Efficient Assertion Based Verification using TLM

Ali Habibi, Sofiène Tahar, Amer Samarah, Donglin Li and O. Ait Mohamed
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve West
Montreal, Quebec, Canada H3G 1M8
Email:{habibi,tahar,amer_sam,li_don,ait}@ece.concordia.ca

Abstract

Recent advancement in hardware design urged using a transaction based model as a new intermediate design level. Supporters for the Transaction Level Modeling (TLM) trend claim its efficiency in terms of rapid prototyping and fast simulation in comparison to the classical RTL-based approach. Intuitively, from a verification point of view, faster simulation induces better coverage results. This is driven by two factors: coverage measurement and simulation guidance. In this paper, we propose to use an abstract model of the design, written in the Abstract State Machines Language (AsmL), in order to provide an adequate way for measuring the functional coverage. Then, we use this metric in defining the fitness function of a genetic algorithm proposed to improve the simulation efficiency. Finally, we compare our coverage and simulation results to: (1) random simulation at TLM; and (2) the Specman tool of Verisity at RTL.

1. Introduction

Transaction Level Modeling (TLM) emerged from the need for fast and efficient system architecture exploration, where both hardware and software components are tight together in an abstract and shallow way. The notion of the clock, inherently used at the Register Transfer Level (RTL), is replaced by direct function calls, which immediately results in faster simulation execution. For faster prototyping, object-oriented system level languages have been of great impact. Consequently, one of the C++ based proposals, SystemC [9], is in its way to become a standard for TLM [9].

Guiding the simulation towards a set of specific objectives is essential for ensuring good coverage. Simulation objectives are commonly defined as assertions. Then, coverage is measured in different ways [6]: code coverage, condition coverage, etc. For efficient verification, it is imperative to consider functional coverage. Raising the abstraction level abstracts the implementation details while preserving the system's behavioral aspects. Such an operation contributes to the reduction of the system's state space. In this paper, we use AsmL (Abstract State Machines Language) [7] to model the design at the transaction level. This choice is firmly linked to the feasible and practical generation of the system's finite state machine (FSM) from AsmL models. The generated FSM plays the role of a golden model to evaluate the functional coverage.

We propose to take advantage of TLM to guarantee higher RTL functional coverage. Optimizing tests at TLM and reusing them at RTL is a quite trivial and straightforward tactic for enhancing simulation results. To bring into play such an idea, two main questions must be answered at the transaction level: (1) how to measure the coverage? and (2) how to improve it?

As a solution to the coverage measurement question, we propose a layered design-for-verification approach involving both TLM and RTL. At TLM, the design is modeled in AsmL, where communication between system's components relies on direct functional calls. This simplified model is more suitable for the generation of the system's FSM. Raising the level of abstraction tackles the problem of state explosion, usually faced with RTL designs. Once the FSM is generated, we define a functional coverage in terms of state space coverage.

In order to improve the coverage, we propose to use a genetic algorithm (GA) aiming to optimize the random test generation. The basic concept is to find a good random distribution of the inputs' ranges offering a higher level of coverage. The final output of this operation is a test vector generator with a high coverage rate (at least in comparison to blind random simulation) w.r.t. a predefined objective. The test generator produced using the GA optimization technique at TLM is reused to validate the RTL design. We propose, in this paper, to compare the coverage results for RTL using our GA and using the random simulation provided by the commercial tool Specman of Verisity [12].

The rest of this paper is organized as follows: Section 2 describes related work. Section 3 presents our proposed methodology to improve the assertion coverage. Section 4 propose metrics to evaluate the coverage. Section 5 depicts the used genetic algorithm to enhance the coverage at TLM. Section 6 illustrates our methodology using the Look-Aside interface standard including experimental results. Finally, Section 7 concludes the paper.

2. Related Work

Genetic algorithms have already been used for a broad range of applications. Godefroid *et al.* [1] addressed the exploration of large state spaces. This work, based on BDDs, was restricted to simple Boolean assertions which is not suitable for high level languages like SystemC. Habibi *et al.* proposed in [3] to use GA for improving assertion coverage for SystemC RTL models. However, neither a coverage metric nor a precise fitness function has been provided. In

3-9810801-0-6/DATE06 © 2006 EDAA

this work, we propose to: (1) use AsmL for transaction level models; (2) define the coverage as function of the system's FSM (at TLM); (3) initialize the GA using the information gathered from the assertion and the system's FSM; and (4) employ TLM designs to identify efficient test generator for RTL implementations.

A variety of EDA tools provide test generation with assertion coverage, e.g., Specman Elite [12] of Verisity. They offer a user-defined constrained random simulation, for RTL designs, where the coverage is a function of the number of times the assertion was executed. In this work, we first define the coverage metrics for TLM as a function of the portion of the assertion's state space that has been covered. Then, we compare the coverage results obtained using our approach to the output of Specman Elite tool for the same RTL model.

3. Proposed Methodology

Performing a full coverage of a system's state space using simulation is not feasible. Consequently, more focus was given to develop smart verification approaches. In the methodology we propose, we aim to make use of two features: transaction models and genetic algorithms.

Transaction models run faster than timed models [4]. Avoiding clocks and raising up the level of abstraction by using channels and direct functional calls accelerates the simulation. These models are conceptually closer to the system's specification, which is, generally, a collection of properties that could be verified by simulation (as assertion monitors). In order for faster simulation to guarantee better verification, a precise measurement for the coverage is needed. Classical ways to measure the coverage at RTL (code, condition, etc.) are not directly linked to system's specification. We propose to take advantage of transaction models to define assertions that could be used to verify the final RTL product.

Generating FSM from transaction models provides a way to define assertion as sequence of states. In the methodology described in Figure 1, we use AsmL as TLM modeling language. Two main reasons influenced our choice: the language features and the FSM generation algorithm available for models written in this language. For instance, AsmL is an object-oriented language rich with several mathematical constructs and data abstraction. A number of algorithms have been developed around this language, collected under the AsmL tool (Asmlt) [7]. In this work, we make use of the algorithm generating FSMs from AsmL models in order to define the functional coverage as function of the system's states.

We use the FSM generated from the system's TLM model as a guidance for relevant variables and input values for testing a specific operation. Optimizing the coverage becomes an issue of guiding the simulation to feed the system with particular values. Unfortunately, it is not always possible to find a direct relation between the variables involved in the assertion and the system's inputs. Hence, what we can extract from the FSM is an over-approximation of this relation. It follows that we cannot define precisely the complete set of tests to validate a specific functionality. A solution to enhance the test generation process is to use genetic algorithms. We use the initial knowledge gathered from the generated FSM as an initialization for a genetic algorithm aiming to enhance the assertions' coverage. The input of the algorithm is a set (population) of test vector generators. After applying a number of tests, the generators are updated to form a new community. The fitness is a function of the assertion coverage.

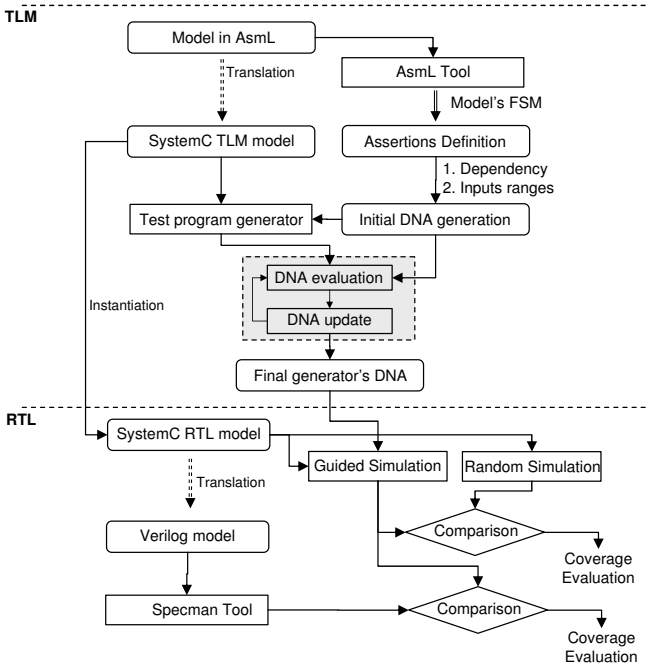

Figure 1. Proposed Methodology.

