MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
KHARKOV NATIONAL UNIVERSITY OF RADIOELECTRONICS

ISBN 966-659-113-8

Proceedings of IEEE East-West Design & Test Workshop (EWDTW'05)

Odessa, Ukraine, September 15 – 19, 2005

CONTENTS

ASSERTIONS BASED VERIFICATION FOR SYSTEMC

Miroslaw Forczek[1], Sergiy Zaychenko[2]

[1] Aldec-ADT, Compilers Division, ul. Lutycka 6, 44-100 Gliwice, Poland, mirekf@aldec.com.pl

[2] Design Automation Department, Kharkiv National University of Radio Electronics, Lenin ave, 14, Kharkiv, 61166 Ukraine E-mail: sergeyz@cooldocteam.com

Abstract. The Assertions Based Verification (ABV) has gained worldwide acceptance as verification methodology of electronic systems designs. Assertions provide basic blocks for building **functional verification** concept. Due the declarative form of the temporal formulas of assertions a lot of verification efforts are being cut down, tending to better product quality and verification speed. Most of implementations are integrated with HDL based design environments. The SystemC open initiative provides an alternative to HDL based design environments by enabling C++ with hardware concepts. SystemC already became a very popular environment for modeling at system-level abstraction. This work enables SystemC designs with industry standard assertions notations. The platform was build upon assertions simulator integrated into Riviera™ HDL based verification environment.

Keywords: device simulation, computer aided design, verification, assertions, system level modeling.

1. Introduction

The Assertions Based Verification (ABV) has gained worldwide acceptance as verification methodology of electronic systems designs. There was number of papers [1-3] that explain in-depth this methodology. The original concept of assertion comes from software development where it (in particular the *assert()* macro defined in C language [4]) has proved to be a very powerful tool for automatic bug and regression detection [5]. Assertions for hardware designs employ Linear Time Logic (LTL) to define expected and/or forbidden behavior. The foundation for ABV are Hardware Verification Languages (HVLs). HVLs combine semantics of LTL with constructs for building reusable verification IP units. Verification IP units need to be bind to some design for effective use. Thus HVLs provide constructs to specify connections with models in Hardware Description Languages (HDLs). Most of ABV implementations are part of HDL–based integrated design environments (IDEs).

The SystemC open initiative [6] provides an alternative to HDLs as it enables C++ [7] – the industry strength notation for complex systems – with hardware concepts of RTL and system-level in form of C++ templates library. In its original approach SystemC models are processed standard C++ toolset and executed as standalone applications. SystemC became a very popular environment for modeling at system-level abstraction. The HDL-based IDEs offer co-simulation capabilities with SystemC engine but it still remain external unit to the HDL simulator. The idea of applying ABV to the SystemC designs is natural step of HDL and SystemC environments integration.

Since HDL design can be co-simulated with SystemC model, there is an easy way to associate verification unit with SystemC one: the SystemC unit needs to be connected to HDL wrapper unit that will provide entry point for verification unit bind – as shown on fig. 1. This method doesn't require any additional tools assuming availability of HDL simulator.

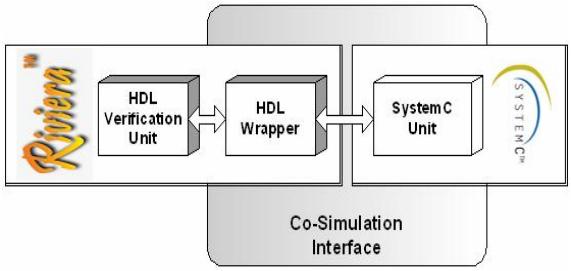

Fig. 1. Connecting verification unit to SystemC unit with intermediate HDL wrapper

However there are some serious limitations for this kind of connection:

- Only units interface signals can be accessed this way. This highly limits usability of assertions. One of the most appreciated features of HVLs is their ability for seamless access to any signal within HDL design across any kind of interface and/or encapsulation constructs. However this rule will mostly not work behind co-simulation interface. Usually co-simulation connections supports only units interfaces signals.
- Verification formulas cannot refer directly to the source object and their native types. SystemC has a rich set of data types and it can even be extended with additional user types. Some HVLs (in example – the PSL) allow to build formulas with use of semantics of description language for attached target unit. Unfortunately co-simulation interface will enforce conversions of all source data types into types system of the intermediate HDL.
- HDL simulator has to be used, even when no HDL models are actually used in a design.

To overcome above limitations assertions shall be compiled into SystemC native units and linked directly with related SystemC model elements. Such approaches were already attempted – in example: Große and Drechsler [8,9] - however they relayed on non-standard notations of formulas.

An approach presented here utilizes industry standard HVL notations: OVA [10] and PSL [11], which are widely used within HDL verification environments. This system does not transform the design model in anyway. Rather than we provide a specialized co-simulation engine and a mechanisms for direct linkage of properties models to the design structures at the native SystemC level. In this way there is no additional limitations for design coding – any C++ constructs (especially dynamic data structures, classes and templates) can be used to build electronic systems models of a very high level abstraction.

The system was derived from already existing assertions entry in Riviera™ HDL oriented simulation-based verification environment [12]. We reused industry proven multi-lingual assertions compiler and high-performance assertions simulation engine. The assertions compiler supports number of industry standards notations as OVA, PSL and others.

Thanks to properly designed framework of the entry and right architectural decisions while defining key interfaces between main components of Riviera™ system, the task of integration with SystemC environment was reduced to finding equivalents between HDL-based system semantics and SystemC semantics and than porting concepts of the entry's building blocks and their realizations from HDL to SystemC domain.

Goal of this paper is to describe construction of the Assertions for SystemC platform and to present key architectural decisions that lead to re-use without problems a proven assertions entry originally designed for HDL based simulator. The *main research topics* include:

- how to model assertions checkers for efficient co-simulation efficiently with minimal impact on related design model processing,
- how to abstract assertions checkers models from related design modeling environment and how to encapsulate assertions into units that will match native binary formats of that environment,
- comparison of HDL and SystemC simulation engines and SystemC kernel supplement with extensions required for handling details of assertions semantics,
- how combination of SystemC (C++) semantics and properties language semantics opens new capabilities for electronic systems modeling and verification.

2. Architecture of assertions entry

The assertions processing entry in Riviera™ consists of the following main components, grouped into the compilation (fig. 2) and simulation parts (fig. 3):

- the compilation front-end including lexical and syntax analyzers for the OVA and PSL notations;
- a common notation-independent modeling layer of assertion semantics;
- a synthesizer of pipelined RTL checkers structures that perform detection of the assertion formula matching sequences occurrences while the design evaluation;
- a specialized simulation engine for evaluating assertions with high-performance interface for the direct sampling of the design data.

Evaluation of assertions is based on RTL structures synthesized upon formulas expressed in declarative style. In principle the RTL structured checkers can be physically implemented in various domains. This paper focus on Riviera™ system, which uses simulation, oriented software based implementation. But there is also Riviera-IPT™ system [13] in which assertions checkers are being compiled and implemented into FPGA based accelerator. These RTL structures could also be used with formal proof engine.

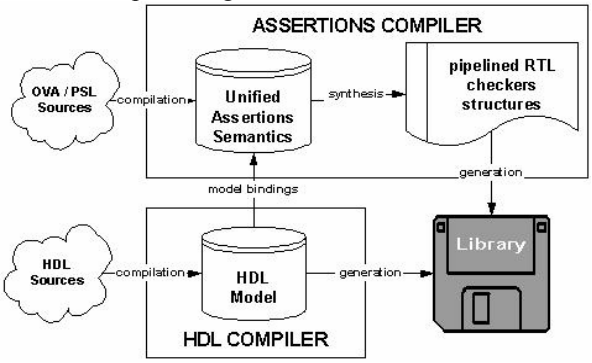

Fig. 2. Assertions engine architecture: compilation part

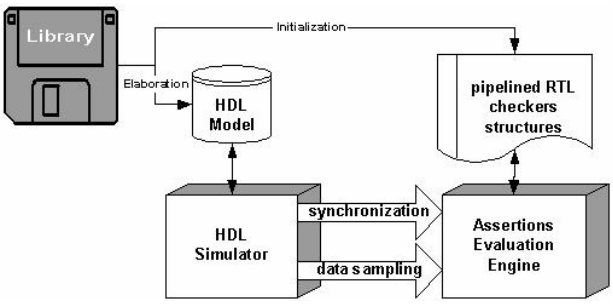

Fig. 3. Assertions engine architecture: simulation part

On fig. 4 a basic idea of the RTL checker synthesis is shown. A more in-depth study of the RTL checkers structures is beyond scope of this paper, one can find more details in [14].

