Environment for efficient and reusable SystemC module level verification

Flavia Gonția
Siemens SRL
flavia.gontia@siemens.com

***Abstract*—In this paper a concept and SystemC implementation of a new method for module level verification is presented. Implemented as a SystemC library, the presented method also facilitates software driven verification. The concept implementation, with its advantages and limitations, is presented together with a comparison of current standard methodologies used in verification (*Universal Verification Methodology – UVM* and *SystemC Verification Standard – SCV*). For software driven verification, a few platforms were used to evaluate the performance of the new method.**

Keywords—ESL; SystemC modeling; SystemC Verification

I. INTRODUCTION

Especially in ESL design, any SystemC component needs a minimal functional verification at module level. Currently, most functional verification is performed using components described in either SystemVerilog or *e* language following the Universal Verification Methodology (UVM) or C++ following the SystemC Verification Standard (SCV). Other approaches, based on a simple usage of the SystemC library and no methodology can also be found. This happens because for minimal testing during development, standard verification methodologies tend to be too complex. In these cases the verification is done by the module's developers and often, to save time and resources, this first verification is performed inside the SystemC environment itself. Developers create simple, dedicated testbenches for particular modules and there is no constraint to approach any methodology (i.e. modules can contain particular port bindings, classes and file-structures). There is no clear separation between testbench and tests and the whole environment lacks reusability and portability. As a direct consequence, when the SystemC module is later on included within an embedded system, the test configurations cannot be reused as software running on the host CPU.

Most of the software running on embedded processors is written in C language and today, in the SoC development process, the software development is becoming more and more important. Creating software for module (SoC) verification, *software-driven verification,* has the great advantage that it can be reused during all of the different phases, from the earliest stages of design and development to physical chip validation. But, because the majority of verification environments following standard methodologies are written in other languages than C++, they do not offer the possibility to perform software-driven verification. Currently, software-driven verification is done using platforms that include a processor model. The processor runs the software and different tools are used to execute and debug this software. Moreover, for verification of each module, a different tool instance is used, the high necessary license costs creating a big disadvantage.

To overcome all of these problems, we propose a method that facilitates:

- software-driven verification with embedded software running on the host computer;
- reusable SystemC verification environment.

The proposed method was created from the embedded software perspective, so that *software-driven verification* can be easily addressed. However, it is not limited only to this aspect. Verification environments using the power of the SystemC library/C++ can be created as well. The proposed method uses a testbench structure comprising a *Device Under Test* (DUT), *pair ports* (wrappers which convert a DUT particular port type to a base port type), *General Central Unit* (which groups *pair ports*) and *tests* (accesses the *pair ports* using the *General Central Unit*). The *tests* can either implement complex scenarios or can be simple data generators, monitors etc. The method offers a high degree of reusability, tests being completely decoupled from the DUT's port types.

The paper starts with the test-bench's description, followed by the detailed description of the concept and the implemented library (SystemC General Environment Verification – SGEV) of the proposed method. A comparison between the UVM and SGEV approaches is presented and, for software-driven verification, a performance evaluation is done, based on the total execution time obtained by simulation runs on three different platforms.

II. SYSTEMC GENERAL VERIFICATION ENVIRONMENT CONCEPT

Functional verification of SystemC modules is performed using a testbench structure comprising one or more DUT(s)(the SystemC module(s) to be verified) and *tests*. The *tests* implement the basic verification components (*checkers, data generators, drivers* or *monitors*) and can address simple or complex scenarios.

Even when standard verification methodologies are used (e.g. UVM, SCV) and a high degree of verification environment reusability is ensured, in most environments there still is a dependency on DUT port interface. For instance, in UVM (or other SystemVerilog based verification environments), the *sequencers* are generating *items* carrying specific information to the DUT port interface. In most of the cases the *drivers* and *monitors* are dependent on the port interface (number and ports type).

With the proposed approach we try to decrease the dependency on DUT port interface and increase the SystemC verification environment reusability. Even from the beginning two new aspects were introduced:

- all the tests are decoupled from the DUT port interface (naming, type, data type, number, etc.)
- each base port (that is connected to DUT ports) is identified by an address space and an interrupt address

1) *Decoupling tests from the DUT port interface*

The decoupling of tests is done by converting the communication (with each DUT port, of any type), to a base type communication. Any type of communication is thus reduced to a series of read and write accesses and a series of events.

