Transactional Level Verification and Coverage Metrics by Means of Symbolic Simulation*

Prakash M. Peranandam, Roland J. Weiss, Jürgen Ruf, and Thomas Kropf

Department of Computer Engineering
University of Tübingen
{peranand,weissr,ruf,kropf}@informatik.uni-tuebingen.de

Abstract. Assuring correctness of digital designs is one of the major tasks in the system design flow. Formal methods have been proposed to accompany commonly used simulation approaches. The vital part missing in all these techniques is the so-called coverage of functionalities. This paper aims to tackle this problem by proposing a new method to verify the sequence of transactions among the modules of a system level design.
To accomplish this, we propose a model that captures the transactions of a given high level design. By symbolically simulating this abstract model we can generate the set of all possible transaction sequences of the system design. Verification of properties is carried out on these transactions stepwise. According to the status of the verification results, every transaction will be assigned a special state which will be inherited by its succeeding transactions. Once reaching the end, we calculate the coverage of the design functionalities by the set of properties.
This approach finds at least two interesting applications. First, it can guide the verification engineer during property specification by providing the set of abstract properties that cover the basic functionalities. Secondly, the generated transaction sequences can be exploited to emit monitors for simulation runs of transactional level SystemC designs.

1 Introduction

Gaining confidence on the correctness of digital designs is one of the major tasks in the system design flow. That's the reason why verification is playing a key role in design and development. Present day needs and requirements force the designer to build increasingly complex designs, which requires the designers to climb up the abstraction level ladder. This is because abstraction is a powerful technique for the design and implementation of complex systems[1]. It allows designers to tackle complexity by first hiding unnecessary details and then working on them later. By systematically testing the functionality of the whole system early in the design process, system level design [5] addresses design-process problems. High abstraction level testing or verification enables the designer to find design flaws earlier and at less expense.

Transactional Level Modeling (TLM) [6, 11] is one of such abstraction levels. It is a high level approach to model digital systems where details of communication among modules are separated from the details of the implementation of the functional units or the communication architecture. At the transactional level, the emphasis is more on the functionality of the data transfers, i.e. what datas are transferred to and from what locations, and less on their actual implementation. This approach makes it easier for the system level designer to verify the design against specification requirements.

In general, because the field of verification has become the major part in system design with around 70% of development time, a number of possibilities and techniques to make the process efficient at various levels of design abstraction have been developed. Though there are quite a

* This work is sponsored by the German Research Grant (DFG-Projects KOMFORT and GRASP)

[1] Systems in our context are hardware systems or embedded hardware/software systems such as bus arbiters, automotive controllers or microprocessors.

number of appealing approaches to solve the verification problem, all of these mostly lack one vital part, namely *coverage* [9, 2].

Coverage metrics can be defined as *What percentage of system functionalities are captured by the set of all specified properties*. This is an important characteristic of the verification process, as most of the commercial and academic tools just deliver whether a property holds or not, and produce a counterexample in the negative case. The problem is that the verification engineer will never be sure whether the set of properties specified covers all system functionalities.

We propose a new idea that solves the problem at a higher level of abstraction, namely TLM. Because of its regular structure of interfaces between the modules, it is usually possible to argue about *transactions* between modules. The idea is that there are only a few permitted sequences of message transactions and only the data change in every instance of the permitted sequence. The catch point of our idea is as follows: *at TLM abstraction level, the set of all functionalities directly corresponds to the set of all transaction sequences*. In other words, every transaction sequence encapsulates one functionality of the design. By verifying this set of transaction sequences against the set of properties specified by the verification engineer it is possible to compute the coverage. To achieve this idea of transaction sequence verification, we start with a much more abstract model called *Message Sequence Model* (MSM). For the rest of the paper transaction sequences and message sequences mean the same.

Keating and Bricaud [10] clarify the verification strategy of transactional and block level verification. Transactional level verification enhances the opportunity to find conceptual and functionality errors in the design. In this paper we deal with verification of TLMs by means of symbolic simulation of MSMs. A MSM is an abstraction of a TLM in the sense that it just models the transactions and not the implementation of the modules. In other words, all the modules are treated as black boxes, only the transactions between modules are modeled. This is of interest because we can symbolically simulate the MSM to get the transaction sequences. We use BDD based symbolic simulation [1]. The verification and generation of the set of all properties are explained section wise. Section 2 and 3 discuss the MSM and its relation to TLM. Section 4 presents the property specification, sections 5 and 6 address the property verification algorithm and coverage metrics, respectively. Finally, section 7 concludes and outlines future work.

2 Message Sequences

Before proceeding, let us define the central terms used in this paper and their intended meaning.

Functionality: The expected or intended behavior of a system that is designed for.
Module: A functional unit in a system design. Often referred to as component.
Message: Data or information that is transferred between modules.
Transaction: Communication between two modules by sending a message. The terms message and transaction can be used interchangeably, although they have slightly different meanings. However, in TLM it is common practice to identify a transaction with the message it transfers.
Transaction Sequence: A finite trace or serial order of transactions that follow in logical order or a recurrent pattern that is given by the model. Also called message sequence.
Function Call: In SystemC, an interface function call realizes a transaction.

Coverage: The number of functionalities of the design that are verified or captured by the properties.

Transactional level modeling [6, 11] in SystemC is a high level approach to modeling digital systems with emphasis on communication of modules or components within the system. Communication mechanisms such as busses or FIFOs are modeled as channels, and are presented to modules using interfaces. Due to its structural regularities of interfaces between the modules, it is usually possible to argue about *transactions* between modules. Transaction requests take place by calling interface functions of these channel models. The key point of this level is to give importance to the functionality of the data transfers. Data can be structured into messages. We list the main characteristics of messages, thereby giving an informal definition.

- Every message has a unique name, a sender module, a receiver module and some information[2].
- No assumptions are made about the mechanism of message delivery except that it is a lossless, order-preserving channel.
- Every message occurs only after all its preceding messages have already occurred in the run so far.
- Message transfers will be modeled in SystemC as an interface function call with message name as function name and other details as parameters of the function.

According to the above informal definition, a simulation run of a SystemC TLM [6] is a sequence of function calls. Also note that every sequence starts with one of the primary input messages and ends with one of the primary output messages of the design because they are the first and last possible messages that can be transacted. Every sequence accomplishes the intended functionality of the design in an abstract sense.

The above mentioned informal definition of message transactions reveals that one complete cycle of simulation of a TLM generates a sequence of finite number of message transactions and moreover this sequence will be one of the permitted sequences. Every such sequence corresponds to a functionality of that design. This is equivalent to the argument that the set of all sequences of the model is nothing but the set of all functionalities of the design. The above idea of message transaction sequence will answer at least two of the missing pieces that are necessary for an effective verification of SystemC transaction based models according to [8]. The above mentioned two missing pieces are:

- Creation of an event and transaction database for effective verification, debugging and functionality coverage analysis.
- Detection of illegal behaviors or transactions.

Let us consider a small example of a vending machine (see figure 1) to explain and support the above arguments. The functionalities of the vending machine design are:

1. To get a coffee, the user has to press a coffee button and then has to insert the money, or vice versa.
2. To get a tea, the user has to press a tea button and then has to insert the money, or vice versa.
3. If the user inserts the money first, and then presses the cancel button, the machine has to give back the money. Only this ordering of transactions is possible.

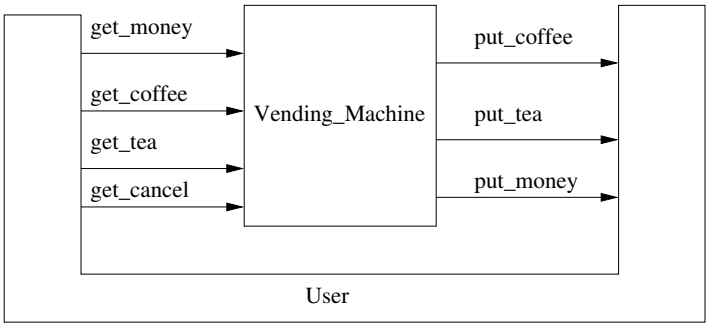

Fig. 1. Transactional level model of a vending machine

As shown in figure 1, the design has only one main module with four input and three output channels. The *user* is treated as a testbench module, which handles the transfer of messages to and from the vending machine. The transactions of input information of pressing a coffee button, tea button or dropping the money is performed by the function calls *get_coffee, get_tea* and *get_money*, respectively. Correspondingly, the output information of coffee and tea is performed by function calls *put_coffee* and *put_tea*, respectively. Similarly, function call *get_cancel* stands for the input information of pressing the cancel button and *put_money* for output information of giving the money back. The above argument applies if every transaction is communicated by separate channels. In contrast, all the input and output messages can also be transferred in a common channel. In this case the above messages will be passed as parameters using a general function *send*. Independent of single or multi channel message passing, the message transactions that occur during the simulation run can be registered using the SystemC verification library. The below sequences show how the registered message sequence of a simulation run looks like.

- The simulation run of the vending machine model for getting a coffee will end with the following sequence of function calls in the channel that connects the two modules (user and vending machine). In simulation sequence (seq 1) the user begins by pressing a coffee button first.

  get_coffee, get_money, put_coffee (seq 1)

  In the next simulation sequence (seq 2), the user begins by inserting the money first.

  get_money, get_coffee, put_coffee (seq 2)

  Both of the above sequences (seq 1) (seq 2) together are covering functionality 1.
- The simulation run of the vending machine model for getting a tea will end with the following sequence of function calls in the channel. In this simulation run the user begins by inserting the money.

  get_money, get_tea, put_tea (seq 3)

  The above sequence (seq 3) is partially covering the functionality 2.

The above explanation describes how the sequences encapsulate the intended functionality of the design. Hence, we are ready to discuss MSM specifications that can be symbolically simulated in order to generate all possible sequences, which in turn are used for the verification process and calculating the functionality coverage.

[2] In our approach, at the transactional level, information transfer is more important than the content of the information.

3 The Message Sequence Model

Essentially, the Message Sequence Model is a basic model of a TLM that captures only the transactions, and the details of transactions are specified by names. This model can also be seen as a special instance of Message Sequence Chart (MSC) [4] which can be symbolically simulated. This model has the information about the sender's module and the function name[3] it is calling, and then the receiver's module and its respective function name called in response to the sender's message. The content of the information is not taken care because we are now more interested in how and where messages are transferred than what is actually transferred. In general, there should be a module serving as testbench that activates the primary input messages and receives the primary output messages. In our example (see figure 1), the *user* is treated as a testbench module, which handles the transfer of messages to and from the vending machine.

The definition of a MSM is as follows:

Definition 1 *A MSM is a quadruple A = (M, C, I, Γ), where $M = \{m_1, \ldots, m_n\}$ is a finite set of message names and $n \in \mathbb{N}$, $C = \{c_1, \ldots, c_k\}$ is a finite set of module names and $k \in \mathbb{N}$, $I \subset M$ is a set of initial or primary input message names, Γ is the set of all transactions, where a transaction T is given as a 6 tuple T = $(C_s, C_r, O, M_s, M_r, \&)$. Where C_s and $C_r \subset$ C are the set of sending and receiving modules respectively. M_s and $M_r \subset$ M are the set of sending and receiving messages respectively. $O = \{\Rightarrow, \rightarrow\}$ is a two element set that is used to specify whether the order is preserved or not. The & symbol is used to syntactically seperate the module and the messages inside the transactions.*

$$c_1 \ \& \ M_s \rightarrow c_2 \ \& \ M_r \qquad \textit{(def 1)}$$

$$c_1 \ \& \ M_s \Rightarrow c_2 \ \& \ M_r \qquad \textit{(def 2)}$$

$$c_1 \ \& \ M_s \rightarrow c_2 \qquad \textit{(def 3)}$$

We write (def 1), (def 2), (def 3) to define the transactions of our model with this semantics:

1. *There exists a transaction between c_1 and c_2.*
2. *$M_s \subset$ M is the set of all messages that can be transacted from c_1 to c_2. Syntactically, this set is represented by conjunction of the elements.*
3. *By (def 1) we do not preserve the order of the elements, all possible combinations of the elements are considered. By (def 2) we do preserve the order of the elements.*
4. *$M_r \subset$ M are activated in response to the messages in M_s. By (def 3) we express that the messages in M_s are the primary output messages. There will be no messages activated in response, i.e. M_r = { }.*

With the above formal definition of a MSM, we will create the MSM of our example. The testbench module is the one which activates the primary input messages and receives the primary output messages. The syntax of the definition part of our example is explained below.

$$\textit{User} \ \& \ \textit{get\_coffee} \ \& \ get\_money \rightarrow \textit{Vending\_Machine} \ \& \ \textit{put\_coffee} \qquad \text{(def 4)}$$

$$\textit{User} \ \& \ get\_tea \ \& \ get\_money \rightarrow \textit{Vending\_Machine} \ \& \ put\_tea \qquad \text{(def 5)}$$

$$\textit{User} \ \& \ get\_money \ \& \ get\_cancel \Rightarrow \textit{Vending\_Machine} \ \& \ put\_money \qquad \text{(def 6)}$$

[3] message name

The above definitions of the MSM state that the sender module *User* is sending messages by sequence of method calls *get_coffee* and then *get_money* to the receiver module *Vending_Machine*. The receiver module *Vending_Machine* in reaction is now ready to send the respective message that is activated by the received messages, i.e. *put_coffee*. The same process holds for the tea, too.

As defined by using $\rightarrow$, we do not enforce the input sequence order, which means for definition (def 4) there will be two possible sequences. Namely, the user can activate the *get_coffee* first and then *get_money* which will end up in sequence (seq 1), or *get_money* first and then the *get_coffee* which will end up in sequence (seq 2). The same holds for definition (def 5). Whereas by using $\Rightarrow$ we do enforce the input sequence as listed in the definition, i.e. definition (def 6) has only one possibility that the user can first insert the money and only then he/she can press the cancel button. The other direction sequence is not applicable.

The other important point in the model is the definition of final or primary output messages. Primary output message always appear at the end of sequences. Their definitions are shown below.

$$\textit{Vending\_Machine} \ \& \ \textit{put\_coffee} \rightarrow \textit{User} \qquad \text{(def 7)}$$

$$\textit{Vending\_Machine} \ \& \ put\_tea \rightarrow \textit{User} \qquad \text{(def 8)}$$

$$\textit{Vending\_Machine} \ \& \ put\_money \rightarrow \textit{User} \qquad \text{(def 9)}$$

Definitions (def 4) through (def 9) constitute the complete model of the vending machine. This MSM can now be symbolically simulated. The process of symbolic simulation and message sequence verification is discussed in section 5. The properties that are to be verified against the design model have a special syntax which is discussed in the following section.

4 Property Specification

Linear-Time Temporal Logic (LTL) [3, 7] is used as property description language. This is the most suitable temporal logic for this approach, as we argue only about the sequence of message transfers in a single path at a time. The properties are specified in LTL, which is always in the form $A \rightarrow B$, in words *if 'A' then 'B'*.

We also allow the time bound enriched LTL called FLTL (Finite LTL)[12], which can be annotated to the temporal operators. The time bound extension is allowed to increase the expressiveness of the properties. For example, time bounded formulas or properties can be used to specify messages at a certain step of the sequence or to specify a message that occurs for a defined number of consecutive steps and so on.

For the rest of the paper, *Vars* = {*set of all declarations of messages in MSM*} is called *variable domain*. LTL formulas are defined recursively over the variable domain. If A and B are (F)LTL formulas, then

Definition 2

$$\phi := v \mid \neg A \mid A \wedge B \mid X\,A \mid X_{[m]}\,A \mid F\,A \mid F_{[m,n]}\,A \mid G\,A \mid G_{[m,n]}\,A$$

is also a (F)LTL formula, where $v \in$ *Vars,* $m \in \mathbb{N}$ *and* $n \in \mathbb{N} \cup \infty$.

$XA \equiv A$ holds at next time step	$X_{[n]}\ A \equiv A$ holds at time step n
$FA \equiv A$ holds at some time step in future	$F_{[m,n]}\ A \equiv A$ holds eventually within time step m and n
$GA \equiv A$ holds at every time step in future	$G_{[m,n]}\ A \equiv A$ holds always from timestep m until time step n

Table 1. LTL & FLTL operators

The temporal operators are defined as in table 1. Following the above definition, a property looks like $A \rightarrow B$, where A & $B \in \phi$. Let us finish this section with example properties of the vending machine.

$$\textit{get\_coffee} \ \wedge\ X\ (get\_money)\ \rightarrow\ X_{[3]}\ (\textit{put\_coffee}) \qquad \text{(prop 1)}$$

$$\textit{get\_coffee} \ \wedge\ X\ (get\_money)\ \rightarrow\ F\ (\textit{put\_coffee}) \qquad \text{(prop 2)}$$

Property (prop 1) expresses that in a sequence if there is a *get_coffee* and at the next step a *get_money* message, then at the third step there should be a *put_coffee* message. By this we are specifying the time step when a message occurs in a sequence, see (seq 1). In contrast, property (prop 2) is more general by stating that if there is a *get_coffee* message succeeded by a *get_money* message, then eventually a *put_coffee* message will occur in the sequence. This property does not argue about the time step when the message occurs.

Hitherto, we have seen how every property argues about occurrence of messages in a sequence. In the next sections we will see how the sequences are verified and coverage is calculated.

5 Verification

Symbolic simulation can be applied in two fundamental ways: breadth first and depth first. Our present approach deals with breadthwise exploration of the search space. This means that we handle one element of all sequences at a time. So these elements will be parent elements for their particular sequences. With these parent elements we compute the child (next) elements of the sequences, which later become parent elements and so on until the end of the sequences.[4]

Basically, every sequence at any point of time will have one of the following four states with respect to every property: *start*, *accept*, *reject* or *pending*. This means that a sequence will have a number of unique states corresponding to every property. To clarify, if there are two properties to be verified, then a sequence will have two states, each corresponding to one property. Explanations of the different states are shown in table 2.

State	**Explanation**
start	Not even the *if* part of the property is satisfied by the sequence.
pending	Only the *if* part of the property is satisfied by the sequence.
accept	The property is fully satisfied (both *if* and *then* part) by the sequence.
reject	The property is not fully satisfied (i.e. only *if* and not *then* part) by the sequence.

Table 2. State explanation

Our aim is to verify the sequence on the fly, i.e. while symbolically simulating. Our algorithm starts with initialization. The initialization process starts with collecting all possible

[4] We use BDD based symbolic simulation. The BDD package used is CUDD from University of Colorado at Boulder.

input messages and storing them in a vector. Every element in this vector is the start of a new sequence. We initialize all the sequences to the *start* state for every property. Then all the start elements of the sequence have to be verified against the set of all properties.

We verify the first element of the every sequence against the set of all properties. Verification here means checking whether the element satisfies the *if* part of the properties. If it is true, we declare the element state to be *pending*, if not we leave the already existing state, i.e. *start* state. If the *if* part of the property consists of temporal operators, then it is handled with some intermediate states, and later gets one of the above mentioned states.

Once initialized, our algorithm starts the routine. Note that the elements of the vector are nothing but one of the message elements of the sequences. The main loop consists of four steps:

1. Symbolically simulate every element and collect all possible next step elements and store them in a new vector.
2. A child element inherits the state status for all properties from its parent element of the sequence.
3. Update the state status for the current elements for all properties. If it is a *pending* state sequence, check for the *then* part of the property. If it is a *start* state, then check for the *if* part of the property. Depending on the satisfiability of the property all elements states are updated. For example, if the inherited state of an element is a *pending* state, and the verification of the property's *then* part is satisfiable then the state will be updated to *accept* state.
4. Replace the current parent elements by the child elements collected in step 1.

These above steps apply until all sequences reach the respective primary output messages. Once we are at end of all sequences, change all *pending* states to *reject* state. This ends the verification process.

6 Coverage Metrics

Once the verification process is completed, we will have all the information needed for coverage [9, 2] computation. Coverage information can be provided in three different flavors. First, we can provide sequence wise property status. Given a sequence, we list the states of all properties with respect to that sequence. Second, we can give a property wise sequence status, i.e. given a property, we list the set of all sequence states with respect to that property. The last coverage information, also the most interesting one, provides the percentage of the total number of functionalities that are covered by the property set. Of course, we can also provide the number of total functionalities and the number of functionalities covered and uncovered. To refresh how functionality and sequence are related see section 2.

Every element of the vector that we obtain from the verification process has a state with respect to every property. This state represents the state of the whole sequence with respect to that property, because every child element in the sequence inherits and updates the state. So the last element of the sequence only represents the updated state of the whole sequence. So the first two flavors of coverage correspond to printing the states of the last elements with respect to every property in two different orders.

The third flavor of coverage comes into focus now. As we saw before in the verification section, every sequence is checked for all properties and will be assigned its respective states. By analyzing the state assigning policy closely, we note that only if a sequence is touched at all

or partially satisfied by any property, then the possible state of that sequence for that property can be either *accept*, *reject*, or *pending*. This means if a sequence satisfies the *if* part of the property then it can never have a *start* state. This makes the argument clear that if a sequence is not satisfying the *if* part of the property then it can only have the *start* state throughout.

The coverage metric k can be calculated by counting the sequences which have only *start* states assigned for all the properties. This count k exactly implies the number of uncovered functionalities. The percentage of the design functionality covered by the set of properties can be computed with count k and the total number of sequences. If count k is greater than zero, then the set of all properties is not complete, and thus needs one or more properties to be added to the property set. Of course, this coverage computation is a trivial one since only finitely many transactional sequences are involved. However this coverage of functionality leads to a complete set of abstract properties. These properties in turn allow us to argue about inputs and their corresponding outputs of every module in the system.

Let us explore this verification and coverage idea with our above vending machine example. We start by specifying the property in order to check whether the design functions properly or not. The main functionality is to get a coffee if we request a coffee, or to get a tea if we request a tea, provided we insert money. Also assume that we start to specify properties with not much information on the design. Then the initial set of properties could be:

$$\textit{get\_coffee} \ \wedge \ X\ (\textit{get\_money}) \ \rightarrow \ F\ (\textit{put\_coffee}) \qquad \text{(prop 3)}$$

$$\textit{get\_tea} \ \wedge \ X\ (\textit{get\_money}) \ \rightarrow \ F\ (\textit{put\_tea}). \qquad \text{(prop 4)}$$

Applying our method on the MSM of the vending machine, the final report will declare that the properties cover only two of the sequences and missed to cover the other three sequences. For explanation let us list all the possible sets of sequences in table 3.

Sequence	1	2	3	4	5
Step 1	*get_coffee*	*get_tea*	*get_money*	*get_money*	*get_money*
Step 2	*get_money*	*get_money*	*get_coffee*	*get_tea*	*get_cancel*
Step 3	*put_coffee*	*put_tea*	*put_coffee*	*put_tea*	*put_money*
(prop 3)	accept	reject	start	start	start
(prop 4)	reject	accept	start	start	start

Table 3. Set of all Sequences

Table 3 shows all possible sequences in different columns. The last two rows show the updated states of the sequences for the properties. Seeing table 3, it is obvious that our properties cover only the sequences in column 1 and 2, leaving the other three sequences untouched. So we need to specify at least three more properties to cover these missed functionalities.

