Implementation of a Transaction Level Assertion Framework in SystemC

Wolfgang Ecker
Infineon Technologies AG
IFAG COM BTS MT SD
81726 Munich, Germany
Wolfgang.Ecker@infineon.com

Volkan Esen,
Thomas Steininger,
Michael Velten
Infineon Technologies AG
TU Darmstadt - MES
Firstname.Lastname@infineon.com

Michael Hull
Infineon Technologies AG
University of Southampton
mh102@ecs.soton.ac.uk

Abstract

Current hardware design and verification methodologies reflect a trend towards abstraction levels higher than RTL, referred to as transaction level (TL). Since transaction level models (TLMs) are used for early prototyping and as reference models for the verification of their RTL representation, the quality assurance of TLMs is vital. Assertion based verification (ABV) of RTL models has improved quality assurance of IP blocks and SoC systems to a great extent. Since mapping of an RTL ABV methodology to TL poses severe problems due to different design paradigms, current ABV approaches need extensions towards TL. In this paper we present a prototype implementation of a TL assertion framework using SystemC which is currently the de facto standard for system modeling.

1 Introduction

The main modeling paradigm in electronic system level (ESL) is transaction level modeling. Due to a reduction of implementation details, transaction level models (TLMs) are used for early system validation and architecture exploration. So-called virtual prototypes are used for early software development and as golden references for the verification of RTL prototypes. Due to this fact, it is vital to ensure the quality of a TLM through elaborated verification methodologies. Current verification methodologies for complete systems consist of a combination of scoreboard and coverage techniques also taking the embedded software into account. Assertion based verification (ABV) has shown a great impact on verification productivity and quality assurance in RTL flows by improving the visibility of internal objects of a design. Hence, leveraging ABV on transaction level is desirable. However, no ABV approach has been established for the transaction level (TL). This is due to the fact that RTL-ABV cannot directly be mapped to TL since the modeling paradigms differ. The main difference is the concept of synchronization. In RTL models it is achieved by the use of clocks that define when state changes can happen. In TL synchronization is obtained by mutual dependencies of transactions and by the use of time annotations in addition to the use of asynchronous communication protocols. Furthermore, the applied synchronization schemes depend on the abstraction layer chosen for a TLM. Since a system representation can consist of TLMs of different abstraction as well as RTL models, an ABV approach has to be chosen that can cope with mixes of abstraction layers. In this paper we gather requirements that we believe are necessary for lifting ABV to TL and introduce a SystemC implementation of an ABV framework which is capable of monitoring properties that reason about both the architecture and the embedded software of a system.

The paper is structured as follows. After discussing related work we explain the required features for transaction level properties. We illustrate the use of these features with an application example and describe the implementation details of the suggested framework followed by experimental results. We close with conclusions and discuss the next steps to be done.

2 Related Work

SystemC, the de facto standard for system level design, does not yet have standard native temporal assertion support. Work has been presented for migrating current RTL-ABV approaches to SystemC as e.g. in [11],[6], and [7]. In contrast to that, the concepts shown in this paper aim at higher levels of abstraction since RTL concepts cannot be mapped directly to TLM. Concurrent assertions on RTL are modeled clock based. The lack of clock synchronous behavior requires more expressive control expressions for the evaluation of concurrent assertions in TLMs.

Work towards formal model checking of system level models exists as shown in [13], [12], [5], and [4]. These approaches rely on state space exploration based on abstract representations of system level models. However, formal methods unfortunately involve state space explosion problems that pose limitations on the tasks at hand. The work presented in this paper rather focuses on dynamic verification approaches using transaction level assertions for simulation rather than for static verification. In [10] and [2] new approaches for transaction level assertions are introduced. However, in [10] transactions are mapped to signals and therefore the approach is restricted only to transactions which are invoked by suspendable processes. Our approach works on the basis of events in contrast to signals. Hence, it is not restricted to a certain kind of transaction. In [2] transactions are recorded and written into a trace to do post processing. Trace based assertion checking however requires that everything to be recorded must be annotated in the code and the creation of simulation data bases can become very resource intensive. Furthermore this approach does not con-

978-3-9810801-2-4/DATE07 © 2007 EDAA

sider start and end of transactions. Therefore overlaps of transactions and parent child relations cannot be detected.

