Requirements and Concepts for Transaction Level Assertions

Wolfgang Ecker
Infineon Technologies AG
Munich, Germany
Email: Wolfgang.Ecker@infineon.com

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

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

***Abstract*— The latest development of hardware design and verification methodologies shows a trend towards abstraction levels higher than RTL, referred to as transaction level (TL). Transaction level models are used for early prototyping and as reference models for the verification of their RTL representation. Hence, ensuring their quality is vital for the design process. Assertion based verification (ABV) has already given a good return of investment for RTL designs. We expect the same benefit from leveraging ABV on transaction level; however mapping RTL ABV methodology directly to TL poses severe problems due to the abstraction of time and different model of computation.**

In this paper we present requirements for TL ABV and introduce a conceptual language for specifying TL properties. We use a simple application example for illustrating the concepts and outline a possible SystemC execution model of the conceptual language.

I. Introduction

Electronic system level (ESL) reflects a huge trend towards modeling systems at higher levels of abstraction. The main modeling paradigm in ESL is transaction level modeling. Transaction level models (TLM), in contrast to classical RTL models, contain less implementation details and are used for early system validation in the design process. Even though TLMs contain less details, they implement complex sets of functionality and therefore need to be verified thoroughly. Especially since TLMs are commonly used as golden references for RTL regressions, the quality of a TLM has to be assured as much as possible. Assertion based verification (ABV) methodology, which has been introduced to classical RTL flows over the last five years, has had great impact on verification productivity and quality assurance. In contrast to RTL, 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 paradigm is different. 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 occur. In TL synchronization is obtained by mutual dependencies of transactions and by the use of time annotations in addition to the use of non-periodic trigger signals. 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 conceptual TL assertion language. We want to point out that this language is not meant to be "yet another verification language". It should rather be considered as a suggestion for further extensions of currently established assertion languages as PSL [1] and SVA [2].

The paper is structured as follows. After discussing related work we clarify some preliminaries related to TL. Following that, we describe the requirements for TL assertions and introduce our conceptual language. We clarify our discussions with an application example. Furthermore, we outline a first implementation of our language in SystemC and close with conclusions and the next steps to be done.

II. Related Work

SystemC which seems to become the de facto standard for system-level design [3], 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 [4], [5], [6], and [7]. In contrast to that, the concepts shown in this paper aim at higher levels of abstraction for RTL concepts cannot be mapped directly to TL.

Steps are also being taken towards formal model checking of system level models as shown in [8], [9], [10], and [11]. The basis of these approaches is 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 instead focuses on dynamic verification approaches using transaction level assertions for simulation rather than for static verification. In [12] and [13] new approaches for transaction level assertions are introduced. However, in [12] transactions are mapped to signals and therefore the approach is restricted only to transactions which are invoked by suspendable processes. Waveforms of these signals are transformed into a Verilog model, which then again is checked using SVA. Our approach is based on events in contrast to signals; hence it is not restricted to a

certain kind of transaction. In [13] 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 consider start and end of transactions. Therefore overlaps of transactions and parent child relations cannot be detected. In [14] extensions to SVA have been developed to make SVA more applicable to system level design. Our approach is different in that it works with an abstraction level independent delay mechanism and also utilizes timer events and localized negative trigger expressions to allow a more flexible property evaluation.

III. Preliminaries

In this section we clarify some preliminary terms used in the remainder of this paper.

A. Transactions

A transaction describes the communication between two modules and hides the protocol specific implementation from the implementation of these modules. The most common way for modeling transactions is the utilization of remote function calls, i.e., the initiator invokes a method in the destination module. Transactions can be grouped in two classes with regard to control flow and two classes with regard to data flow. For the former we distinguish between "blocking" versus "non-blocking" transactions and for the latter we distinguish between "unidirectional" and "bidirectional" transactions. A transaction is considered "blocking" if the execution of this transaction can take at least one delta cycle. Hence the initiator has to block until the transaction is complete. A transaction is considered "non-blocking" if the execution finishes in the same delta where it was called. The distinction with respect to data flow is straight forward and does not need further explanation.