SystemC[1] and TLM-2.0[2] libraries provide a large range of port and socket types (`tlm_target_socket<>`, `tlm_initiator_socket`, `sc_in<>`, `sc_fifo_out<>`, etc.) Using these types, different performance levels can be achived by increasing the model accuracy or by shortening the execution time. In this context, decoupling the tests from port types represents a major advantage. The communication with the DUT is reduced to read/write accesses and interrupt service routines, simplifying tests and allowing also inexperienced SystemC programmers to be involved in verification.

2) *Base ports identification*

Starting from the important observation that embedded software always interacts with the "outside world" (the communication with different modules) by means of registers and interrupts, the *base ports identification* idea comes to facilitate the software-driven verification.

In the proposed method, similar to embedded software, the tests communicate with the module ports only through an address space and interrupts address list. In this way, all kind of ports accesses (whether they use an address or not) are actually transformed into an address based communication. This increases the portability and reusability, by creating a simplified and standardized way of communication. In the end, to each module port or groups of ports (no matter the type) there will be an address space and an interrupt address associated.

Based on these two restrictions, a modified testbench structure is proposed, comprising: *Device Under Test* (DUT), *Pair ports*, *General Central Unit* and *Tests* (Figure 1).

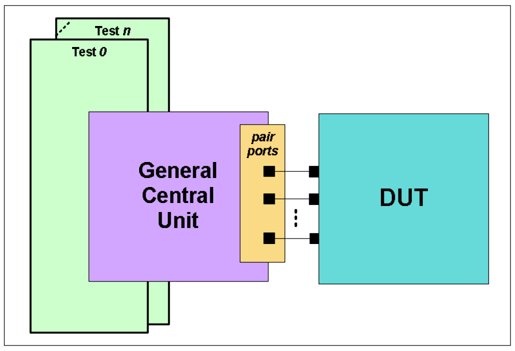

Figure 1 SystemC General Verification Environment overview

A. *Pair Ports*

Pair ports are wrappers adapting the communication from a specific port (e.g. `sc_port<>`, `sc_in<>`, `sc_fifo_in<>`, `tlm_target_socket<>`, `tlm_initiator_socket`, etc., or even a user defined port type) to a *base port* communication. The communication protocol with a specific port (or set of ports) is converted to a series of events and read/write accesses that carry data of type BASE_DATA_TYPE.

The base port defines the communication interface using the following elements:

- *Address space:* needed for *base port* identification; any read/write access to an address found inside this address space will access this port; a *base port* can be identified by multiple addresses; besides identifying the port, this information can represent the address space that the *base port* can access.
- *Interrupt address*: the address where callback routines are registered
- *BASE_DATA_TYPE*: the data type to which any data specific data must be converted
- *Read* and *Write* functions: propagate data of BASE_DATA_TYPE
- *Events*: will trigger the callback routines registered at *interrupt_address*; different callback routines can be attached to this *base port*, at *interrupt address*, on different events.

In the end, a unified communication with any *pair port* will be obtained, by means of read/write functions and interrupt callback routines, through an address space and interrupts address. The *pair ports* will be bound to the DUTs ports on one side and added to a list of base ports to the General Central Unit on the other side.

The *pair port* naming was preferred as these ports will be or will contain the mirror ports to which DUT-ports are bound. A *pair port* of an input port is an output port, and vice-versa.

If the interaction with any pair port is the same, then they can easily be grouped in lists thus different operations, or decisions for groups of ports can be applied.

B. *General Central Unit (GCU)*

The General Central Unit (GCU) groups the *pair ports* that are bound to DUT-ports. Any number of ports can be registered, from any number of DUTs. All the DUT's ports will be accessed with the help of the GCU, through *pair ports*. The GCU will implement the *base port* communication interface and, like the *pair ports*, it can propagate only data of BASE_DATA_TYPE. The Central Unit self-adapts to the containing environment. This way, no useless resources are consumed.

The GCU provides the means for *pair ports* registering. Groups in a list all *pair ports* that are registered to it. The list holds the ports map information. GCU can access all the *pair ports* that have been registered to it. It implements the read/write functions used by the tests to access the DUT in the end and is routing the data to/from the corresponding port, based on the address. GCU can route only one type of data, BASE_DATA_TYPE.

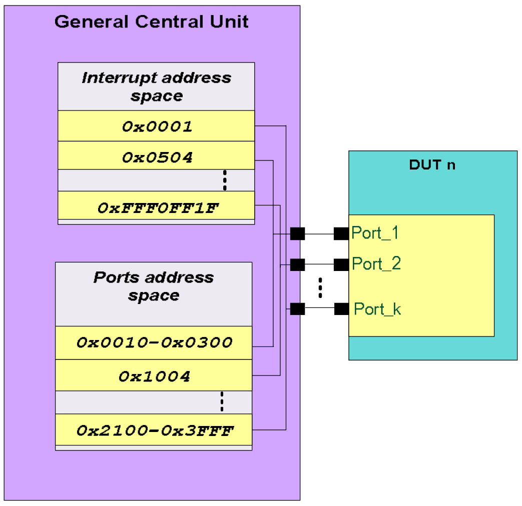

Figure 2 Example of how *ports address space* and interrupt *ports address space* are formed

The *pair ports* map contains two address spaces: ports address space and interrupt ports address space. For each GCU, the pair port map will vary depending on the number of ports, the address where the pair ports are mapped etc., will adapt to the containing environment.

The *pair ports* are accessed by performing read/write to an address that is found in *ports address space*. Callback routines can be registered to an address found in *interrupt ports address space.*

C. *Tests*

The *tests* access the DUT's ports through the General Central Unit. Only the ports that are bound to the Central Unit can be accessed.

Test perspective is reduced to:

- the GCU's ports address space and interrupt ports address space;
- communication with the DUTs using read/write functions provided by the GCU that propagates data of BASE_DATA_TYPE;
- registering callback routines to events signaled by the DUTs.

Conversion of communication with a specific port to a base communication type provides a high degree of reusability. The current method's advantages are:

- *tests* are decoupled from the DUT's port interface, becoming easily reusable;
- test-portability is increased by using a Central Unit that provides the necessary interface and means for DUT-communication; tests can be simply taken and run on a different GCU instance that groups different number of ports;
- libraries with reusable components (pair ports, data generators, drivers, monitors, tests etc.) can be created and ported to different platforms; a reusable entity has an increased usability; e.g. a random data generator of BASE_DATA_TYPE can be used to stimulate a large number of DUT-ports, since all the communication with the ports is translated to a base communication; for example in a SystemC environment a random data generator of *int* type can stimulate DUT ports of the next types: `sc_in<int/bool/char…>`, `sc_fifo_in<int/bool/char…>`, `tlm_simple_target` port etc.

III. SystemC General Environment Verification Library

A SystemC library, SystemC General Environment Verification (SGEV) that implements the presented concept was created.

The SGEV library provides base classes, macros and fully reusable self-configurable classes for building up the verification environment.

The SGEV library comprises:

- *Base port* class *<BASE_DATA_TYPE>*: defines the read/write communication functions and data members. Pair ports class will be extended from this class.
- *Base pair port interface class:* the pair ports are grouped in interfaces. The class that will contain the instances of pair ports will have to be extended from this class.
- *General Central Unit (GCU) class <BASE_DATA_TYPE>*: the pair ports will be registered to this class instance.
- *Register object class <BASE_DATA_TYPE>:* is used to handle register aliases from embedded software.
- *Base test class:* all the test classes will be extended from this class.

- *Main test class:* in this class the instances of tests or main tests that run on different or same general central units will be found.
- *Macros:* ease the creation and running of tests.
- *Pair ports classes:* pair port classes for base types of SystemC ports; currently only the communication with loosely timed models is implemented
- *Data Check Statistics functions*
- *BASE_DATA_TYPE:* can be of any type, C++ fundamental data types, structures or classes.

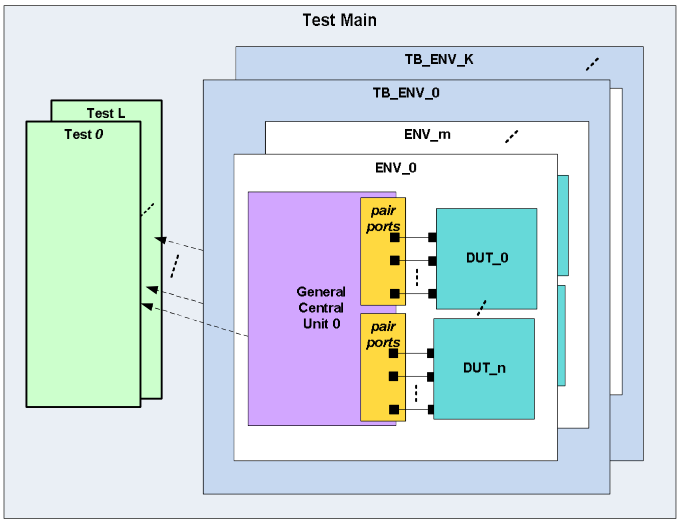

Figure 3 General example of the presented SystemC verification library

In Figure 3 is presented how the SystemC verification library can be used: the *tests* (running either in parallel or serial) communicate with all the DUTs bound to the *General Central Unit* through *pair ports.*

A. Base Port class

The *Base port* class contains the methods and fields used for the base communication interface.

The *base port* interface:

- *first address* and *last address*: delimitates the address space identifying this port;
- *interrupt address*: the address where test will register the callback routine (interrupt service routines);
- *read/write functions*: virtual communication functions; the base class has an empty implementation of this function; the *pair port*, if is needed, will overwrite this implementation of the functions; not all of the functions need to be overwritten;
- *array of events*: will trigger the service routine call; for each event an interrupt callback routine at *interrupt address* can be registered*;*
- *BASE_DATA_TYPE*: the type of data the *base port* can process/propagate.

In the current version, two types of read/write functions are defined. The first type can be used to propagate single data while the second one can be used to propagate a data array. Using this interface, the GCU can communicate with the *pair ports.* The user can extend the *base port* class type and the GCU to add other functions for communication.

B. Pair Ports classes

The *pair ports* are found at the boundary of GCU and DUT ports. In the direction GCU-to-DUT port, the *pair port* must adapt the *base* communication to *port* specific communication. On the other direction, DUT port-to-GCU, it must translate the *port* specific communication to *base* communication.

A *pair port* can communicate with one or more DUT ports. Each *pair port* must be extended from *base port class.* On one side the *pair ports* will be bound to the DUT's ports and on the other will be registered to a GCU. The *pair port* on one side will communicate with the DUT's port specific protocol and on the other side the GCU will access the data throughout base read/write functions. A higher communication protocol layer can be established and the data exchange will be performed using read/write functions and a series of events to which callback routines can be attached.

The data members and read/write functions are inherited from the *base port* class. The read and write functions are virtual functions, so the *pair port* must implement the needed logic to convert the specific port communication that it is bound to.

In the next example, a *pair port* implementation that will be bound to a `sc_out<PORT_DT>` port is presented.

```
template< class PORT_DT, class B_DT >
class  pport_sc_in:  public  SGEV_port_base<B_DT>,
public sc_signal<PORT_DT>{
public:
       pport_sc_in ()
       {
          this->set_name(this->name());
          this->set_onchange_event((sc_event*)
                (&this->value_changed_event()));
       }

       virtual bool read (unsigned long &addr,
                            B_DT &data ){
          PORT_DT temp_data;
          temp_data=   sc_signal<PORT_DT>::read();
          data = (B_DT)temp_data;
          return true;
       }

 };
```

The pair of an output port is an input port. To reduce the effort at port binding, the `SGEV_sc_in` was extended from `sc_signal<>` class. This *pair port* being an input port, only the read function is implemented. A write to this port is not allowed.

The user can implement its own pair ports to process the needed data type. The *base port class* provides means to create an array of events of any size. A pair port can also play the role of a complex Driver or Monitor. In these cases, the pair port will most probably have to drive more than one port. For this approach, based on a protocol, the data is ported to the DUT or extracted from the DUT and will be provided to base read and write functions.

For the highest tests and testbench reusability degree, it is recommended for each DUT-port to have a corresponding *pair port*. In case of complex communication protocols, the drivers or monitors should communicate with the DUT ports throughout the GCU. