligently transforming the test stimuli can significantly improve the coverage.

Figure 6 shows another important parameter: simulation speed. We have achieved more than double in coverage rate, with 10% overhead. This has confirmed that TST is an effective technique in improving simulation coverage.

The TST algorithms are implemented as a set of generic PLI procedures and the test bench is modified to include calls to these procedures. The change is simple: At initialization phases, a PLI procedure, $\text{tstObserver}$, is called to specify the FSM states in the controller as the representative for state abstraction, and a PLI procedure, $\text{tstTransformer}$, is called several times to create transformers for picture type, motion type, etc.

For each statement with $\text{random}$, we add a call to the transformer:

\[
picture_type = \text{random} \mod 10;
\]

where $pTypeIdx$ is an integer corresponding to the transformer of picture type.

The results shown in Figures 7 and 8 are obtained using 1MByte of memory to store the abstract state exploration history. The statement, decision, and expression coverage, and coverage metric are taken from a commercial coverage measurement tool. With the simple changes in the test bench, 30% increase in state and transition coverage, 15% increase in statement coverage and 20% increase in expression coverage are obtained. Figure 8 shows that, without TST, 10 times more simulation time is required to reach the same coverage.

<table>
<thead>
<tr>
<th>simulating 10^8 cycles</th>
<th>total transition coverage</th>
<th>transition coverage per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>typical</td>
<td>6632s</td>
<td>168240</td>
</tr>
<tr>
<td>heavy</td>
<td>20567s</td>
<td>162710</td>
</tr>
<tr>
<td>transformed</td>
<td>7325s</td>
<td>439750</td>
</tr>
</tbody>
</table>

**Figure 6: simulation speed**

### 4.2 MPEG2 Decoder

TST was also evaluated using the motion controller of a MPEG2 decoder, obtained from a design service team at Cadence Design Systems. Both the test bench and the design were written in Verilog and simulated using a commercial Verilog simulator.

The test bench for this controller was written in a directed randomization format, in which a specific sequence of instructions are used with randomized values for some fields. The fields that are randomly generated include picture type, motion type, structure type, field type, motion vector format and whether to do chroma 422. These random values are typically generated using a simple call to $\text{random}$, such as,

\[
picture_type = \text{random} \mod 10;
\]

<table>
<thead>
<tr>
<th>simulating 3x10^5 cycles</th>
<th>without TST</th>
<th>with TST</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>253</td>
<td>330</td>
</tr>
<tr>
<td>transition</td>
<td>282</td>
<td>374</td>
</tr>
<tr>
<td>statement</td>
<td>35%</td>
<td>40%</td>
</tr>
<tr>
<td>decision</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>expression</td>
<td>50%</td>
<td>61%</td>
</tr>
<tr>
<td>Coverage Merit</td>
<td>0.75</td>
<td>0.827</td>
</tr>
</tbody>
</table>

**Figure 7: coverage for the MPEG2 decoder**
5. Conclusion

The concept of abstracted state exploration histories is introduced in this paper as a coverage metric for simulation. A test stimulus transformation strategy is proposed to utilize this concept to improve simulation coverage. Although full coverage is not guaranteed, lots of the redundant work due to ineffective test stimuli are removed.

The resulting tool fits well into the traditional simulation methodology; it is easy-to-use and does not require any knowledge about formal verification. It also serves as a natural step from traditional simulation towards formal verification, since it introduces the concept of environment constraints (degree of freedom in test stimuli). While there are many requirements in the use of constraints in formal verification, the constraints specification in a test suite can be arbitrarily strict to target certain aspect of the design.

Using a transformation framework instead of a test generation framework, this technique allows reuse of the investment put into a traditional simulation test suite and the simulation technology. Good coverage enhancements have been obtained using a simple transformation strategy, and further enhancement should be possible using more advanced techniques to perform the transformation.

We are currently investigating how to extend the coverage enhancement strategy presented in Section 3.1, to further improve the history in one and in \( n \) simulation steps. Potential strategies include using ATPG to guarantee a larger \( R' \) in the next stimulation steps (c.f. [4]), or to discover a sequence of \( n \) stimuli that will guarantee a history with improved coverage in \( n \) simulation steps.

Acknowledgment

The author would like to thank Jerry Burch, Vigyan Singhal, and Howard Wong-Toi for valuable discussions throughout the project, Gary Young for providing the MPEG decoder as a case study, and the anonymous reviewers and Andreas Kuehlmann on feedback in the presentation.