Once a satisfactory coverage is achieved at TLM, we compare the achieved coverage using the best generation of test generators output of the previous phase to the coverage obtained using: (1) random simulation; and (2) Specman. The first comparison aims to show that improvement of the coverage is preserved when lowering the abstraction level from TLM to RTL, while the second illustrates the efficiency of our proposed approach when compared to commercial tools.

3.1. Transaction Level Model in AsmL

The system's TLM model in AsmL includes a light description of the system's functionalities. All the components are communicating using transactions involving direct functional calls. The simulation environment includes the notion of *updates*, i.e., a variable value is not changed until an update is requested. For this reason, we embedded in AsmL a light simulation environment in order to manage events and processes. The system components are objects instantiations of classes (also called $Module^{A}_{TLM}$) defined as follows:

Definition 3.1. (AsmL TLM Module: $Module^{A}_{TLM}$)
An AsmL TLM module is a set $\langle AS_{DMem}, AS_{Mth}, AS_{Ctr}\rangle$, where AS_{DMem} is a set of the module data members, AS_{Mth} is a set of methods (functions) definition and AS_{Ctr} is the module constructor.

For every method in AS_{DMem} corresponds a Boolean pre-condition enabling its execution. This is a critical issue in constructing the actual AsmL TLM design because a wrong definition of the pre-condition may totally change the behavior of the system and consequently modify the verification results. The pre-condition

rules define the way of communication between the design's components. The design is defined as a collection of modules and an initialization method according to following:

Definition 3.2. (AsmL TLM Design: $Design^{A}_{TLM}$)
An AsmL TLM design is a set $\langle LModule^{A}_{TLM}, INIT\rangle$, where $LModule^{A}_{TLM}$ is a set of AsmL TLM modules and *INIT* is the initialization function of the model.

In order to perform adequate partial and total updates, we make use of a light simulation manager (in AsmL). This simulator includes an initialization function that is executed after all design's modules have been initialized. The same method includes a second pre-condition setting that the method is only executed at the initialization phase. This illustrates how the pre-condition constructor is used to manage the exploration algorithm performing the reachability analysis.

3.2. FSM Generation

We use an FSM generation algorithm defined by Gurevich et *al.* in [2]. It requires the following inputs: domains, methods, actions and variables. The transitions in the FSM are the method calls (including argument values) in the test sequences. The methods in the model program that appear in the transitions are called actions. The states in the FSM are determined by the values of selected variables in the model program, called state variables. The algorithm keeps track of the actions while recording the states it visits (*exploration* process). The FSM generation process requires a set of Boolean guards in order to reflect the state distinction that the model designer cared enough about to make explicit. The algorithm takes a distinguishing sequence as an additional input to produce corresponding equivalence classes, called *hyperstates*. The required items for the FSM generation algorithm [2] are:

- data types and static functions
- declarations of state variables $v_1, v_2, \dots v_s$ (s is the total number of states) that characterize the state space of the considered system.
- rules that describe the transition relation of the system: $c_1, c_2, \dots c_v$ (c refers to a Boolean condition (rule), v is the total number of rules).

The classical problem challenging FSM based approaches is state explosion. For this reason, the notion of states indistinguishability represents the main key feature of the algorithm in [2] in comparison to other techniques. Furthermore, this notion fits well to the conceptual structure of TLM where states can be combined according to their affiliation to a specific transaction.

Definition 3.3. (States Indistinguishability)
Let $\mathcal{C} = \{c_i(v_1, v_2, \dots v_s), i \in \{1 \dots n\}\}$ be a set of Boolean conditions on state variables. Two states $s_a = \{a_1, a_2, \dots a_s\}$ and $s_b = \{b_1, b_2, \dots b_s\}$ are indistinguishable if: $\forall c \in \mathcal{C}$, $c(s_a) = c(s_b)$.

Defining the conditions in $\mathcal{C}$ is an additional effort required in building the TLM models in AsmL. A good tactic to surmount this problem is to define a specific condition c in $\mathcal{C}$ for each transaction. Hence, a natural link will be defined between transactions and hyperstates.

3.3. Assertion Scope

Considering a generated FSM, the assertion could be represented as a collection of hyperstates and transitions. Classical FSM coverage, used at RTL, always deals with the full system's FSM (state or transition coverage) [6]. In contrast, our target is not to go for all possible combinations raising from the system's FSM. We aim to cover a set of hyperstates (for the sake of simplicity, we will use the word state to refer to hyperstates in the rest of this paper) and transitions in the generated FSM. In following, we define an assertion as a collection of state variables and Boolean conditions involving at least one of the assertion's state variables.