B. Abstraction Levels

We focus on three main abstraction levels as defined in the OSCI TLM standard [15]:

- Programmer's View (PV): The API to the system does not include specifics about timing. Hence, it does not matter whether the system includes timing or not. PV is mostly used in the conceptual phase of an embedded system design cycle, i.e., the designed models are closer to software than to hardware.
- Programmer's View + Timing (PV+T, PVT): The API to the system does include timing information. However, timing is modeled by time annotations in contrast to periodic signal changes. This level is mainly used at the architectural exploration phase of an embedded design cycle. By trying different HW/SW partitioning schemes performance data is gathered considering the overall timing of the system.
- Cycle Accurate (CA): The API to the system includes more granular timing information. Actions performed within the system are cycle accurate compared to an RTL implementation. However, the cycle accurate behavior does not imply the use of clocks. Cycle accurate models contain more micro architectural information when compared to PV+T models and higher.

C. Transaction Sequences

The functionality of a transaction level model can be characterized by the use of transaction sequences as described in [9]. A transaction sequence corresponds to a user specifiable pattern of consecutive transactions which occur during a simulation run. Such sequences can be used to specify properties about the system behavior. Later in this paper we show how to combine transaction sequences with boolean sequences in order to enable an expressive way for system level property specification.

IV. Requirements for TL Assertions

In this section we gather the requirements for transaction level assertions.

A. Monitoring Transactions

In order to allow specification of transaction sequences it has to be first clarified which information needs to be monitored. Obviously the occurrence of a transaction has to be detected and all arguments attached to it have to be tracked. However, detecting that a transaction has occurred is not enough. As an example, figure 1 shows a scenario where a blocking PUT transaction is issued on a full FIFO. The PUT transaction blocks until at least one GET transaction has been called on the FIFO. If only the completion of PUT and GET transactions was monitored it would be impossible to detect that a PUT blocks until a GET has finished. However, a requirement of a performance validation during the architecture evaluation phase could be to detect and minimize blocking scenarios. This example shows that the monitoring has to include the detection of both begin and termination of a transaction. This also allows the detection of overlapping transactions and thus, provides a more granular view of the system behavior. Throughout the remainder of this paper we refer to the start and the end of a transaction as start and end events respectively. The detection of start and end of a transaction is especially useful with the identification of transactions or the synchronization of any assertion to the design it is associated with. Pre- and postconditions which can be sampled at the start and end respectively can reveal enough information for either the identification of a transaction or the state the monitored design is in.

Keeping in mind that non-blocking transactions execute within one delta cycle, and can thus be called from a non suspendable context monitoring must not introduce any delta delays. In fact if delta cycles are introduced by monitors any assertion reactive to the monitored transactions will lose synchronization to the design. Therefore the events detected by monitors have to be propagated immediately to the assertions, avoiding any scheduling.

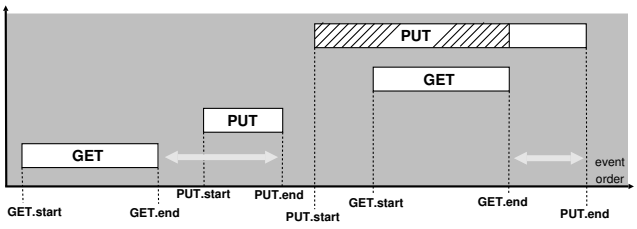

Figure 1. Blocking PUT Transaction

B. Sequence Expressions

As for the same reasons as in RTL, transaction level assertions have to keep track of sequences of boolean propositions as well. However, on RTL it is well defined when to sample the design states in order to evaluate these propositions. Usually sampling on RTL is relative to the clock of the domain which is monitored. On transaction level clocks are usually omitted because the concept of time is abstracted. The efficiency of TLMs is based on the reduction of the number of scheduled events, context switches, and the amount of detail to be simulated. Hence, synchronization schemes for TLMs strongly depend on the chosen abstraction layer. The synchronization schemes can be characterized as follows:

- PV: Processes are synchronized using abstract handshake protocols or by the use of delta delay annotations.
- PVT: Additionally to PV schemes processes are synchronized by time annotations and timed protocols.
- CA: Processes are synchronized as on PVT; however, time annotations are based on cycle periods. Besides time annotations clock like signals can also be used.