$$\textit{get\_money} \ \wedge \ X\ (\textit{get\_coffee}) \ \rightarrow \ F\ (\textit{put\_coffee}) \qquad \text{(prop 5)}$$

$$\textit{get\_money} \ \wedge \ X\ (\textit{get\_tea}) \ \rightarrow \ F\ (\textit{put\_tea}) \qquad \text{(prop 6)}$$

$$\textit{get\_money} \ \wedge \ X\ (\textit{get\_cancel}) \ \rightarrow \ F\ (\textit{put\_money}) \qquad \text{(prop 7)}$$

After including these properties into the set of already existing properties, the verification algorithm will result in a 100% coverage of functionalities. Of course, we can add more properties to this set which argues about specific details of the sequences, for example (prop 8).

$$\textit{get\_coffee} \ \wedge \ X\ (\textit{get\_money}) \ \rightarrow \ X_{[3]}\ (\textit{put\_coffee}) \qquad \text{(prop 8)}$$

The other experiment that we carried out is the verification of a small part of a wireless protocol. This protocol basically enables the lower module (lower layer in network protocol) to establish a wireless connection by means of message transactions within the network protocol layers. Due to the page limitation we did not include the results.

7 Conclusions and Future Work

This new verification approach is appealing because it results in fast verification of properties on a high abstraction level. Furthermore, it provides information on coverage metrics. The approach is orthogonal to established verification approaches and can complement existing tools.

We see at least two interesting applications of our methodology. First, it can help guiding the verification engineer with the set of abstract properties during verification at a lower level. This means the verification engineer can always check whether the set of properties is complete with respect to the design functionalities. Second, a depthwise algorithm can be exploited as a monitor for SystemC TLM simulation runs, i.e. in SystemC it is possible to store the sequence of message transfers. Having adopted a naming convention for both TLM and MSM, it is trivial to monitor TLM simulations run by this approach.

The list of applications correspond with our goals to be achieved in future. Our immediate work will focus on two main features. First, we want to implement the depth first symbolic simulation and compare it with the breadth first algorithm. Secondly, integration with the SystemC verification library is targeted, so that the simulation kernel can call this application in order to monitor its TLM simulation runs.

References

1. Randal E. Bryant. Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification. *Proc. of Int. Conf. on Computer-Aided Design (ICCAD '95)*, pages 236–243, 1995.
2. Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics for temporal logic model checking. *Lecture Notes in Computer Science, Springer-Verlag Heidelberg*, 2031 / 2001:528–542, 2001.
3. E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In D. Gabbay and H.J. Ohlbach, editors, *Temporal Logic*, Lecture Notes in Artificial Intelligence. Springer-Verlag, July 1994.
4. Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
5. Avi Gal and Stuart McGarrity. Wring Out Better, Faster Results Through System-Level Design. *www.mathworks.com, Design Solutions EDA*, 2003.
6. Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*, volume Chapter 8. Kluwer Academic Publishers, 2002.
7. O. Grumberg. LTL model checking. Tutorial auf der ICTL'94, Computer Science Department Technion, Haifa. Israel, Bonn, Germany, July 1994.
8. C. Norris Ip and Stuart Swan. Using transaction-based verification in systemc. White paper, Cadence Design Systems, Inc., www.SystemC.org, June 2002.
9. Sagi Katz, Orna Grumberg, and Danny Geist. "Have I written enough properties?" A method of comparison between specification and implementation. *Lecture Notes in Computer Science, Springer-Verlag Heidelberg*, 1703 / 1999:280–297, 1999.
10. Michael Keating and Pierre Bricaud. *Reuse Methodology Manual For System-On-A-Chip Designs*, volume Chapter 11. Kluwer Academic Publishers, 2002.
11. Wolfgang Müller, Wolfgang Rosenstiel, and Jürgen Ruf (eds.). *SystemC Methodologies and Applications*. Kluwer Academic Publishers, 2003.
12. J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation based validation of FLTL formulas in executable system descriptions. In R. Seepold, editor, *Forum on Design Languages (FDL 2000)*, pages 311–319, Tübingen, Germany, September 2000. Sig.-VHDL and ECSI.