Definition 3.4. (Assertion Definition: A)
Let $Design^{A}_{TLM}$ be an AsmL model, $\mathcal{V} = \{v_1, v_2, \dots v_s\}$ its state variables and $\mathcal{C} = \{c_i(v_1, v_2, \dots v_s), i \in \{1 \dots n\}\}$ be a set of Boolean conditions on state variables. An assertion A is the set $\langle A_v, A_c\rangle$ where A_v is a subset of $\mathcal{V}$ and A_c is a subset of $\mathcal{C}$ involving at least one variable $v_i \in \mathcal{V}$.

The states that we are interested in are those where one of the assertion's guards (conditions) is evaluated to true. This subset is called *Assertion Scope* A_{sco}.

Definition 3.5. (Assertion Scope: A_{sco})
Let $Design^{A}_{TLM}$ be an AsmL model, $\mathcal{F}$ its generated FSM, $\mathcal{V} = \{v_1, v_2, \dots v_s\}$ its state variables, $\mathcal{C} = \{c_i(v_1, v_2, \dots v_s), i \in \{1 \dots n\}\}$ be a set of Boolean conditions on state variables and $A = \langle A_v, A_c\rangle$ be an assertion. Then, the assertion scope A_{sco} = $\{s \in \mathcal{F}$, where $\exists c \in A_c \mid c(s) = true\}$.

The assertion scope A_{sco} collects all the states that are of interest for the assertion A. Definition 3.5 does not guarantee that the assertion's scope will be an automata [11] (the commonly used mathematical model to represent system properties). Nevertheless, since we are interested in verifying the assertion using simulation, defining the assertion scope A_{sco} as a set of states is sufficient.

4. Coverage Evaluation

Considering the assertion's definition and scope, we propose two coverage metrics: sate coverage and transition coverage. In contrast to the classical RTL coverage, we are dealing with hyperstates. A trivial way to define the FSM state coverage is to count the number of states visited by the test vectors over the total number of states. This kind of coverage cannot be used when the FSM is formed from hyperstates. Visiting a hyperstate does not only depend on the state itself but also on the guards elements of the set of Boolean conditions on state variables, $\mathcal{C}$. It is possible for a guard to be true for different combinations of the state variables. Therefore, counting all possible combinations leading to a hyperstate is mandatory for obtaining a true evaluation of the state's coverage.

4.1. State Coverage

Before giving the state coverage's definition, we first introduce the notion of assertion state space, A_{sp}. This latter refers to all the possible state space combinations involved in the assertion as follows:

Definition 4.1. (Assertion State Space: A_{sp})
Let $A = \langle A_v, A_c\rangle$ be an assertion. The assertion state space is:

$$A_{sp} = \{(v_{c1}, v_{c2}, \ldots v_{cs}) \text{ instance of } (v_1, v_2, \ldots v_s) \in A_v \mid \exists c \in A_c \mid c(v_1, v_2, \ldots v_s) = true\}$$

where $(v_{c1}, v_{c2}, \ldots v_{cs})$ is a concrete instance of $(v_1, v_2, \ldots v_s)$.

Considering concrete instances of variables in Definition 4.1 is important because it is possible to use abstract variables in AsmL. The number and the nature of the concrete elements depend on the variable's domain.

Next, we use the assertion's state space definition, in order to evaluate the assertion coverage.

Definition 4.2. (State Coverage: S_{cov})
Let A be an assertion, A_{sco} its scope and A_{sp} its state space. Let T_{sp} = $\{(v_{tc1}, v_{tc2}, \ldots v_{tcs})$ concrete instance of $(v_{t1}, v_{t2}, \ldots v_{ts}) \mid v_{ti} \in A_{sp}\}$ be a set of test vectors. The state coverage obtained by executing T_{sp} is:

$S_{cov} = \frac{Card(T_{sp})}{Card(A_{sp})}$, where *Card* is the set's cardinality.