The implementation work presented within this paper is SystemC compliant and hence, other TL verification approaches can be used along. Therefore our assertion approach can complement verification in frameworks like the Advanced Verification Methodolgy [9].

3 Required Features for Transaction Level Assertions

In this section we briefly gather features that enable and ease the monitoring of transaction level properties.

The functionality of a transaction level model can be characterized by the use of transaction sequences as described in [12]. A transaction sequence corresponds to a user specifiable pattern of transactions which occur during a simulation run. Such sequences can be used to build properties about the system behavior.

On RTL, sequences are patterns of boolean propositions observed on signals or registers. The evaluation of a sequence is performed synchronously to the design by using the value change of the design clock or a similar signal as a trigger for the evaluation of boolean propositions. Temporal relations between such propositions are mainly specified in terms of clock cycles. On TL, however, clocks are not modeled in order to reduce the number of delta cycles and events that lead to context switches. Instead the design functionality is modeled in a more abstract way, neglecting micro architectural details. Synchronization is used only at points where the default scheduling of processes would lead to data and control conflicts. Since TL sequences do not necessarily start and end at a synchronization point, we require that the evaluation of sequences is triggered asynchronously, i.e., by using the start and end events of transactions as triggers.

This approach however might lead to starving sequence evaluations since the occurence of awaited events is no longer ensured. Using timeout mechanisms (called timer events) easily resolves this issue.

Another required feature is the notion of positive and negative triggering events. If an event is declared negative, the evaluation of a transaction level sequence returns a negative result. This can be used for terminating a sequence if a transaction occurs that is not supposed to happen during the evaluation.

Further on, combining single events using boolean operators and combining these event expressions with time constraints offers further possibilities when formulating properties.

TL sequences also need to support multi threaded evaluation modes, as e.g. in SystemVerilog Assertions (SVA, [8]).

Finally properties that form an implication require evaluation modes that allow both the overlapping concept of SVA and the “Restart”, “NoRestart” modes known from the open verification library (OVL, [1]).

4 Application Example

In this section we introduce a simple transaction level PVT (Programmer’s View with Timing) model and explain how the required features from Section 3 can be used for specifying properties. The Listings used here show examples of a descriptive assertion language that we developed. Descriptions in this language are used as input for a compiler that generates the SystemC implementation which is discussed here. A detailed introduction of the language can be found in [3].

4.1 CPU-Queue

Figure 1 depicts the application model including proxy monitors used for the detection of start and end of a transaction passing the monitored port.

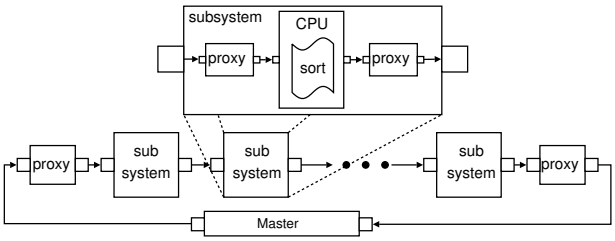

Figure 1. CPU-Queue

The model consists of a queue of 16 subsystems, each including one CPU and I/O ports for data transfers. The communication between a CPU and its I/O devices utilizes blocking transactions for reading and writing the peripherals. The CPU blocks when the addressed device is not ready for that access. The input port of the first subsystem is accessed from the outer driving module. The output port of the last subsystem is connected to the outer module’s input port.

The software running on the CPUs implements a distributed algorithm for sorting non zero values. At first the number of data values to be sorted is read in and then passed on to the next subsystem. After the first sorted value has propagated through the queue, the remaining values have to follow with exactly 10 time steps distance.

Further details of the application model are not relevant for the remainder of this paper.

4.2 Assertions for the Application Model

Many properties can be specified for this simple system. We will focus on two properties we want to monitor:

- prop-SortVal-pv: Each subsystem propagates either the value at the input or the internally stored value to the output, depending on which is greater; the other one is stored.
- prop-17in17out-pvt: Pushing 17 values in the queue implies 17 values at the output of the queue where the first value pushed in equals the first value at the output. Additionally, the last 16 values have to have a temporal distance of 10 time steps to each other.