PSL definition:

```
vunit v
{
  default clock = @rose(clk);
  property e4 = always {e1; e2};
  assert e4;
}
```

Synthesized RTL checker structure:

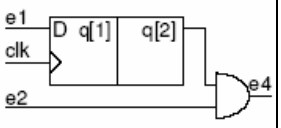

Fig. 4. Basic RTL checker structure

Although the assertions entry contain separate compiler it does not provide it's own binary storage format nor loader and linker toolset. It depends on design verification toolset. Compiled assertions units shall be binary compatible with the librarian system and simulation engine of the targeted verification environment. To meet this requirement the assertion checker logic is packed into standard container unit within the verification environment (fig. 5).

The container unit encapsulates assertion checker construction details and provide binary format for compiled storage form compatible with rest of the system. At the model startup phase when design components are loaded from library and linked together inside simulator, the container unit interface mapping automatically solves problem of connecting to design objects data structures that were referenced from assertion formula (fig. 6).

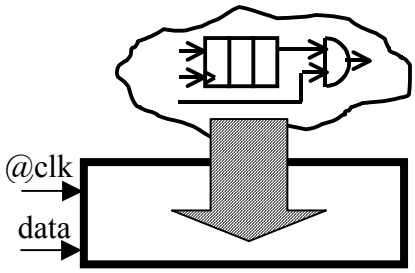

Fig. 5. Packing assertions models into the native container unit of the verification environment

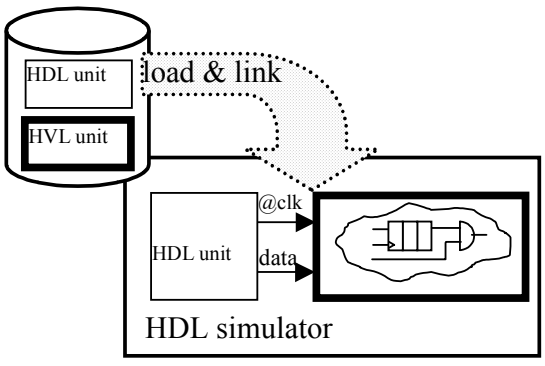

Fig. 6. Startup phase of the simulation with assertions

The RTL structure of the checker is encoded within container unit in form of initial process that contains sequence of calls to so-called *builder-API* of the assertion engine. The build-script is executed while design model initialization phase and thus the assertions RTL structures are created too. The container unit is filled also with continuous processes synchronized with clock signals as defined in assertion formula. The processes provide synchronization service for assertion engine while run-time phase making a call to *Run()* routine from the assertions API at each clock tick (fig. 6).

As the assertions engine API is invoked only from within container unit, there is no need for physical static dependency between simulator module and assertions module. All the calls can be realized with means of a standard extensions interfaces like PLI interface defined in Verilog-HDL. The assertions engine remains a fully independent and portable module.

The assertions engine has to be fed with stream of data from the design model and synchronized with related clock signal events. Just at the beginning of the system construction the requirement was defined that the interface for data sampling and synchronization needs to provide hi-performance throughput but is has to be abstracted enough to separate assertions engine implementation from HDL simulator. In this way the assertions engine actually can be fed from various sources – not only from simulator module (HDL – based one or event non-HDL) – but in example from simulation database streams: files, waveforms, etc.

To separate assertions logic from exact types of values accessed from design model, the concept of abstract data samplers was introduced (fig. 7).

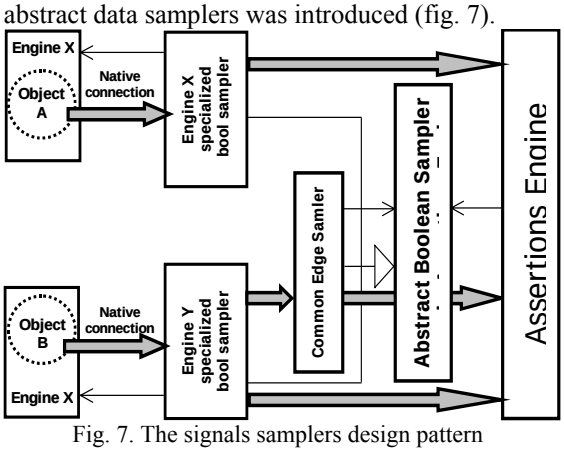

Fig. 7. The signals samplers design pattern

Almost all semantic of assertions formulas is defined over Boolean and linear time domains. The underlying HDL typing system is converted into Boolean domain just at the references to HDL objects from assertions formulas. Placing the HDL

data conversion into Boolean at the sampling interface highly reduces the size of the data stream for assertions engine. Finally it is possible to provide specializations of the samplers for each data type (*logic bit*, *vector*, *vector* selection or slice, *real*, etc.) and each kind of design evaluation engine – optimized for performance. The samplers are linked directly to the references objects representations at the design simulation engine (fig. 8) – with full knowledge of the native data format in the simulation engine and no intermediate communication interfaces nor channels, the samplers offer best possible performance. Also the simple samplers concept made a foundation layer for higher order samplers like *edge samplers*, which are implemented over simple samplers. Initially the assertions engine was supplied with data samplers family specialized for Riviera™ simulator.