The state coverage, S_{cov}, computes the fraction of states (element of the assertion's state space) visited during the execution of the test vectors' set. An optimal testing case will provide a state coverage equal to one. For a general graph structure, we cannot guarantee the existence of an optimal test sequence. However, when the FSM is a *connected* graph, We can guarantee that such an optimal case exists (Theorem 4.1). The theorem's proof provides a construction of such a sequence.

Theorem 4.1 *(Optimal Test Sequence for S_{cov})*
Let $Design^{A}_{TLM}$ be an AsmL model and $\mathcal{F}$ its generated FSM. If $\mathcal{F}$ is a connected graph, then, there exists a test sequence T_{sp} such that $S_{cov} = 1$.

Proof If $\mathcal{F}$ is a connected graph, then for any two states a and b in $\mathcal{F}$, there exists a path from a to b. In this case, the proof of the theorem can be done by constructing a test sequence satisfying $Card(T_{sp})$ = $Card(A_{sp})$. Such a test sequence is formed from a set of test vectors each starting from the initial state and getting to a state in the assertion's state space A_{sp}. By defining at least a path for every element in A_{sp}, we ensure that $Card(T_{sp})$ = $Card(A_{sp})$. ■

4.2. Transition Coverage

We define transition coverage, T_{cov}, by identifying all the states involved in a transition from or to a state element of the assertion's space. In the following, we first introduce the assertion transition space, A_{tp}. Then, we will define the transition coverage.

Definition 4.3. (Assertion Transition Space: A_{tp})
Let $Design^{A}_{TLM}$ be an AsmL model, $\mathcal{F}$ its generated FSM, $\mathcal{V} = \{v_1, v_2, \ldots v_s\}$ its state variables, $\mathcal{C} = \{c_i(v_1, v_2, \ldots v_s), i \in \{1 \ldots n\}\}$ be a set of Boolean conditions on state variables and $A = \langle A_v, A_c \rangle$ an assertion. The assertion transition space is:

$$A_{tp} = \{ (v_{c1}, v_{c2}, \ldots v_{cs}) \text{ instance of } v \in \mathcal{V} \mid \exists v_a \in A_v \text{ and } tr \in \mathcal{T} \mid (tr(v, v_a) = true) \vee (tr(v_a, v) = true)\}$$

where $\mathcal{T} = \{tr_1, \ldots tr_m\}$ is the set of the transition in $\mathcal{F}$.

Similarly to the assertion state coverage (see Definition 4.2), we can define the assertion's transition coverage, T_{cov}, using the assertion transition space, A_{tp}.

Definition 4.4. (Transition Coverage: T_{cov})
Let A be an assertion, A_{sco} its scope and A_{sp} its state space. Let T_{tp} = $\{(v_{tc1}, v_{tc2}, \ldots v_{tcs})$ concrete instance of $(v_{t1}, v_{t2}, \ldots v_{ts}) \mid v_{ti} \in A_{tp}\}$ be a set of the test set. The state coverage obtained after executing T_{tp} is: $T_{cov} = \frac{Card(T_{tp})}{Card(A_{tp})}$.

The transition coverage, T_{cov}, computes the fraction of states (element of the assertion's transition space) visited by a test vector. The optimal test case will provide an assertion transition coverage equal to one. When the FSM is a *clique* graph, following theorem guarantees that such an optimal case exists. The proof provides a construction of such a sequence.

Theorem 4.2 *(Optimal Test Sequence for T_{cov})*
Let $Design^{A}_{TLM}$ be an AsmL model and $\mathcal{F}$ its generated FSM. If $\mathcal{F}$ is a clique graph (complete graph), then there exists a test sequence T_{sp} such that $T_{cov} = 1$.

Proof If $\mathcal{F}$ is a clique graph, then for any two states a and b in $\mathcal{F}$, there exists a transition from a to b. In this case, the proof of the theorem can be done by constructing a test sequence satisfying $Card(T_{tp})$ = $Card(A_{tp})$. Such a test sequence is formed from a set of all test vectors each starting from the initial state and getting to a state in the assertion's state space (A_{sp}). By covering all the elements in A_{tp}, we ensure that $Card(T_{tp})$ = $Card(A_{tp})$. ■

5. Coverage Enhancing

Theorems 4.1 and 4.2 guarantee the existence of optimal test sequence (with coverage equal to one) for the particular cases of connected and clique graphs, respectively. However, in the general case, finding or even proving the existence of an optimal test sequence is not trivial. Furthermore, even when an optimal sequence exists, its size could be very large. Hence, in this section, we propose a genetic algorithm based technique aiming to optimize the coverage using a randomly generated test sequences.