The first property does not involve timing, whereas the second property requires further timing information.

Property “prop-SortVal-pv” has to hold 15 times for each instantiated subsystem. Listing 1 shows a sample notation of this property. The antecedent sequence matches whenever *CPU_read_end* event occurs after which the CPU’s *R1* register does not equal zero. The delay element (#) delays the evaluation of the boolean condition by one *CPU_read_end* event. The antecedent also contains local variables *L1* and

L2 which store the two values to be compared by the program for a later point in the evaluation.

The consequent sequence is only evaluated after a match of the antecedent and is triggered by the end of a write transaction invoked by the CPU.

```
property prop-SortVal-pv(AnyMatch, ReportOnRestart)
 sc_uint<12> L1, L2;
 #1(CPU_read_end) (CPU.R1 != 0, L1 = CPU.R1, L2 = CPU.R0)
 |->
 #1(CPU_write_end) ((L1 > L2) ? CPU_write.data == L1
                              : CPU_write.data == L2);
endproperty
```

Listing 1. Property: "prop-SortVal-pv"

It matches when the right value has been propagated out using the write transaction. The identifier *CPU_write.data* is a reference to the monitor that detects write transactions and stores their payload in a member variable called *data*.

The evaluation mode for the antecedent is "AnyMatch"[1] since the sequence has to detect any completed read transaction. The evaluation mode of the property is "ReportOnRestart"[2] for the occurrence of a further read transaction prior to a write transaction is illegal behavior.

The second property is formulated in Listing 2.

```
property prop-17in17out-pvt(AnyMatch, ReportOnRestart)
  sc_uint<12> L1;
  #1(b_put_in_st) (true, L1 = b_put_in_st.data)
  #16(b_put_in_st) true
  |->
  #1(b_put_out_end) (L1 == b_put_out_end.data)
  #1(b_put_out_end) true
  #15(b_put_out_end@10; timer(11)) true;
endproperty
```

Listing 2. Property: "prop-17in17out-pvt"

The identifier *b_put_in_st* refers to the start of the blocking transaction *put_in* that drives a value into the queue. The identifier *b_put_out_end* denotes the end of the transaction *put_out* that propagates a value out of the queue.

For the antecedent we chose the start event of *put_in* as a trigger because the blocking mechanism allows that the first value propagates out of the queue while the last value is driven in. The first delay operator is for catching the first occurrence of *put_in* in order to store its payload in the local variable *L1*. After 16 further occurrences of *put_in* the antecedent produces a match which starts the implication. Note that the property mode is specified as "ReportOnRestart". Since several evaluation attempts of a sequence may overlap, new evaluation attempts are created with every occurrence of *put_in*. By choosing this property mode we check that there is no further *put_in* transaction while the consequent is under evaluation.

The first delay operator of the consequent sequence catches the first *put_out* transaction in order to compare its payload with the first value being sent into the queue. The second delay operator matches the second *put_out* transaction and the third delay operator matches the remaining *put_out* transactions, while checking that they occur exactly in a 10 time steps distance to each other. The delay operator in the middle is necessary since the temporal delay between the first and the second *put_out* transaction is not specified. The last delay operator utilizes a time constraint operator (`@10`) to ensure that the corresponding event triggers only if it occurs 10 time steps after the last triggering event of this sequence. The timer operator is applied in order to ensure that the sequence will produce a result. Note that within this delay operator's trigger list, we utilize a semicolon to separate positive (left) from negative (right) triggers. Therefore in case the timer occurs the sequence will produce a "not matched" result.

[1]The default matching strategy in SVA for sequences that are not used as consequent expressions

[2]This OVL mode throws a report if the antecedent has matched while an implication was being active.

5 Implementation

In this section we give a detailed overview of a SystemC framework that supports the specification of transaction level assertions.

5.1 Monitor

In order to detect transactions we used proxy monitors which are linked in between two communicating modules. The monitor inserts call backs at the beginning and at the end of a transaction. This is similar to an instrumentation for transaction recording using SCV (SystemC Verification Library). Note that the monitor does not utilize *sc_events*. Such events interact with the scheduling algorithm of the SystemC kernel; however, throwing events does not suspend the monitored transaction and the process that called it. Hence, the corresponding transaction would have to reach a synchronization point to invoke a context switch in the kernel which then takes the notified events into account for further scheduling. Since a transaction can be "non-blocking", the information of the start and end has to be propagated to an assertion immediately through callbacks, i.e., child transactions of the monitored transaction. The monitor also stores the values transported by a transaction. These values are not passed to the assertions, but are accessible through cross module references. Therefore the values are only read when required by an evaluation of an assertion.

5.2 Structure

Since SystemC supports object oriented design it is possible to model all functionality as classes or modules. Such a modular concept allows an easy assembly of any sequence, property, and assertion. Our implementation is divided into three hierarchical layers:

- Verification Layer: Specifies which properties are to be asserted or covered
- Property Layer: Specifies properties as either a combination of simpler properties using property operators or as an implication built up from several sequences
- Sequence Layer: Specifies sequences as either a combination of simpler sequences using sequence operators or as basic boolean expressions correlated by events

Figure 2 demonstrates the interaction between the different layers, as well as how the different components are connected together. The "B L" objects are modules that perform the evaluation of boolean conditions and handle local variables. These modules are skipped in the remaining sections

since they do not introduce TL specific features. For details concerning the other modules see the next sections.

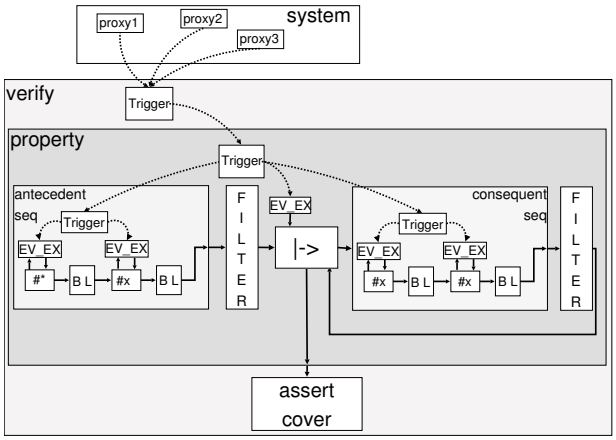

Figure 2. Structural Overview

5.3 Sequences

This section describes the basic building blocks of a sequence. A sequence is constructed of delay operators, event expressions, and boolean expressions. Before explaining the blocks in detail we explain how the evaluation is handled in general.

5.3.1 Token Based Evaluation

The key idea in our implementation is the fact that all objects perform their operations on dynamic data structures. One evaluation thread of a sequence is modeled with a token that propagates through all elements of a sequence[3]. The solid arrows in Figure 2 denote the path of tokens through an assertion. A token object contains the following entries:

ThreadID	Global identifier per evaluation attempt
SubthreadID	Identifier per branch of an evaluation attempt; the evaluation is split into several branches called subthreads in case a sequence contains delay ranges (e.g., #[1:3](...))
IDSpace	Specifies maximum number of Subthreads and is used for calculating SubthreadIDs dynamically within a sequence evaluation
Creation Time	Stores at which simulation time a thread has started
Result	The result of the evaluation attempt that is represented by ThreadID
LocalVars	Declaration of local variables

5.3.2 The Delay Operator

Figure 3 depicts the general structure of a delay operator. A token is put in a delay operator using its input port "start_p". At first the token is stored in a "deque" object, which preserves the order of arriving tokens. In the second step the token is combined with an initial counter value and sent to the event expression logic via the event control port. When a token arrives at the event expression logic it enables the logic. The token arrives back at the delay operator when

[3]Where necessary the token object is stored in dynamic lists.