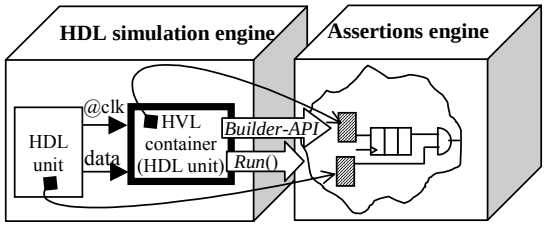

Fig. 8. HDL and assertions engines deployment diagram. Samplers are linked directly to the native data source objects of design

Despite of the fact that assertion checker logic description is packed into container unit and initially the design objects references are obtained thru standard port-maps connection mechanism, the whole RTL structure is created and resides at assertions module side and the samplers re-link the connections into direct references to the referenced objects while initialization phase to eliminate any performance loss.

3. Assertions entry for SystemC

SystemC design platform can be easily enabled with assertions using the previously described architecture. Due the flexibility of assertions engine and extensibility of SystemC library, only minimal extensions are required in both to use assertions at system level.

The front-end of assertions entry remains untouched - OVA and PSL sources can still be processed by the same compiler as for HDL architecture. In this case, to meet the requirement of the formats compatibility with the design verification toolset, the assertions RTL structures logic should be generated in form of SystemC source code.

Assertions model startup at the SystemC design elaboration phase is implemented with already discussed *builder-API* of the assertions engine – only recompiled for C linkage binary compatibility. In the SystemC native container unit the assertions compiler generates the SystemC modules that implement container units with build-scripts placed in the constructors. These modules encapsulate SystemC environment specific implementation details from the assertions simulation engine and provide native high-performance streams of input data and synchronization.

The idea of value and edge samplers, introduced in previous section, can also be easily applied to SystemC designs. Assertions engine should have support for samplers, which can query values from various types of SystemC ports and channels while preserving common interface requirements for all samplers (fig. 9). Such data access could be done without difficulties, as the SystemC data structures are shared as open source.

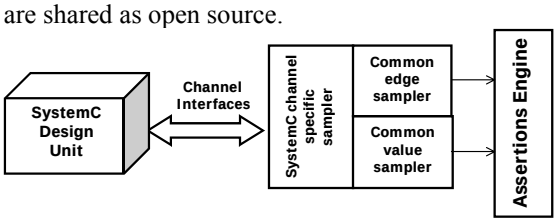

Fig. 9. SystemC channel specific samplers

The requirement to the channel samplers is that they must implement specific access mechanism for stored values. For primitive channels, like *sc_signal*, *sc_semaphore*, *sc_fifo*, or for SystemC and C++ data types, like *sc_logic*, *sc_int*, the implementation of sampling is obvious, as it only demands proper mapping between common sampling and channel value access interfaces.

Complex user-defined channels can also be linked to the assertions engine, if they meet the requirements of common value/edge sampling interfaces. Generally, value of the user-defined channel is not bound to integral types, like in HDLs. For complex data structures – i.e.: system-level transactors - user has to define converters to bool type, to clearly state which value should be considered as false, and which value change should trigger signal edge events.

Having input data from specific samplers, the assertions engine can be linked to the standalone executable SystemC design as an additional dynamic linked library (fig. 10).

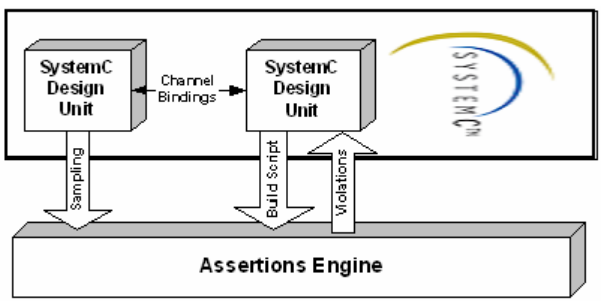

Fig. 10. Simple architecture of assertions for SystemC

After the design elaboration phase assertions engine becomes part of the executable SystemC design model, reacting on all assertions violations within the simulation process, which is controlled by the master SystemC verification toolset.