5.1. Genetic Algorithm

Genetic algorithms belong to a family of computational models inspired by evolution [5]. They encode a potential solution to a specific problem on a simple chromosomes like data structure and apply recombination operators to these structures to preserve critical information. Since their introduction by Holland [5], genetic algorithms have been applied to a broad range of learning and optimization problems [10]. Typically, a GA starts with a random population of encoded candidate solutions (test generators for our case), called chromosomes. The objective is to maximize the likelihood of generating an optimal solution. This can be guaranteed by: (1) evaluating the *fitness* of each candidate solution in the current population; (2) selecting the fittest candidate solutions; and (3) recombining candidates and mutating them to generate offsprings.

The state variables of the system are classified into three groups: inputs, outputs and internal variables. This step is required in order to define the connection between the system and test vectors generator.

Definition 5.1. (System Variables Classification)
Let $Design^{A}_{TLM}$ be an AsmL model and $\mathcal{V} = \{v_1, v_2, \ldots v_s\}$ a set of its state variables. We classify $\mathcal{V}$ into three subsets:

$\mathcal{V}_{in}$ = $\{v \in \mathcal{V} \mid v$ is an input variable$\}$
$\mathcal{V}_{out}$ = $\{v \in \mathcal{V} \mid v$ is an output variable$\}$
$\mathcal{V}_{int}$ = $\{v \in \mathcal{V} \mid v$ is an internal variable$\}$

In our context, the search space to be explored is the assertion's state space A_{sp} (see Definition 4.1). Candidate solutions are finite sequences of input ranges and probability weights. Each candidate solution is identified by a unique chromosome (a finite string of bits). The information encoded in the chromosomes is composed of: (1) a list of input variables; (2) their domains Definition (5.2), and (3) a probability distribution of the domain (Definition 5.3).

Definition 5.2. (Variable Domain)
Let $Design^{A}_{TLM}$ be an AsmL model and $\mathcal{V}_{in}$ its input variables set. To each variable $v \in \mathcal{V}_{in}$, there is a corresponding domain d.

Definition 5.3. (Variable Domain Distribution: p)
Let $Design^{A}_{TLM}$ be an AsmL model and $\mathcal{V}_{in}$ its input variables set. Then, for every variable $v \in \mathcal{V}_{in}$ corresponds a function p providing the variable's values distribution over its domain.

Definition 5.4. (Chromosome Encoding)
Let $Design^{A}_{TLM}$ be an AsmL model and $\mathcal{V}_{in}$ its input variables set. Then, the test generator's chromosome is the set $Chrom = \langle \mathcal{V}_{in}, D_i, P_i \rangle$.
where:

- $D_i = \{d_1, d_2, \ldots d_s\}$: set of variables domains.
- $P_i = \{p_1, p_2, \ldots p_s\}$: set of variables distributions.

The chromosome encoding is the most important aspect of our algorithm. The variable values generation is controlled by their domains distributions. For example, for a variable of type *Integer*, we can use the following chromosome encoding:

- Variable Domain $d = [-2^{16}, 2^{16} - 1]$
- Variable Domain Distribution: $p([-2^{16}, 0[) = 0.3$ and $p([0, 2^{16} - 1]) = 0.7$

5.2. Fitness Criteria

The proposed fitness criteria serves to guide the genetic search towards covering the whole assertion's state space. The intuitive idea relies on modifying the shape of the variable's domain distribution, p. For the sake of improving the efficiency of the search, we keep track of the best and worst chromosome fitness in each generation; if both fitness values become equal, we increase the mutation rate, in order to help the genetic evolution get out of local maxima. Once there is an improvement in the overall fitness, we restore the original mutation rate to continue the evolution normally.

Definition 5.5. (Test Vector Generator: T_{Gen})
Let $Design^{A}_{TLM}$ be an AsmL model and $\mathcal{V}_{in}$ its input variables set. Then, a test vector generator is defined by a unique chromosome encoding $Chrom$.

For every coverage, there is a corresponding fitness function. For state coverage, S_{cov} (see Definition 4.2), the fitness function is given in following, where the fitness identifies the best test generator by checking for the one having a maximum state coverage.