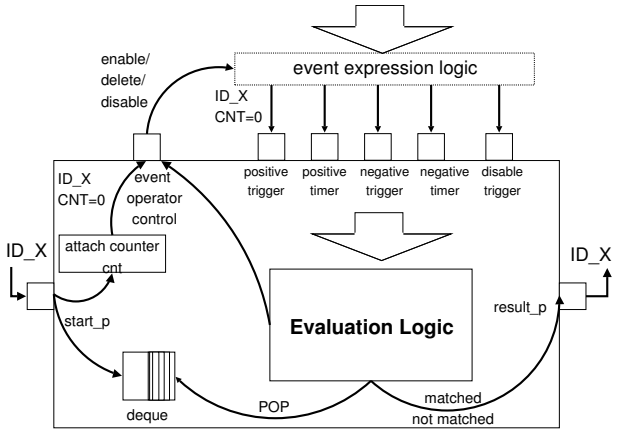

Figure 3. Delay Module

the event expression logic produces a result. Depending on the result the token arrives at the corresponding input port of the delay operator; i.e., if a negative timer occurs the token arrives via the negative timer port of the delay operator and invokes a corresponding method in the evaluation logic. When a token arrives back at the delay operator this token and its counter value is deleted from the event expression logic. Then the counter of the token is incremented in the evaluation logic in order to indicate that the token has been delayed one step. If the token has been delayed the desired amount of steps a positive result is produced for this token and sent via the result port and the token is deleted from the input list; otherwise the token is sent back to the event expression logic. If the token arrives at the negative ports a negative result for this token is produced and sent via the result port and deleted from the input list. The disable input is used for canceling all pending evaluations.

The first delay element in a property has to be a specialized version of the one depicted in Figure 3, since the first element has to create the tokens for the further evaluation.

5.3.3 Event Expression Operators

For building event expressions we implemented special event operators "and" and "or". Both operators send the corresponding tokens out via a call on their result port if the operator evaluates to true. The "and" operator produces a call when its two operands have occurred within the same simulation time slot. The "or" operator produces a call as soon as one of its operands occurs.

Additionally we implemented a time constraint operator. This operator has only one operand. It requires that the operand occurs within the specified time interval. The time interval is relative to the point in time when the operator is enabled by a token object. All arriving tokens are stored in an associative object and linked to their corresponding time of arrival. When the operand occurs, all associations are checked for their time stamp value. All associations with a time stamp that is already out of range are cleared. The associations where the time stamp is in the specified range are cleared as well, however, the tokens associated with that time stamp are sent out via the output port in the same order they had arrived.

5.3.4 Timer Operator

This operator has no operands. It is enabled with each arriving token. An event in an "sc_event_queue" is scheduled for the specified time when a token arrives; the token is then stored in a list along with a boolean value that signals whether there was a delete attempt for this token[4] or not. When the scheduled event arrives, this boolean value is checked. If there was no delete attempt for this token, the operator produces a call on its output port.

5.3.5 Splitter and Merger

If ranges of delays are required, one evaluation thread is split into parallel subthreads by explicitly unrolling the ranges. Hence, for each step in a range a delay is instantiated with an exact delay value. A range of 1 to 3 steps for instance is unrolled to three delay operators in parallel with delay amounts 1, 2, and 3, respectively. Everything following a delay operator is instantiated three times as well, and so forth. Since the evaluation of subthreads has to happen in parallel, the token has to be copied for each subthread with a unique subthread ID. This is accomplished by a splitter module. The splitter module has one input and the required amount of outputs. The splitter computes the new subthread IDs by analyzing the available space, which is stored in the token. Each new token has a new entry for its subthread ID and its available ID space.

At the end of a sequence the results of all parallel instantiations have to be merged again such that the outgoing tokens can be streamed through one port.

5.3.6 Sequence Evaluation Modes

In SVA two sequence evaluation modes are defined:

AnyMatch	Every subthread producing a match results in a corresponding match for the whole sequence.
FirstMatch	The first subthread producing a match results in a match for the whole sequence. All remaining subthreads are ignored.

These evaluation modes do not affect the basic structure of the sequence, however they determine how to interpret the tokens which are propagating out of a sequence. Thus, a module is connected to the end of a sequence that filters out irrelevant tokens.

For "AnyMatch" mode the filter basically does nothing, whilst in "FirstMatch" mode the filter utilizes a hash object[5] for storing the first matching token of one Thread ID by using the ID value as the key. Hence, the filter ignores all tokens belonging to one thread ID as long as this ID exists as a key in the hash object. If the token of the last possible subthread arrives, the filter checks for the key in the hash object. If the key exists it is deleted. If it does not exist the result indicated for that token is returned.

5.4 Properties

This section describes the basic building blocks of a property. A property is constructed of an implication operator and event expressions.

[4] A delete attempt is sent by the delay operator as soon as there is a legal trigger event.

[5] We used a map from the Standard Template Library.

5.4.1 The Implication Operator

Figure 4 depicts the general structure of an implication operator. A token arrives through the port "ante_p" and is

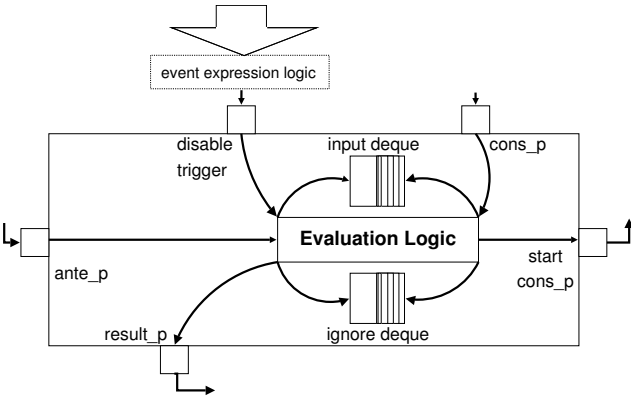

Figure 4. Implication Module

passed to the evaluation logic. The evaluation logic determines when to start the consequent depending on the chosen evaluation mode (see next section). The consequent is enabled by passing the token to the port "start_cons_p". If an evaluation is started the antecedent token is also stored in the input list. A successful completion of the consequent sequence is signaled via the "cons_p" port. If the implication is fulfilled the token is finally passed to the "result_p" port and deleted from the input list. The ignore list is only used for some modes (details see below). Again the disable input is used for canceling all pending evaluations.

5.4.2 Property Evaluation Modes

In OVL three property evaluation modes are defined while SVA uses a fourth one:

Restart	Every match of the antecedent restarts the evaluation of the implication.
NoRestart	Any further match of the antecedent while an evaluation is active are ignored.
ReportOnRestart	Any further matches of the antecedent while an evaluation is active are reported.
Overlap	Any further matches of the antecedent while an evaluation is active start further evaluation threads.

These evaluation modes do not affect the basic structure of the property, however they determine how to interpret the evaluation results of antecedents and consequents.

In Restart mode a match of the antecedent during an active evaluation results in copying the input list to the ignore list and clearing the input list afterwards. As long as the ignore list contains data all matches of the consequent lead to a step by step reduction of the ignore list entries. Thus, consequent evaluations matching the tokens in the ignore list do not affect the result of the implication.

In NoRestart mode further matches of the antecedent are ignored as long as the input list is not empty.

In ReportOnRestart mode the behavior is similar except for the fact that the user is notified of the restart attempt.

In Overlap mode every match of the antecedent is stored in the input list and the consequent is started immediately.

A successful completion of the consequent removes the topmost token from the input list.

5.5 Verification Layer

This layer always associates properties with corresponding verification directives. A verification directive can be parameterized by the severity level and an info string; further on it can be specified if the property should be asserted, covered or both.

6 Experimental Results

In this section we present some experimental results that we obtained using several different application scenarios:

- Single CPU: In combination with a multiplication program several properties for the correct execution of the most frequently executed instructions were used.
- CPU Queue: The system was built from 16 and 32 subsystems respectively; the properties for sorting data as presented in the previous chapters had to be adapted for the larger example.

We measured the necessary execution time for all examples both with and without assertions on a single CPU system with 1.6 GHz and 1 GB RAM running a Linux OS. We used SystemC 2.1.v1 in combination with the GNU C Compiler (gcc) version 3.4.4 with optimization -O3. In order to lessen the impact of workload or similar effects all tests were run several times and afterwards the average time was computed.

All tests were run several times with varying iteration depths. The results of the evaluations with iteration depths of 1000, 2000, and 3000 are presented in Table 1. The first line shows the execution time in seconds per CPU without and with assertions, while the second line holds the total number of successful property evaluations per CPU for the given test case; this number represents the overall activity of all assertions.

	1000	**2000**	**3000**
SORT16	0.2 / 0.5	0.4 / 1.0	0.6 / 1.6
	15078	30141	45203
SORT32	0.4 / 1.4	0.8 / 2.8	1.1 / 4.2
	31063	62094	93125
CPU	0.3 / 3.5	0.6 / 7.0	0.9 / 10.5
	129000	254000	381000

Table 1. Experimental Results

As can be seen there are big differences concerning the execution times, especially for the single CPU example. On the other hand, the properties show a much higher activity compared to the other systems.

The relative performance impact of the assertions remains almost constant for each example and is thus nearly independent from the iteration depth.

Even though the performance loss due to the inclusion of the assertions can become quite large, this does not invalidate our approach. Since the implementation presented here represents mostly a proof of concept, no performance optimizations were included within the assertion framework, yet.

7 Conclusion and Outlook

The discussions in the previous sections show that applying ABV to TLMs offers new ways of high level system verification. We have gathered requirements for TL-ABV where we introduced transaction events (START,END) for synchronizing the concurrent evaluation of assertions. Further on we abstracted time by using timer events and time constraints. Additionally we also combined evaluation modes which are distributed over several ABV approaches into one concept. An application example was given in order to demonstrate a couple of TL properties. Afterwards we provided details about a possible implementation of the different operators based on SystemC classes and the integration of these operators in order to build up the complete assertion monitor.

Further additions have to be developed that allow e.g. combinations of sequences or properties using corresponding operators. In addition to that we work on a fully pipelined evaluation mode that allows exact matches in sequences.

Acknowledgment

This work is partially funded by the European Commission under SPRINT IST-2004-027580.

8 References

[1] Accellera. *Open Verification Library*. http://www.accellera.org/activities/ovl/.

[2] X. Chen, Y. Luo, H. Hsieh, L. Bhuyan, and F. Balarin. Assertion Based Verification and Analysis of Network Processor Architectures. *Design Automation for Embedded Systems*, 2004.

[3] W. Ecker, V. Esen, M. Hull, T. Steininger, and M. Velten. A Prototypic Language for Transaction Level Assertions. In *Design and Verification Conference (DVCon)*, February 2007.

[4] D. Große and R. Drechsler. Formal Verification of LTL Formulas For SystemC Designs. In *International Symposium on Circuits and Systems*, volume 5, pages 245–248, May 2003.

[5] A. Habibi and S. Tahar. Assertion and Model Checking of SystemC. In *North American SystemC Users Group Meeting*, San Diego, California, USA, June 2004.

[6] A. Habibi and S. Tahar. On the extension of SystemC by SystemVerilog Assertions. In *Canadian Conference on Electrical & Computer Engineering*, volume 4, pages 1869–1872, Niagara Falls, Ontario, Canada, May 2004.

[7] A. Habibi and S. Tahar. Towards an Efficient Assertion Based Verification of SystemC Designs. In *In Proc. of the High Level Design Validation and Test Workshop*, pages 19–22, Sonoma Valley, California, USA, November 2004.

[8] IEEE Computer Society. *SystemVerilog LRM P1800*. http://www.ieee.org.

[9] Mentor Graphics. *Advanced Verification Methodology Cookbook*. Mentor Graphics, http://www.mentor.com, 2006.

[10] B. Niemann and C. Haubelt. Assertion Based Verification of Transaction Level Models. In *ITG/GI/GMM Workshop*, volume 9, pages 232–236, Dresden, Germany, February 2006.

[11] T. Peng and B. Baruah. Using Assertion-based Verification Classes with SystemC Verification Library. *Synopsys Users Group, Boston*, 2003.

[12] P. Peranandam, R. Weiss, J. Ruf., and T. Kropf. Transactional Level Verification and Coverage Metrics by Means of Symbolic Simulation. In *ITG/GI/GMM Workshop*, February 2004.

[13] R. J. Weiss, J. Ruf, T. Kropf, and W. Rosenstiel. Efficient and Customizable Integration of Temporal Properties into SystemC. Lausanne, Switzerland, September 2005.