However, because of scheduling differences between SystemC and HDL engines (fig. 11), a few extensions are also required in the core of SystemC library:

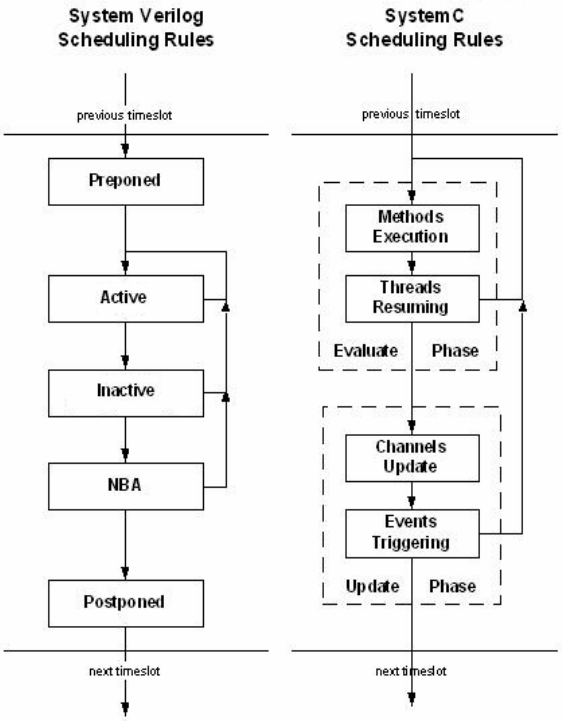

Fig. 11. Comparison between System Verilog and SystemC scheduling rules

To avoid errors with sampling values at incorrect moment of processing, which are connected with unpredictable simulation order of the parallel threads, special *preponed processes* have to be introduced within the SystemC simulation engine. Implementation of the support for these processes is trivial: preponed methods just need to be executed at the beginning of the timeslot before starting the first evaluation phase. The intention is to sample the values before any signal has changed the state within the considered timeslot. Preponed processes have the following important properties:

- preponed processes cannot modify any object under control of scheduler, in other words, they should not schedule execution of any other process;
- preponed processes have to be only methods; there should be no possibility to suspend the execution by call to wait-construction.

It is also necessary, that the assertions engine step should be activated as *postponed process.* The reference materials for the assertions notations explicitly specify to evaluate the asserted property expressions only after all other design processes had stabilized. This approach prevents the potential improper evaluation of the assertions in case, when the clock domain signals are affected several times across the delta-cycles of the single simulation step.

Postponed processes can be easily enabled within the SystemC scheduler by invoking them after the stabilization of all regular processes. Similarly to the preponed ones, postponed processes cannot suspend the execution with the wait-construction. Besides, they can affect any signals and trigger new events, which will be handled at the next timeslot.

To enable simulation and assertions checking of SystemC units among with ordinary HDL modules within the single harmonic design, much more complex co-simulation environment is required (fig. 12).

In this environment SystemC engine is no longer a primary component. Instead, the simulation control is transferred to the special co-simulation interface, synchronizing cooperation of HDL simulator, SystemC scheduler and other external simulation components (f. ex., embedded software debugger, analog and mixed-signal AMS simulator). Much deeper changes are required within original sources of the SystemC library to convert it from master simulator into slave one under the control of co-simulation environment.

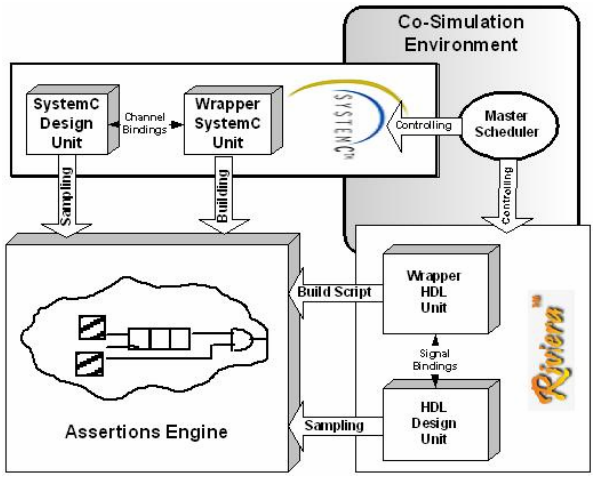

Fig. 12. Advanced co-simulation environment for checking assertions upon SystemC models

One more noticeable issue is a model binding. In HDL architecture binding modules for assertions are generated automatically, as compiler either processes HDL modules in the same compilation pass (HDL and assertions compilers of Riviera™ are integrated together and can exchange data while compilation), or has access to the compiled library through flow-specific interface. It is than possible to check the binding connections during assertions compilation.

In case, when the SystemC modules are used among with HDL units within co-simulation environment, binding wrapper can be also generated easily, assuming the information about the SystemC unit interface, structure and internal signals is already presented in the co-simulation design library. Typically, this step is required anyway to enable co-simulation of SystemC module within HDL-oriented environment.

Less convenient case is the standalone SystemC application. Assertions compiler can still generate connections between design model and assertions engine, basing on the OVA/PSL binding specification, but unfortunately without strict type checking at compilation time. Implementation assumes that any kind of type mismatch problems will be caught automatically while building the SystemC application within C++ IDE. This could be improved, but would require an integration with C++ compilation system.

4. Assertions for SystemC: a case study

Let's demonstrate the benefits of the assertions based approach on a SystemC design, which contains a transaction level model of a dual-port RAM block (fig. 13). Such memory blocks are typical for many ASIC and FPGA designs. Simulating RAMs at such high level of abstraction is sufficient, as RTL or gate-level memory block representation leads to large performance overheads. Having RAM model at high software level is very effective on the early design phases, as it strongly increases the simulation speed.

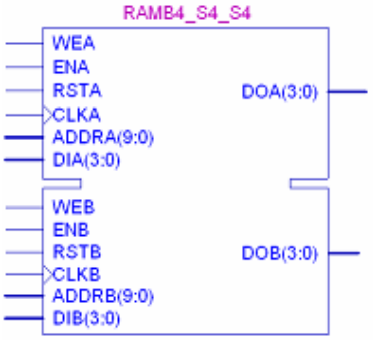

Fig. 13. Example of dual-port RAM block

RTL and lower description levels of the RAM involve many ports, such as clock, reset, address and data i/o buses, write and clock enable signals, which altogether should be set in the right order to perform verification operations. For a dual-port RAM the groups of signals are duplicated for each port. Using the SystemC, such RAM interface can be replaced by a convenient C++ transaction level channel, hiding the RAM control details behind the high-level entry methods:

```
SC_MODULE DPRAMTrans
{
  ////////////////////////////////////
  /** Transactor sampling codes */
  static const int NO_ACTIVE_WRITE = 1;
  static const int WRITE_STARTED   = 2;
  /** Clock signal */
  sc_in<sc_logic> clk;
  /** RAM component */
  DPRAM * theRAM;
  ////////////////////////////////////
  /** Memory reset method */
  virtual void reset ();
  /** Port enabling/disabling method */
  virtual void set_enabled (bool enabled);
/** Accesses the specified memory word
   *
   * @param addr    Address of the memory word
   * @return        Stored value
   */
  virtual sc_int read (sc_int addr);
  /** Modifies the specified memory word
   *
   * @param addr    Address of the memory word
   * @param value   Value to store
   * @return        Actually stored value
   */
  virtual sc_int write (sc_int addr,
                        sc_int value);
  /** Generic sampling operator */
  template<int SMPL_TYPE> bool Evaluate () const;
  /** Checks whether the active addresses
   *  of two specified ports overlap
   *
   * @param portA   First port channel
   * @param portB   Second port channel
   */
  static bool AddrOverlap (DPRAMTrans* portA,
                           DPRAMTrans* portB);
  ////////////////////////////////////
};
```

As the interface of each RAM port is identical, the RAM itself should contain two instances of the port transactor class (fig. 14):

```
SC_MODULE DPRAM
{
  // …
  DPRAMTrans* portA;
  DPRAMTrans* portB;
  // …
};
```

Fig. 14. Dual-port RAM with transactor ports

Duality of the RAM interface allows performing two simultaneous read/write operations at the same time, including different clock domains. Typical problem of the designs with RAMs are collisions: two simultaneous write operations upon the same memory cells. At simulation level this typically brings the memory into the unknown state, but such situation is highly undesirable in the actual hardware. The following PSL assertion (with SystemC flavor), detecting the possible collision situa-

tion, might be very helpful for verification of the entire design:

```
vunit ram_write_collision_detector (DPRAM)
{
   sequence active_wr(DPRAMTrans p) =
      {p<DPRAMTrans::NO_ACTIVE_WRITE>;
       p<DPRAMTrans::WRITE_STARTED>;
      [*1]} @(p.clk);

   sequence a_wr = active_wr(a);
   sequence b_wr = active_wr(b);
   sequence sim_wr = {{{a_wr} within {b_rw}}
                |{{b_wr} within {a_wr}}}
                @(a.clk or b.clk);

   property collision = {sim_wr} |=>
       DPRAMTrans::AddrOverlap(a, b);

   assert never collision;
}
```

This assertion checks whether there is an overlapping of the addresses within two active simultaneous write operations. It should be checked at each edge of the RAM clocks, meaning starting/finishing moments of any read/write operation. Assertion uses values of the elementary SystemC signals (a.clk, b.clk) and also the user-defined conditions for determining the writing activity, address overlapping. To enable such high-level constructions for the assertions engine, being described in this paper, user has to define proper value sampling interfaces for unique design elements:

```
template<>
bool  DPRAMTrans::Evaluate<NO_ACTIVE_WRITE>
()
{
   // …
}

template<>
bool DPRAMTrans::Evaluate<WRITE_STARTED> ()
{
   // …
}
```

Here, such transactor state condition, as writing activity, is considered as atomic for assertions engine. However, these user-defined transaction level conditions can group several other sub-transactions to hierarchically manage various interaction protocols. With the suggested flexible event sampling architecture, the verification is not limited to a particular C++ coding style or design organization. Obviously, enabling assertions for custom SystemC design elements requires some efforts from the engineer, but they are incomparably smaller than equivalent testbenches without the assertions notations. Usage of transaction level sampling methods is much more convenient at lower levels of the abstraction, as it does not demand tracing of the complex signal sequences. So, the demonstrated example case clearly shows the benefits of the suggested assertions based approach for SystemC units, which simplifies the designer's work by cutting off large verification efforts.

5. Conclusions

The implementation of assertions entry for SystemC was presented in this paper. The assertions evaluation engine is a key component of Assertions Based Verification (ABV), Functional Verification (FV) and Coverage Driven Verification (CDV)–already proven design methodologies for large-scale electronic systems [15-17]. While assertions support is a standard option now for HDL-based environments, it is still a novel for SystemC-based . The system presented here enables designers using SystemC with all the mentioned methodologies that were lack of until now.

To build reliable and industry strength verification platform for SystemC we ported commercial assertions entry proven in the HDL-based Riviera™ verification environment. The stress was put on demonstrating how the properly structured implementation of the original HDL-based entry lead in easy finding of corresponding means in SystemC domain and in porting the whole solution with minimal implementation efforts and without breaking up the original fundamental concepts put in the original system.

A similar attempts were made in a past but they didn't reach an industry applications level for number of reasons.

Both systems of Große and Drechsler use formal proof engine for evaluation of the design and properties specifications. From theoretical point the formal engine is better than simulation based one as it is able to perform exhaustive proof of property correctness while simulation based evaluation is limited to the narrow path of testbench stimulations. But on the other hand the design logic and properties specifications needs to be transformed into finite state machines (FSM) representations before they can be processed by the proof engine. This requirement seriously limits spectrum of constructs that can be used for design modeling. The transformation step needs advanced algorithms that are able to map source design logic into FSM logic. While this is not a problem for designs described at gates or RTL level, the complexity of the task quickly increases for higher level of abstractions: behavioral and system-level. It would be very hard to develop so advanced transformation algorithm that would support all of possible C++ (that is underlying for SystemC) coding styles.

Unfortunately this reject the main benefits of SystemC concept which enables designers with almost unlimited variety of system-level abstractions (as design patterns, interfaces, iterators etc.) built on

top of rich C++ object oriented set of constructs. It was shown here that the combination of assertions semantics with SystemC and underlying C++ semantics results in a very powerful toolset for building fully re-usable system-level models and verification units. Especially in conjunction with the Transactions Level Modeling (TLM) [18] the assertions based methodologies enter into new dimension of electronic systems abstraction, which allow easily formulate constraints for a highly complicated designs.

Considering the compilation entry architecture various attempts were made too. The CheckSyC system [9] has an external compiler of non-standard property language. This unfortunately declines possibility of using commercially available verification libraries for popular standard IP blocks.

The compilation entry architecture used in [8] is more interesting – the syntax used widely in scientific community for temporal properties specifications (as in CTL [19], PSL [11]) was emulated directly at C++ level with means of overloaded operator functions and lambda expressions [20]. In this way the properties formulas could be written together with SystemC code and compiled with standard C++ compiler. But again it is not possible to fully emulate all details of industry standard notations so such system was unable to support commercial verification libraries too.

The system presented here is not ideal too. There are limitations in errors checking while compilation of the assertions specifications, especially in respect to the referred SystemC items from formulas. To enable assertions compilation with C++ strict type checking an integration with C++ compilation system would be required. This can be addressed while further research work.

References:

[1] "*Assertion*-Based Verification", Synopsys, March 2003, www.synopsys.com / products / simulation /assertion_based_wp.html

[2] "*Components* of a Complete Assertion-based Verification Solution", Cadence, 2005, www.cadence.com/products/functional_ver/abv_dt.aspx

[3] "*Assertion*-Based Verification for Complex Designs", 0-In Design Automation Inc., January 2002, http:// www.0-in.com / whitepapers / Archer_Whitepaper.pdf

[4] *International* Standard ISO/IEC 9899:1999 "Programming languages – C", second edition, International Standardization Organization, International Electrotechnical Comission, American National Standards Insitute, 1999, 554p.

[5] *Lakos J.* "Large-Scale C++ Software Design", Addison – Wesley 1996, 896p.

[6] *Open* SystemC Initiative Consortium – www.systemc.org

[7] *International* Standard ISO/IEC/ANSI 14882:1998: "Programming Languages – C++", International Standardization Organization, International Electrotechnical Comission, American National Standards Insitute. 1998, 748p.

[8] *Große D., Drechsler R.* "Formal Verification Of LTL Formulas For SystemC Designs", IEEE International Symposium on Circuits and Systems, 2003, pp. 245-248.

[9] *Große D., Drechsler R.* "CheckSyC: An Efficient Property Checker for RTL SystemC Designs", IEEE International Symposium on Circuits and Systems, 2005, pp. 4167-4170.

[10] "*OpenVera*™ Language Reference Manual: Assertions", Version 1.4, Synopsys, April 2004, 136p.

[11] "*Property* Specification Language: Reference Manual", Version 1.1, Accellera, April 30, 2004, 131p.

[12] *Aldec* Inc., Riviera-IPT Overwiev, 2005, http://www.aldec.com/products/riviera-ipt/

[13] *Aldec* Inc., Riviera Overview, 2005, http://www.aldec.com/products/riviera/

[14] *Forczek M., Hrynkiewicz K.* "Formal Properties Evaluation", Proceedings of Programmable Devices and Systems Workshop, 2003, pp. 305-311.

[15] *Gonen E.* "PSL in Action – ABV Experience Report", PSL/Sugar Consortium Meeting in DAC 2004,http://www.pslsugar.org/papers/pm2_EyalGonenDAC04.pdf

[16] *Ho R.* "Maximizing Synergies of Assertions and Coverage Points within a coverage-Driven Verification Methodology", Proc. of DesignCon 2005, www.designcon.com / conference / 2005 /9-ta3_ho.pdf

[17] *Yeung P.* "Four Pillars of Assertion-based Verification",Euro DesignCon 2004, http://www.cs.huji.ac.il/~jarom/vlsi_seminar/papers/4%20Pillars%20of%20ABV%20Euro%20DesignCon%202004.pdf

[18] *Ghenassia F.* "Transaction level modeling with SystemC: TLM concepts and applications for embedded systems", Springer, 2005, 200p.

[19] *Jang J.-E., Moon I.-H., Hachtel G.* "Iterative abstraction-based CTL model checking", Proceedings of DATE'2000, Paris, France, 2000, pp. 502-509.

[20] *Vandervoorde D., Josuttis N.* "C++ templates: the complete guide", Addison-Wesley, 2002, 552p.

Camera-ready was prepared in Kharkov National University of Radio Electronics
by Dr. Svetlana Chumachenko and Volodymyr Obrizan
Lenin ave, 14, KNURE, Kharkov, 61166, Ukraine

Approved for publication: 05.09.2005. Format 60×841/8.
Relative printer's sheets: 33,4. Order: without cash transfer. Circulation: 100 copies.
Published by SPD FL Stepanov V.V.
Ukraine, 61168, Kharkov, Ak. Pavlova st., 311