Definition 5.6. (Fitness Criteria for State Coverage: F_{Scov})
Let $Design^{A}_{TLM}$ be an AsmL model, $\mathcal{V}_{in}$ its input variables set, A be an assertion, A_{sp} its space state and $\mathcal{T} = \{T^1_{Gen}, T^2_{Gen}, \ldots T^n_{Gen}\}$ a set of n-test generators. Then, the fitness criteria corresponding to state space coverage is:

$$F_{Scov} = \max_{T^i_{Gen} \in \mathcal{T}} (S_{cov}) = \max_{T^i_{Gen} \in \mathcal{T}} (\frac{Card(T^i_{sp})}{Card(A_{sp})})$$

where T^i_{sp} is a sequence of test vectors generated by T^i_{Gen}.

The fitness criteria corresponding to the transition state coverage is as follows:

Definition 5.7. (Fitness Criteria for Transition Coverage)
Let $Design^{A}_{TLM}$ be an AsmL model, $\mathcal{V}_{in}$ its input variables set, A be an assertion, A_{sp} its space state and $\mathcal{T} = \{T^1_{Gen}, T^2_{Gen}, \ldots T^n_{Gen}\}$ be a set of n-test generators. The fitness criteria corresponding to the transition state space coverage is:

$$F_{Tcov} = \max_{T^i_{Gen} \in \mathcal{T}} (T_{cov}) = \max_{T^i_{Gen} \in \mathcal{T}} (\frac{Card(T^i_{tp})}{Card(A_{tp})})$$

where T^i_{sp} is a sequence of test vectors generated by T^i_{Gen}.

The genetic mutation operation updates the set of test generators, $\mathcal{T} = \{T^1_{Gen}, T^2_{Gen}, \ldots T^n_{Gen}\}$, according to the coverage results. We propose to deduce the new population of generators using the following operations: inheritance, mutation and recombination. We keep track of all the populations using a unique sequence Seq_{TGen}.

Definition 5.8. (Generations of Test Generators: Seq_{TGen})
Let $\mathcal{T}_i$ be a set of test generators. Then, the sequence of generations of test generators is:
$Seq_{TGen} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_i, \ldots, \mathcal{T}_m\}$
where:

- $\mathcal{T}_1$ is the initial generation.
- $\forall i \mid 1 < i \leq m$, $\mathcal{T}_i$ is the updated generation obtained from $\mathcal{T}_{i-1}$ by applying inheritance, mutation and recombination operations.

The convergence of the algorithm to the optimal solution w.r.t. the fitness criteria is granted if the sequence Seq_{TGen} is increasing w.r.t. to an order based on the coverage. This could not be derived for a general case. It requires defining: (1) the variables domains; (2) the variables domains distributions; and (3) the inheritance, mutation and recombination operations. In general though, using a simple uniform distribution and preserving the best generator from the previous generation grants that the sequence will not be increasing.

6. Experimental Results

In this section, we illustrate our proposed methodology on the case of a Look-Aside interface standard (LA-1) [8]. This interface is used to interconnect network-processing units (NPUs). Its major features include: (1) concurrent read and write; (2) unidirectional read and write interfaces; (3) single address bus; (4) 18 pin DDR data output path; (5) 18 pin DDR data input path; and (6) Byte write control for writes.

6.1. Coverage Results

In order to evaluate the methodology proposed in this paper, we considered a set of five assertions, Table 1 (Table 2) compares the assertion state coverage (assertion transition coverage) results obtained with:

Assertion	A1	A2	A3	A4	A5
Rand. TLM (%)	10	8	4	12	14
GA TLM (%)	64	72	66	82	55
Guided SystemC RTL (%)	61	71	75	81	56
Rand. SystemC RTL (%)	5	3	6	7	4
Rand. RTL (Specman) (%)	12	11	14	17	9

Table 1. State Space Assertions' Coverage.

Assertion	A1	A2	A3	A4	A5
Rand. TLM (%)	15	17	13	12	15
GA TLM (%)	51	55	52	48	47
Guided SystemC RTL (%)	45	44	42	31	33
Rand. SystemC RTL (%)	4	3	5	6	4
Rand. RTL (Specman) (%)	12	11	8	7	13

Table 2. Transitions Space Assertions' Coverage.