Communication in all these abstraction layers is modeled using transactions and therefore actions in modules are invoked through transaction calls. Due to this fact we can derive that transactions trigger the evaluation of sequence expressions. As described in the previous section we can use the start and the end of a transaction as a trigger similar to a clock event. Listing 1 shows a simple sequence expression:

```
#1(PUT.end | GET.end) (index == 1)
  #1(PUT.end | GET.end) (index == 2);
```

Listing 1. Simple Sequence Example

A delay operator is denoted by the # symbol. The sequence in listing 1 consists of two delay operators which are triggered with the end of either a PUT or a GET transaction on a FIFO. Identifier *index* denotes how many cells in the FIFO are occupied. This sequence matches whenever the index changes from 1 to 2 on either a PUT or a GET transaction. Since this sequence is triggered by transactions only, it is compatible to all three abstraction levels. Using transactions as triggers however introduces a new issue to be addressed. When triggering with clocks we can usually assume that the clock will produce an event in a definite time window, i.e. the clock period. In contrast to that, we cannot assume that the transactions and respectively their events will occur at all. It is possible that the evaluation of such a sequence cannot produce a result because the triggers it is sensitive to do not occur. To overcome this, sequence expressions should support the specification of negative triggers as well, i.e., triggers that prematurely cancel the evaluation of a sequence and lead to a "not matched" result[1].

As mentioned above, the PVT abstraction level considers timing as well. In order to be able to keep track of the timing information, additional requirements for sequence expressions can be formulated. For example we could express a sequence which is triggered by a PUT/GET pair where GET occurs within a specifiable time range after PUT. Listing 2 shows a sample notation of such a sequence.

```
#1(PUT.end) (some boolean expression)
  #1(GET.end@[20:30]) ...;
```

Listing 2. Time Constraints Example

The first delay operator of the sequence in listing 2 is triggered only by the end of a PUT transaction, and the second delay operator is triggered with the end of a GET transaction only if it occurs 20 to 30 time steps[2] after the last PUT. This example also shows that there is no need to restrict all delay operators of one sequence to the same set of triggers.

In case the GET transaction does not finish within the specified time range the corresponding evaluation will starve and not produce any result. As described earlier the utilization of negative triggers can overcome this problem. However, in case there is no other event in the design to serve as a negative trigger, a timer event could be specified as shown in listing 3.

```
#1(PUT.end) true
  #1(GET.end@[20:30]; timer(31)) true;
```

Listing 3. Timer Example

In case the GET transaction does not fulfill the time constraint, the sequence evaluation will produce a negative result[3] at the 31st time step after the PUT transaction.

Listing 3 shows that if boolean expressions are omitted the sequence would match as soon as this special PUT/GET pair occurs.

In contrast to PV, the synchronization schemes in PVT and CA models allow actions to happen at the same time[4]. Thus transaction level assertions should be able to capture transactions which happen at the same time. We can derive the requirement that sequence expressions should allow boolean combinations of events as triggers to achieve a higher expressiveness. Using our FIFO example we could specify a sequence expression which produces a "not-matched" result if both a PUT and a GET transaction occur simultaneously (see listing 4). Note that simultaneity is detectable only on

[1]Note that this is a negative result of the sequence evaluation rather than a simple disabling.

[2]The smallest time unit chosen in a simulation.

[3]The semicolon separates the positive from the negative triggers. Thus, the timer clause is part of the negative trigger list.

[4]The term "Time" denotes simulation time.

abstraction levels lower than PV.

```
#1(;PUT.start & GET.start) true;
```

Listing 4. Simultaneous Transactions

Listing 4 shows that the start of both a PUT and a GET is concatenated and the result of this expression is used as a negative trigger.

Additionally to the previous considerations we want to mention that use cases of transaction level assertions will most likely be in a multi-abstraction environment, i.e., mixed level simulation (RTL/TLM co-simulation). The RTL trigger concept which is based on clock events can directly be mapped to the concepts introduced so far.

C. Evaluation Modes

In this section we propose the use of different evaluation modes, which allow a more concise description of properties and sequences. These modes reflect concepts from different established ABV approaches and are not specific to transaction level assertions.

1) Sequence Evaluation Modes: In both PSL and SVA, an evaluation attempt of a sequence is started with every occurrence of the sequence trigger. The length of the pattern which is to be matched determines the length of one evaluation attempt. If a sequence contains ranges of delays all alternatives of one attempt are evaluated in parallel. Two distinct modes for obtaining a result of one sequence evaluation can be found:

- AnyMatch: Produce a match for the sequence for each matching alternative.
- FirstMatch: Stop evaluation as soon as a match for one alternative is found.

Both strategies allow overlapping evaluation attempts to match at the same time.

2) Property Evaluation Modes: The evaluation modes described in this section refer to the evaluation of implication operators within properties. An implication consists of an antecedent and a consequent expression where a match of the antecedent implies a match of the consequent. The evaluation of an implication begins with the evaluation of the antecedent. If the antecedent has matched, the consequent is evaluated. Since both antecedent and consequent can be any kind of sequence expression, the length of the evaluation of a property is the aggregation of the antecedent and consequent evaluation. Several evaluation attempts of one property can be active at the same time. Hence, evaluation attempts may overlap. This is the default behavior known from both SVA and PSL and can be expressed with the underlying formal semantics of sequence and property operators. We refer to this mode as "Overlap" mode. However, considering the Open Verification Library (OVL) [16] we find that three non-overlapping modes are introduced which are very useful for a concise property description. We refer to these modes as follows:

- Restart: An active implication is restarted on a further match of the antecedent.
- NoRestart: A further match of the antecedent is ignored while the implication is already under evaluation.
- ReportOnRestart: A further match of the antecedent is reported while the implication is already under evaluation. This match does not influence the current evaluation.

For transaction level assertions we suggest the combination of both the common SVA/PSL and OVL evaluation modes in order to gain the advantages of both approaches.

V. A conceptual TL Assertion Language

The main goal of our approach is to provide a consistent formal framework that allows the specification of properties of systems which are composed of sub blocks on different abstraction levels.

In this section we introduce a way for specifying transaction level assertions which fulfill the derived requirements given in section IV using five layers.

- **Boolean Layer:** Includes all operators returning a boolean value
- **Event Layer:** Includes all operators returning events
- **Sequence Layer:** Includes the definition of sequences and sequence operators
- **Property Layer:** Includes the definition of properties and property operators
- **Verification Layer:** Includes the verification directives

A. Boolean Layer

The boolean layer contains all common boolean operators in order to allow the specification of propositions in a system at the occurrence of events. With the `last_event(`*event*`)` function we introduce, a new operator. Whenever the evaluation point reaches this operator, it checks if the last trigger event is equal to the event given in the `last_event` operator. Depending on the result of this condition the operator evaluates to `false` or `true`. Boolean expressions are resolved after each delay operator. Delay operators are introduced later in section V-C.

Local variables are known from SystemVerilog Assertions. They are declared within the property layer hence each boolean expression within one property can access local variables. Local variables can be used in combination with boolean expressions. Everytime a boolean expression evaluates to `true` operations can be performed and results can be stored in a local variable.

B. Event Layer

As explained in section IV, verification of transaction level models has to be done on an event driven basis. An event happens in zero time and can be used for triggering processes. Our verification approach assumes that both start and end of a transaction produce a definite event that is used for triggering evaluation attempts of verification directives. As shown earlier, different levels of abstraction pose different requirements. The event layer defines all operators and functions that produce events. Table 1 shows an overview.

Symbol	Definition	Works on
ev_expr	event expression built from events and event operators;	PV, PVT, CA
e1 | *e2*	*ev_expr* produces event if *e1* or *e2* occurs;	PV, PVT, CA
e1 **&** *e2*	*ev_expr* produces event if *e1* and *e2* occur in the same time slot;	PVT, CA
ev_expr@**[m:n]**	produces event if *ev_expr* occurs within given time range; time range is relative to the evaluation point with $m \leq n$;	PVT, CA
timer(timeval)	produces event at specified time value; **timer**(10) → event scheduled at 10 time steps later than the evaluation point;	PVT, CA
posedge(signal)	produces an event on a positive edge of the specified signal	CA
negedge(signal)	produces an event on a negative edge of the specified signal	CA

Table 1. Event Layer Operators and Functions

C. Sequence Layer

This layer is used for the specification of transaction or event sequences. A sequence is a description of a pattern which is attempted to be matched against design states. The result of such an attempt can be either "matched" or "not matched". The key feature of the sequence layer is a general delay operator which works independently from the abstraction layer and thus allows the specification of sequences across abstraction levels. Figure 2 depicts the overall structure and functionality of the general delay operator. In contrast to an SVA delay,

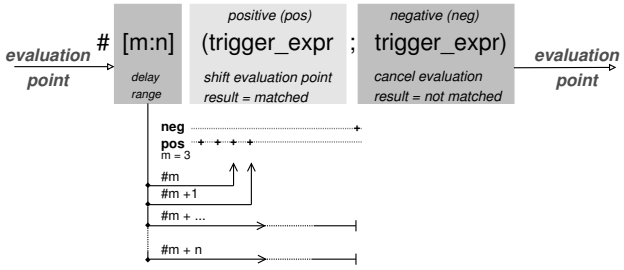

Figure 2. General Delay Operator

for example, it is possible to specify triggers that shift or specifically stop the evaluation. A trigger expression is built from event expressions (possibly using time constraints), and an optional timer event. A trigger expression may only have one timer event. Triggers that shift the evaluation are positive triggers, triggers that stop the evaluation are negative triggers. A user specifiable priority determines whether the result will produce a "match" if both the positive and negative trigger occur at the same time. This is denoted by the * operator. Per default, the negative expression has a higher priority. One evaluation attempt of a sequence built from this delay operator is called a thread. The specification of a delay range leads to a split into several so-called subthreads. Listing 5 shows a sample configuration of the general delay operator.

```
#5((e1 | e2)@[45:50] *; e3, timer(51))
        (bool_expr) ...
```

Listing 5. Sample Event Sequence

This configuration delays the evaluation until *e1* or *e2* have occurred five times with a temporal distance of 45 to 50 time steps. If the positive trigger occurs, the delay operator results in a "match" and the boolean expression to the right is evaluated. The delay operator results in "not matched" if either *e3* occurs, or the evaluation per delay step takes 51 time steps. In this example, the positive trigger has a higher priority than the negative trigger.

For a formal description[5] of the delay operator, we define an alphabet Σ that consists of all possible trigger expressions. Furthermore we define two arrays ϵ, ω. ϵ represents a trace of events, and ω a trace of boolean propositions. Thus, the elements of ϵ hold all events occurring during simulation and the elements of ω hold boolean propositions sampled on the occurrence of these events. According to that, it is given that $|\epsilon| = |\omega|$ while $|\epsilon|$ and $|\omega|$ denote the length of the corresponding arrays. An abstract notation of the delay operator is given in listing 6.

```
#N(pos; neg) a
```

Listing 6. Formal Event Sequence

In listing 6, *a* is a boolean expression while *N* denotes the number of times the delay operator has to be triggered. The positive and negative trigger expressions are referred to as *pos* and *neg*. In order for this sequence to match the following conditions have to be fulfilled:

- $\epsilon, \omega \models \#N(pos; neg)\ a$:
  ϵ and ω have to model the sequence
- $\epsilon \vDash \#N(pos; neg)\ a$ iff there exists at least one interval $\epsilon_k = [\epsilon(k); \epsilon(k{+}N{-}1)]$ with $(\epsilon_k(i) \vDash pos) \wedge (\epsilon_k(i) \nvDash neg)$, $0 \leq i \leq N-1$ and $0 \leq k \leq \infty$
- $\omega \vDash \#N(pos; neg)\ a$ iff there exists at least one element $\omega(k+N-1) = x$ with $x \vDash a$

Hence, the general delay operator can be used to track both sequences of events and boolean propositions along these sequences. The operator does not depend on the abstraction layer, since the different semantics are obtained by the definition of events on different abstraction levels. From a syntactic point of view, a sequence can be defined using delimiter keywords as shown in listing 7.

```
sequence s1
   #1(e1) true #1(e2) ... ;
endsequence
```

Listing 7. Sequence Syntax

Note that since triggers are specified locally in every delay operator, a sequence specification has to start with a delay operator.

[5]The formal description is similar to Annex E of the SystemVerilog P1800 standard. Therefore only basic relations are shown.

D. Property Layer

The property layer is located on top of the sequence layer. A property has a boolean notion and describes the intended behavior of a design. The syntax of a property is shown in listing 8.

```
property p1(AnteMode Amode, PropMode Pmode)
   antseq |-> conseq;
endproperty
```

Listing 8. Property Syntax

As shown in listing 8 a property instantiates sequences and specifies implications. Furthermore, local variables are declared within this layer. The implication operator is untimed - i.e., no delay is introduced - and works like a boolean implication. While the sequence mode of the left hand side (`AnteMode Amode`) is parameterizable, the sequence mode of the right hand side is fixed; this is explained later in this section. The way antecedent and consequent are related to each other depends on the property mode (`PropMode Pmode`). As explained in section IV we identify four different modes: Restart, NoRestart, ReportOnRestart and Overlap.

Combining sequence and property modes results in a very limited number of feasible possibilities.

A sequence on the left hand side of an implication can have any of the sequence modes, for there is no dependency between it and the property mode (note that the standard mode of SVA for the left hand side is "AnyMatch").

The mode of a sequence at the right hand side of an implication is fixed. Here, only mode "FirstMatch" makes sense. A sequence at the right hand side run in mode "AnyMatch" might produce several matches which would result in several property results (possibly even both fail and pass for a single antecedent match).

E. Verification Layer

The verification layer is used for specifying which properties have to be monitored during verification. Listing 9 gives an example about the syntax of the verification layer.

```
verify SLV
   directive(p1(AnyMatch,Overlap),
             assert_cover(ERROR, "msg" ,all));
endverify
```

Listing 9. Verification Layer Example

Within the `directive` statement we specify the property (*p1*) which should be checked. As parameters we specify the evaluation modes of the antecedent sequence (`AnyMatch`) and the property (`Overlap`). The second parameter within the `directive` specifies whether to assert, to cover or to both assert and cover the property. The `assert` directive interacts with the simulator and invokes actions if a property to be asserted fails. Therefore it can be parameterized with a severity level, in order to specify the degree of action. Furthermore, an assertion message can be specified. The `cover` directive can be used instead of the `assert` directive. It does not assert if the property fails but information about vacuous and non-vacuous successes will be collected. Therefore the `cover` directive provides a parameter in order to specify the type of coverage. It is possible to cover vacuous successes, non-vacuous successes, fails, and any combination of those. With the `assert\_cover` directive as a further possibility, both assertion and coverage actions will be performed. No new features are needed for this layer to fit the requirements for TL-Assertions.

VI. Application Example

In this section we introduce a simple transaction level PVT model and explain how to apply some of the introduced concepts.

A. CPU-Array

Figure 3 depicts the application model including the proxy monitors used for the transaction detection.

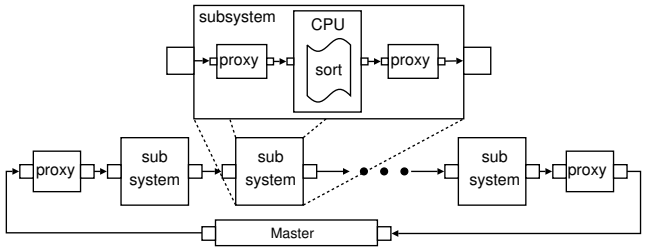

Figure 3. CPU-Array

The model consists of an array of 16 subsystems, each including one CPU and I/O ports for data transfers. The communication between a CPU and its I/O devices uses blocking transactions for reading and writing to the peripherals. The CPU blocks when the addressed device is not ready for that access, i.e., an IN device can only be read by the CPU if its data register contains valid data and an OUT device can only be written if its data register is empty. Each subsystems output port is connected to the input port of the next subsystem. 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. The program flow is depicted in figure 4.

At first the number of data values to be sorted is read in and then passed on to the next subsystem. This value determines the number of iterations of the implemented loop. Following that, the first sort value is read in and stored within the *R0* register of the CPU. Then the second sort value is read in to register *R1*. After a comparison between *R0* and *R1* the greater of both values is sent to the next subsystem. Then the execution loops back to reading in the next sort value. Once all iterations are done the subsystem sends out the remaining value.

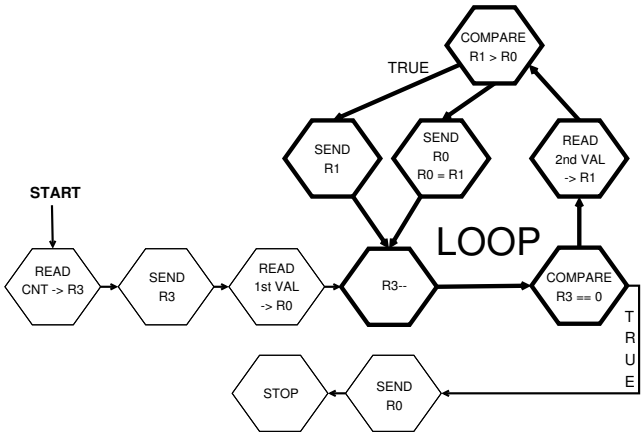

Figure 4. Program Flow of Sort Algorithm

When the first sorted value propagates to the output of the array, all remaining values are expected to arrive with exactly 10 time steps distance at the output.

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

B. Assertions for the Application Model

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

- prop-SortVal-pv: Within the loop of the sort algorithm in one instance of a subsystem, a value that is read in is propagated to the output if it is greater than the value stored in the CPU's *R0* register and vice versa.
- prop-17in17out-pvt: Pushing 17 values in the array implies 17 values at the output of the array 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 is formulated regardless of time, whereas the second property requires further time information.

Property "prop-SortVal-pv" has to hold 15 times for each instantiated subsystem. Listing 10 shows the notation of this property. The antecedent sequence is triggered with the end of a read transaction that is invoked by the CPU. The consequent sequence is triggered with 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 10. Property: "prop-SortVal-pv"

Since the sort algorithm requires two values to be read into the CPU prior to starting the loop, the antecedent has to be formulated such that the first two read transactions are ignored. The loop starts with a read transaction that stores the value in the *R1* register. Hence, we can detect the beginning of the loop by requiring *R1* to contain a non zero value when the transaction has finished. Once this condition is detected, the two values to be compared by the algorithm are stored in local variables. The consequent sequence evaluates whether 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" for the sequence has to detect any completed read transaction. The evaluation of the property mode is "ReportOnRestart" for the occurrence of a further read transaction prior to a write transaction is illegal behavior within the loop.

