REQUIREMENTS AND CONCEPTS FOR TRANSACTION LEVEL ASSERTION REFINEMENT

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

Abstract: Both hardware design and verification methodologies show a trend towards abstraction levels higher than RTL, referred to as transaction level (TL). Transaction level models (TLMs) are mostly used for early prototyping and as reference models for the verification of the derived RTL designs. Assertion based verification (ABV), a well known methodology for RTL models, has started to be applied on TL as well. The reuse of existing TL assertions for RTL and/or mixed level designs will especially aid in ensuring the functional equivalence of a reference TLM and the corresponding RTL design. Since the underlying synchronization paradigms of TL and RTL differ - transaction events for TL, clock signals for RTL - a direct reuse of these assertions is not possible. Currently there is no established methodology for refining the abstraction of assertions from TL towards RTL. In this paper we discuss the problems arising when refining TL assertions towards RTL, and derive basic requirements for a systematic refinement methodology. Building on top of an existing assertion language, we discuss some additional features for the refinement process, as well as some examples to clarify the steps involved.

Keywords: ABV, Mixed-level Assertions, TL Assertions, Assertion Refinement

1. INTRODUCTION

Early prototyping with the usage of higher abstraction levels than RTL has become increasingly popular during recent years [4] and is used more and more in industrial workflows. The most common modeling paradigm in this regard

Ecker, W., Esen, V., Steininger, T., Velten, M, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 1–14.

is transaction level modeling. Transaction Level models (TLMs) are used for architecture exploration, as early platforms for software development, and later on as golden reference models for verification of the corresponding RTL designs. Besides, using so called transactors which translate TL protocols to RTL and back, an existing TL model can also be used for testing an RTL component in a TL environment without the need for the whole system to be implemented in RTL already. First steps have been taken to apply Assertion Based Verification (ABV), which has been successfully used for RTL verification for years, to TLMs as well. Some of these attempts try to enhance existing approaches like SystemVerilog Assertions (SVA) [1][11] or the Property Specification Language (PSL) [8] in order to support TL designs and paradigms. In order to really use TLMs as golden reference, a full equivalence check between TLM and RTL would be desirable. One possibility to enhance the current state of the art would be to reuse existing TL assertions for the RTL design or mixed level designs. The main problem for this reuse attempt is based on the totally different synchronization methods in TLM and RTL. On RTL all synchronization is based on dedicated clock signals. In TL it is obtained by mutual dependencies of transactions and potentially by the use of time annotations in addition to the use of non-periodic trigger signals. Furthermore, the applied synchronization schemes differ on the various TL sublevels. Since a reuse of TL assertions for RTL and especially mixed level assertions has to support all synchronization schemes involved, we chose to develop our own assertion language [6][7]. As with every refinement process, e.g. synthesis, this assertion refinement requires additional information and thus can never be fully automated. A partial automation is possible by providing refinement related information upfront in order to avoid the necessity for user interaction. This automated refinement decreases time for rewriting assertions and guarantees a higher degree of consistency. In this paper we discuss which additional information is necessary for the refinement and how the process could be simplified and automated.

The paper is structured as follows. After discussing related work we give an overview of the used assertion language. Afterwards we discuss some requirements and useful features for the mixed level assertions followed by a small example. As a next step we describe a methodical refinement process from TL to RTL. We illustrate the assertion refinement by an application example which also demonstrates the usefulness of the proposed features. After a summary we give an outline of ongoing work towards packaging of assertions in a SPIRIT conformal way.

2. RELATED WORK

The application of assertion based verification to TLMs is a relatively new development. Work has been presented for migrating current RTL-ABV ap-

proaches to SystemC - the industry standard for TL modeling - as e.g. in [14], [9], and [10]. These approaches show the problem that RTL concepts cannot be directly mapped to TL, since the main synchronization mechanism for RTL designs, i.e. clock signals, normally does not exist in most TLM designs; some designs do not model timing at all. This restricts the application to the less abstract sublevels of TL or at least requires a lot more effort. In [13] a new approach for real transaction level assertions is introduced. However, transactions are mapped to signals and therefore the approach is restricted to transactions invoked by suspendable processes. Another approach is presented in [5]. Here, transactions are recorded and written into a trace in order to do post processing. As a disadvantage this approach - as every kind of trace based assertion checking - 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 consider start and end of transactions. Therefore overlaps of transactions and parent child relations cannot be detected.

In [2] an approach for TL assertions is presented. In [3] it is shown how to apply these assertions directly to an RTL design using transactors. This approach however only allows the verification of transactions within the system while a reference to current design states is not possible. Besides, it is not meant for real mixed level assertions, since the presented assertions are all applied on the TLM side of the transactor.

In [6][7] we presented an approach working on the basis of events which covers all sublevels of TL - i.e., programmer's view (PV), programmer's view with timing (PVT), and cycle accurate (CA) - as well as RTL. We developed this language since we needed a support of both TL and RTL features as well as functionality for mixed level assertions which was not possible to the required extent with any of the existing assertion languages.

The term "Assertion Re nement" is already used in another context. In [12] an approach is presented for feeding information from the verification process back into the assertions in order to make them more suitable for the given task. Our approach for assertion refinement on the other hand deals with a transformation from TL assertions to RTL while keeping the logical relations inherent in these assertions intact.

3. TL ASSERTION LANGUAGE

In this section we give an overview of the assertion language introduced in [6][7] which provides full support for both transaction level and RTL. We show some examples of the language features.

TL models work on the basis of transactions and events. An event happens in zero time - though it might use delta delays - and can be used for triggering

threads. The language works on an event driven basis and assumes that both start and end of a transaction produce a definite event that can be used for triggering evaluation attempts of verification directives. Different levels of abstraction pose different requirements concerning the handling of events. Table 1 shows an overview of event operators and functions as well as the TL sublevels where they can be applied.

Symbol	Definition	Level
e1 | *e2*	produces an event if *e1* or *e2* occurs;	PV, PVT, CA
e1 **&** *e2*	produces an event if *e1* and *e2* occur in the same time slot;	PVT, CA
ev_expr@(*bool_expr*)	produces an event if *bool_expr* evaluates to true when *ev_expr* occurs	PV, PVT, CA
timer(*n*)	produces an event at the specified time value; **timer**(*n*) $\rightarrow$ event scheduled *n* time steps later than the current evaluation point;	PVT, CA
$delta_t	time to last event; can be used in expressions (**$delta_t** == 20)	PV, PVT, CA
signal'**POS**	produces an event on a positive edge of the specified signal	CA
signal'**NEG**	produces an event on a negative edge of the specified signal	CA
last_event(*event*)	evaluates to *true* if the last trigger event equals the specified event	PV, PVT, CA

Table 1. Event Operators and Functions

One of the key features of the assertion language is a general delay operator which works independently from the abstraction layer and thus allows the specification of sequences across abstraction levels. Figure 1 depicts the overall structure and functionality of this delay operator.

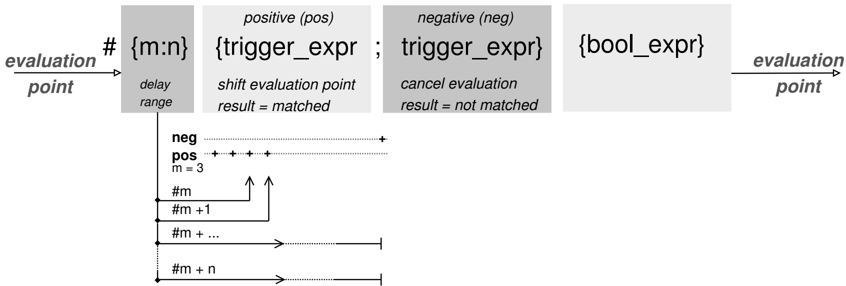

Figure 1. General Delay Operator

One evaluation attempt of a sequence built from this delay operator is called a thread[1]. The specification of a delay range leads to a split into several so-called subthreads.

Listing 1.1 shows a sample use of this delay operator. The first set of curly brackets specifies the delay range, i.e. the required number of occurrences of a trigger expression. The trigger expressions are specified within the second set of curly brackets - divided into positive triggers in front of the semicolon which shift the evaluation and negative triggers after the semicolon which stop the evaluation - while the third set of curly brackets contains the Boolean propositions to be checked at the time of the trigger.

```
#{3:5}{(e1|e2)@($delta_t >=45 && $delta_t <=50); e3, timer(51)}{A == B};
```

Listing 1.1. Sample Event Sequence

This configuration delays the evaluation until *e1* or *e2* have occurred between three and five times with a temporal distance of 45 to 50 time steps. If the positive trigger occurs three to five times, the Boolean expression ($A == B$) to the right is evaluated. If this expression evaluates to “true” the delay operator results in a “match”. The delay operator results in “not matched” if either *e3* occurs, or the evaluation per delay step takes 51 time steps, or the Boolean expression evaluates to “false”.

4. MIXED LEVEL ASSERTIONS

Mixed level assertions can be a great help when trying to verify an RTL component within an existing TL environment or also for checking a transactor for correct behavior.

In this section we gather some requirements for mixed level assertions and show possible applications.

Requirements

Any assertion language supporting mixed levels has to support the underlying semantics of both RTL and TL. RTL designs usually make use of dedicated clock signals in order to synchronize the different parts of the design. In contrast to that, TL designs use transaction events for synchronization purposes. For convenience reasons, the assertion language should handle clock edges and events in the same way in order to keep the modeling style for TL and RTL as similar as possible.

[1]We are using our own thread concept here, this is not a SystemC thread

Application

One common application for mixed level assertions is the validation of mixed level designs including transactors. Figure 2 shows a block diagram for two alternative setups for a simple FIFO used to synchronize a sending module (SND) with a receiving module (RCV). The FIFO can be accessed by two transactions (PUT, GET). Both transactions block the calling process in case the FIFO is full (PUT) or empty (GET). The lower part of Figure 2 shows a cross abstraction model. Here, the protocol at the receiving end is modeled with a clocked handshake and the gap between the abstraction levels is bridged by a transactor.

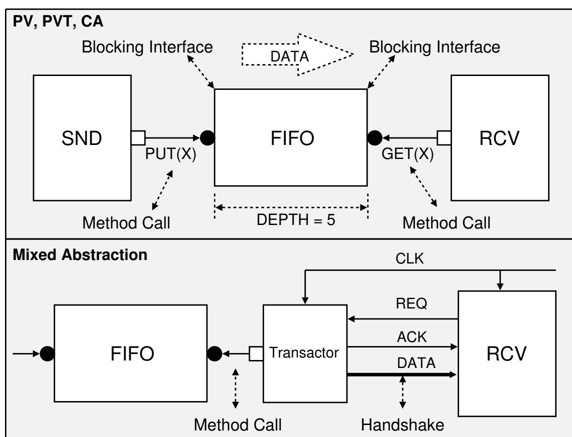

Figure 2. FIFO PUT-GET Example

Listing 1.2 shows two properties that check that data which is put into the FIFO propagates after one to six GET transactions have occurred. The upper property is working on the pure TL model whereas the lower property works on the mixed level model from Figure 2.

```
property p_DATA_PIPE_pv
 int D1;
 #1{PUT'END}{true , D1=PUT.X} |-> #{1:6}{GET'END}{GET.X == D1};
endproperty

property p_DATA_PIPE_cross
 int D1;
 #1{PUT'END}{true ,D1=PUT.X} |-> #{1:6}{ACK'POS & CLK'POS}{DATA==D1};
endproperty
```

Listing 1.2. FIFO Mixed-Level Properties

In the lower property the end of a GET transaction is captured by triggering the consequent expression only when a rising edge on both the clock and the acknowledge signal occur at the same time.

It has to be noted that in most cases mixed level assertions are written completely manually which means that the verification engineer has to know both the transaction relations involved and the underlying signal protocols. In order to ease the writing of mixed level assertions existing TL assertions could also be modified by some partial level transformations using assertion refinement as described in the section below.

5. ASSERTION REFINEMENT

Due to the very different synchronization concepts involved, an assertion refinement from TL down to RTL has to cope with several different problems: Synchronization based on events has to be transferred to one based on clocked conditions, as well as mapping the information bundled together within a transaction to a bunch of distributed registers and signals. Besides, TL designs rarely consider reset mechanisms while this is very common for RTL designs.

In this section we define our understanding of the term refinement, discuss our approach for representing transactions for RTL assertions, and introduce some new operators necessary for the refinement process.

Definition

Refinement of assertions necessarily has to follow the refinement of the design they are supposed to check. A design simulated in a simulator can basically be considered containing three different information categories:

DEFINITION 1 *Essential Information is basic information provided by the user which is derived from the original design specification, e.g. the fact that a given component shows some functionality.*

DEFINITION 2 *Modeling Information is additional information provided by the user due to the choice of the modeling implementation, e.g. the fact that the functional component uses this or that algorithm.*

DEFINITION 3 *Simulation Information is additional information brought in by the simulation semantics of the used simulation software, e.g. the fact that within this component process* A *is executed before process* B.

The first category is the only one which is completely independent from any modeling decisions made by the designer or from the choice of the software

used for testing. Hence, this kind of information[2] is also the only one which will be left definitely unchanged by refining a TL design to an RTL design. Since this information is found in both designs, a TL assertion checking this can be transformed into a similar RTL assertion by adding some RTL specific information and removing TL specific information.

Refining assertions that check information of the second category can only be refined if the verification engineer is completely familiar with both the TL and the RTL implementation, which runs against the purpose of black box testing, or if strict coding guidelines are applied to the design refinement.

Assertions checking the third category cannot be refined in general, since this would require in depth knowledge about the way the corresponding simulation software works.

DEFINITION 4 *Assertion Refinement describes a transformation of an existing assertion of a specific abstraction level to another abstraction level (usually, but not necessarily, a lower one) while ensuring that the same piece of information is checked. This transformation happens by adding required information of the new abstraction level and removing information of the old abstraction level which is no longer available.*

An example for winning information during the refinement process is the additional timing information within RTL designs which does not necessarily exist in a TL design. On the other hand the visible causal dependency between two events (e.g. *a -> b*) might disappear if they occur simultaneously in the RTL design.

Since assertion refinement does only add necessary information, all TL assertions checking essential information (see 1) can be used on all lower TL sublevels without change, i.e. a PV assertion can easily be used to verify the correctness of a PVT or CA design.

DEFINITION 5 *Partial Assertion Refinement describes the transformation of parts of an existing assertion to another abstraction level while the remaining parts are not changed.*

An example for partial refinement is the transformation of a pure TL assertion to a mixed TL-RTL assertion.

DEFINITION 6 *Illegal Assertion Refinement describes adding more information to an assertion in order to verify correlations introduced at the lower level that were not part of the upper level.*

[2]The information representation may differ, but the information itself must be available

The easiest example for that is the check of a certain timing behavior within a PVT design; the PVT level introduces timing while PV as the abstraction layer above did not contain any timing information. Thus, the original PV assertion contains less information than the derived PVT assertion and both assertions do not check the exact same behavior anymore. Since this derivation is not a legal refinement anymore, it is not discussed in the further parts of this work. Instead of illegal refinement, a completely new assertion has to be modeled.

Transaction Representation for RTL

Transactions are used for transporting data from one module to another and synchronizing their execution. A TL transaction is usually modeled as a remote function call which might be either blocking or non-blocking. On the other hand, communication in an RTL design is done via signals. The information bundled together in the transaction is usually distributed among several signals which show a certain change pattern within one or several clock cycles. Hence, each TL transaction involved has to be converted to a sequence of signal changes representing that transaction.

Here it has to be noted that it is necessary to still provide a mechanism for triggering the RTL assertions with events representing start and end of the different transactions. Otherwise, if clock edges are used instead additional information is added to the assertion which consequently does not check the same behavior as the original one anymore.

Detecting the end of a transaction is relatively easy. The simulator only has to check for the complete occurrence of the corresponding signal sequence. On the other hand detecting the start of an RTL transaction is a lot more complicated. While a TL transaction is called explicitly and can thus be easily identified from the very beginning, an RTL transaction might take several clock cycles for its complete execution; if there are more than one transaction starting with the same signal pattern in the first few clock cycles and only showing differences later on, it might be difficult to tell which transaction (if any at all) has been started by a certain signal sequence. The only possible solution to that problem lies in declaring all checked assertion results temporary until the check of the corresponding transaction sequence has been completed. At that time the result will either become valid or be discarded. The behavior of the assertion does not change in this case, only the computation becomes more resource intensive.

Parameters and return values of transactions have to be mapped to corresponding RTL registers or signals. Both these matching points and the transaction sequences have to be provided by the user.

New Operators for Assertion Refinement

The problem of still needing a representation for transaction start and end when transforming transactions to sequences can be remedied by the introduction of several new operators which can then be used as parts of the event expressions triggering a delay operator or as part of its Boolean propositions. Table 2 shows an overview of these operators.

Operator	Definition
*seq***'END**	produces an event if *seq* has matched
*seq***'START**	produces an event if *seq* has started
*seq***.ENDED**	evaluates to “true” if *seq* has matched
*seq***.STARTED**	evaluates to “true” if *seq* has started

Table 2. New Operators for Assertion refinement

These operators have to be evaluated with the occurrence of every new event, in an analogous manner to how SVA starts new evaluation threads for all assertions with the occurrence of every new clock edge.

While the detection of the end of a sequence is no problem, detecting its start is a bit more difficult. If several sequences with the same beginning signal pattern exist within the same module, a non-ambiguous detection is only possible with a certain delay as explained in the section above.

Primary and Secondary Events

If the matching of a sequence is used for triggering other sequences another problem might occur for the implementation, though: If one sequence reacts not only to the matching of another sequence but also to the event responsible for this match, it might be triggered several times instead of once.

```
sequence s1;
  #1{e1}{A == B} #1{e2}{A == C};
endsequence

sequence s2;
  #5{e2} true;
endsequence

sequence s3;
  #2{e2 | s1'END | s2'END}{B == C};
endsequence
```

Listing 1.3. Example for Primary and Secondary Events

Listing 1.3 shows an example of this situation. Sequence *s3* can be triggered by the event *e2* as well as by the match of *s1* or *s2*. In some cases the occurrence

of *e2* produces a match of one or even both sequences which might lead to an early sequence match, since one design event basically triggers both delay steps of *s3* at once. Thus, it is essential that the delay operator is only triggered once in this case.

As a solution for this we classified all events in two groups:

- Primary Events - either explicitly modeled design events or transaction events on the one hand, or external events from the timer operators used within the assertions on the other hand

- Secondary Events - derived events, in our case the start or matching of transaction sequences

If a primary event occurs during the normal execution, first all associated secondary events are computed. A specific sequence evaluation thread may be triggered at most once per primary event, regardless of the number of occurring associated secondary events.

In our example *e1* and *e2* are primary events while *s1'END* and *s2'END* are secondary events associated with *e2*. An evaluation thread for *s3* may never be triggered twice, i.e. start and complete successfully, with only one occurrence of *e2*, even if this event also causes an occurrence of *s1'END* and / or *s2'END*.

6. APPLICATION EXAMPLE FOR REFINEMENT

In this section we introduce a smart CPU subsystem and several assertions for checking its correct behavior. Using the methods and features discussed in Section 5 we show an example of TL assertion refinement.

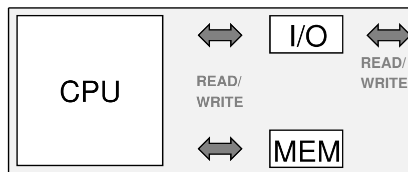

Figure 3. CPU Subsystem

Figure 3 depicts the CPU subsystem including a CPU, a memory module, and several I/O devices. All transactions in the system are modeled as remote function calls where the CPU acts as master and calls the functions provided by the memory and the I/O devices.

The correct functionality of the instruction set is checked by some existing assertions. Listing 1.4 gives an example of such an assertion. This property

```
property STD_prop
  int l_addr;
  #1{p_INSTR_WRITTEN}{p_op == cmd_STD};
  |->
  #1{read_mem'END}{true, l_addr = read_mem.param2}
  #1{write_mem'END}{(write_mem.param1 == l_addr)
                     && (p_r[p_src1] == write_mem.param2)};
endproperty
```

Listing 1.4. TL Assertion

checks the correct execution of a store instruction which first reads in the address at which to store the data and then executes the write access. The antecedent expression is triggered by writing a new instruction to the instruction register; if this instruction is a Store instruction, the implication is evaluated. The consequent first waits for the completion of a memory read access in order to get the destination address from the memory; this address is stored in the local variable *l_addr*. As a second step the completion of a memory write transaction is checked where the previously received address should be used and the data to be written has to equal the content of the specified source register *p_r[p_src1]*.

```
property STD_prop
  int l_addr;
  #1{clk'POS}{(INSTR_EN == 1) && (p_op == cmd_STD)}
  |->
  #1{clk'POS}{RD_MEM == 1, l_addr = DATA_IN}
  #2{clk'POS}{(WR_MEM == 1) && (ADDR_OUT == l_addr)
                     && (p_r[p_src1] == DATA_OUT)};
endproperty
```

Listing 1.5. Inequivalent RTL Assertion

When trying to refine these assertions without the features for detecting start and end of a transaction as listed in Table 2 the resulting assertions would look as shown in Listing 1.5. The transaction parameters have been mapped to signals and the triggering has been switched to clocked conditions. As mentioned earlier, these assertions contain additional information - in this case timing information in the form of delays by a definite number of clock cycles - and thus are not a legal refinement of the original ones. As can be seen the structure of the assertion has changed - the check for completed / started transactions has moved from the event layer to the Boolean layer and the number of delay steps has changed as well. As a consequence, as soon as the underlying RTL protocol changes, the assertions have to be adapted.

Using the operators in Table 2 on the other hand leads to the assertions depicted in Listing 1.6. The mapping of transaction parameters to signals still

takes place, but the structure of both the event trigger expressions as well as of the Boolean expressions is basically the same as that of the TL assertion.

```
sequence read_mem
  #1{clk 'POS}{RD_MEM == 1}
endsequence

sequence write_mem
  #2{clk 'POS}{WR_MEM == 1}
endsequence

property STD_prop
  int l_addr;
  #1{p_INSTR_WRITTEN}{(INSTR_EN == 1) && (p_op == cmd_STD)}
  |->
  #1{read_mem 'END}{true , l_addr = DATA_IN}
  #1{write_mem 'END}{(ADDR_OUT==l_addr) && (p_r[p_src1]==DATA_OUT)};
endproperty
```

Listing 1.6. Equivalent RTL Assertion

The additional protocol information is placed in the additional sequence declarations for the transaction sequences. A protocol change would only require an adaptation of these sequences while the actual properties remain the same. If the user provides this protocol information, the refinement process could easily be automated.

7. ACKNOWLEDGEMENT

This work has been partially funded by the European Commission under IST-2004-027580.

8. CONCLUSION AND OUTLOOK

Within this paper we gathered requirements for mixed TL-RTL assertions as well as TL to RTL assertion refinement. Further on, we suggested a methodical approach for this kind of assertion refinement which can be seen as the basis for an automated transformation from TL assertions towards RTL. We introduced the use of transaction sequences for generating the start and end event triggers as well as the concept of primary and secondary events.

An application example was given in order to demonstrate how to write mixed level assertions as well as how to apply the refinement process to given TL assertions.

Further work includes research of how to automate the refinement process by the use of meta data which guides transformation tools. This meta data might be provided manually or even obtained from existing sources.

Further useful features include partial refinement, e.g., transforming TL assertions to mixed level assertions in order to check an RTL component within

a TL system as well as the mentioned process of generating enhanced lower level assertions that check for information not available on the original higher level, e.g. generating PVT assertions with timing information from an untimed PV assertion.

Currently there are attempts of standardizing assertion APIs and generating a SPIRIT package for TL, RTL, and mixed checkers.

REFERENCES

[1] Accellera. *SystemVerilog LRM*. http://www.systemverilog.org/SystemVerilog_3.1a.pdf.

[2] N. Bombieri, A. Fedeli, and F. Fummi. On PSL Properties Re-use in SoC Design Flow Based on Transaction Level Modeling. In *6th International Workshop on Microprocessor Test and Verification (MTV)*, November 2005.

[3] N. Bombieri, F. Fummi, and G. Pravadelli. On the Evaluation of Transactor-based Verification for Reusing TLM Assertions and Testbenches at RTL. In *9th International Conference on Design, Automation and Test in Europe (DATE)*, March 2006.

[4] L. Cai and D. Gajski. Transaction Level Modeling: An Overview. In *1st International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS)*, October 2003.

[5] 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.

[6] Wolfgang Ecker, Volkan Esen, Michael Hull, Thomas Steininger, and Michael Velten. Requirements and Concepts for Transaction Level Assertions. In *24th International Conference on Computer Design (ICCD)*, California, USA, October 2006.

[7] Wolfgang Ecker, Volkan Esen, Michael Hull, Thomas Steininger, and Michael Velten. Specification Language for Transaction Level Assertions. In *11th IEEE International High Level Design Validation and Test Workshop (HLDVT)*, Monterey, California, Nov. 2006.

[8] H. Foster, E. Marschner, and Y. Wolfsthal. *IEEE 1850 PSL: The Next Generation*. http://www.pslsugar.org/papers/ieee1850psl-the_next_generation.pdf.

[9] 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.

[10] 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.

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

[12] Andrew Ireland. Towards Automatic Assertion Refinement for Separation Logic. In *21st International Conference on Automated Software Engineering (ASE)*, September 2006.

[13] 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.

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