- Blind random test generation of the TLM SystemC code.
- Guided simulation using a test generator obtained after 30 iterations of the GA.
- Guided random test generation of the RTL SystemC code.
- Blind random test generation of the RTL SystemC code.
- Testing the RTL Verilog code using Speman Elite.

We used 10^9 functional calls and 10^9 simulation cycles for the TLM and RTL models, respectively. We iterated the genetic algorithm for 30 generations (each with 10^9 tests). We used a uniform distributions over the variables domains.

6.2. Discussion

A the transaction level, our proposed GA provided an enhanced coverage in comparison to the blind random simulation by a factor of five to seven. The value of the coverage vary according to the assertion. When applying the GA, we noticed that it takes relatively quick progress in the beginning stages of evolution. We noted that there exists some phases, where the algorithm hits local maxima before mutating further, which improves its performance. We even noticed that the coverage sometimes decreases slowly from generation to generation due to the fact that the evaluation of the assertion is based on weighted random generation.

We used the final test generator obtained from the GA procedure at TLM to verify the same assertion at RTL. Coverage results were comparable. Random simulation at RTL provided very low coverage results due to a larger system state space at this level. By defining a suitable environment using the e-language [12], we succeeded to improve the coverage results in comparison to the blind random simulation. Nevertheless, the coverage remained low in comparison to our GA algorithm by a factor of four to five for all the assertions. We also noticed that the execution time using TLM SystemC was very fast (a factor of 50 to 100) in comparison to the simulation of the Verilog implementation with Specman Elite.

7. Conclusion

In this paper, we presented a methodology to enhance assertion coverage using transaction level models as intermediate step in the design process. We used AsmL as a TLM language for the sake of automatically generating an FSM of the system. We defined assertions as a set of states (part of the system's FSM). We introduced two assertion coverage metrics: state and transition coverage. For *connected* FSM, we provided a construction technique for an optimal test sequence that covers the whole assertion's state space (Theorem 4.1); and for *clique* FSM, we generate an optimal test sequence that covers the whole assertion's transition space (Theorem 4.2). In the second part of the paper, we proposed a genetic algorithm to enhance the coverage. An application on the Look-Aside Interface standard, showed an improvement of the assertions coverage by a factor of seven in comparison to blind random simulation and a factor of four in comparison to a guided simulation using Specman Elite of Verisity. As future work, we target to: (1) optimize the genetic algorithm by finding optimal variables domains and distributions over theses domains; and (2) establish a formal link between both the TLM and RTL coverage.

References

[1] P. Godefroid and S. Khurshid. Exploring Very Large State Spaces using Genetic Algorithms. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2002.

[2] Y. Gurevich, B. Rossman, and W. Schulte. Semantic Essence of AsmL. Technical report, Microsoft Research Tech. Report MSR-TR-2004-27, March 2004.

[3] A. Habibi and S. Tahar. Towards an efficient assertion based verification of SystemC designs. In *Proc. High Level Design Validation and Test Workshop*, pages 19–22, Sonoma Valley, CA, USA, November 2004.

[4] A. Habibi and S. Tahar. Design for verification of SystemC transaction level models. In *Proc. Design Automation and Test in Europe*, pages 560–565, Munich, Germany, March 2005.

[5] J. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[6] J. Y. Jou and C. N. Liu. Coverage analysis techniques for hdl design validation. In *Proc. Asia Pacific CHip Design Languages*, pages 48–55, Fukuoka, Japan, October 1999.

[7] Microsoft Corp. AsmL for Microsoft .NET Framework. research.microsoft.com, 2005.

[8] Network Processing Forum. *Look-Aside (LA-1) Interface, Implementation Agreement, Revision 1.1*. Kluwer Academic Publishers, 2004.

[9] Open SystemC Initiative. www.systemc.org, 2005.

[10] H. Rudin. Protocol development success stories: Part i. In *Proc. International Symposium on Protocol Specification, Testing, and Verification*, pages 149–160, Lake Buena Vista, Florida, USA, June 1992.

[11] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *Proc. Conference on Software Maintenance*, pages 302–310, Montréal, Québec, Canada, September 1993.

[12] Verisity Ltd. Website: http://www.verisity.com/, 2005.