The second property is formulated in listing 11.

```
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 11. 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 array. The identifier *b_put_out_end* denotes the end of the transaction (*put_out*) that propagates a value out of the array.

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 array 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 ensure 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 array. 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 the time constraint operator to ensure that the corresponding event is only accepted if it occurs exactly 10 time steps after one step of this delay has begun. The timer operator is applied in order to ensure that the sequence will produce a result.

VII. Implementation

In this section we outline a SystemC implementation of the concepts introduced above. Since SystemC supports object oriented design it is possible to model most of the operators from section V as classes or modules respectively. The benefit is that these operators can be interconnected using the same mechanics as in a TLM. The modular concept allows an easy assembly of any sequence expression.

Generally all operators communicate via tokens, which include all necessary data for evaluating a sequence. The operators work in a pipelined way and utilize dynamic data structures - mainly from the Standard Template Library - to preserve the order of execution.

The transaction detection works on the basis of proxy modules which intercept transactions and extract a copy of their payloads. To avoid any effect on the design the transaction detection works seamlessly from the design's point of view. The transaction start and end events are not implemented using "sc_events" for this would involve scheduling and could lead to non determinism and delta delays. Start and end events are implemented as call backs to the assertions which are located parallel to the design. By using call backs we ensure that all assertions react to the occurrence of a transaction before further state changes in the design can happen. Hence, the assertions remain synchronous to the design and react within the same delta where a transaction starts or ends respectively.

A more detailed description of the implementation and experimental results cannot be given for it would exceed the scope of this paper, by far. The utilized algorithms and the assertion architecture including its effectiveness will be subject of another publication.

VIII. Conclusion and Outlook

The discussions in the previous sections show that applying assertion based verification to transaction level models offers new ways of high level system verification. We first have gathered the requirements to a TL-ABV approach. Based on these requirements we introduced a conceptual assertion language. We were able to combine verification on all abstraction levels into one consistent event based verification approach by considering transactions as a sequence of a start and an end event. Furthermore, we abstracted time by replacing timed delays by timer events and time constraints. Therefore it was possible to define a general delay operator that allowed building sequences of transactions regardless of the abstraction level. In addition to these definitions we also combined evaluation modes which are distributed over several ABV approaches into one concept. Our application example showed that the concepts are feasible enough to justify further research in this direction.

Further additions have to be developed that allow for instance combinations of sequences using sequence operators and boolean expressions formulated on properties. In addition to that we work on a fully pipelined evaluation mode that allows exact matches in sequences.

Furthermore, a methodology on top of our concepts has to be developed to maximize the benefit of ABV for TL.

References

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

[2] IEEE Computer Society, "SystemVerilog LRM P1800." [Online]. Available: http://www.ieee.org

[3] G. Martin, "Systemc and the future of design languages: Opportunities for users and research," in *16th Symposium on Integrated Circuits and Systems Design*, 2003, pp. 61–62.

[4] T. Peng and B. Baruah, "Using assertion-based verification classes with systemc verification library," *Synopsys Users Group, Boston*, 2003.

[5] A. Habibi and S. Tahar, "On the extension of systemc by systemverilog assertions," in *Canadian Conference on Electrical & Computer Engineering*, vol. 4, Niagara Falls, Ontario, Canada, May 2004, pp. 1869–1872.

[6] W. Ecker, V. Esen, J. Smit, T. Steininger, and M. Velten, "Implementation of a systemc assertion library," in *IP Based SoC Design (IP/SOC)*, December 2005, pp. 9–13.

[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*, Sonoma Valley, California, USA, November 2004, pp. 19–22.

[8] R. J. Weiss, J. Ruf, T. Kropf, and W. Rosenstiel, "Efficient and customizable integration of temporal properties into systemc," Lausanne, Switzerland, September 2005.

[9] 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.

[10] A. Habibi and S. Tahar, "Assertion and model checking of systemc," in *North American SystemC Users Group Meeting*, San Diego, California, USA, June 2004.

[11] D. Große and R. Drechsler, "Formal verification of ltl formulas for systemc designs," in *International Symposium on Circuits and Systems*, vol. 5, May 2003, pp. 245–248.

[12] B. Niemann and C. Haubelt, "Assertion based verification of transaction level models," in *ITG/GI/GMM Workshop*, vol. 9, Dresden, Germany, February 2006, pp. 232–236.

[13] 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.

[14] E. Cerny, "Personal discussions yet to be published."

[15] A. Rose, S. Swan, J. Pierce, and J. M. Fernandez, "Transaction level modeling in systemc." [Online]. Available: http://www.systemc.org

[16] Accellera, "Open Verification Library." [Online]. Available: http://www.accellera.org/activities/ovl/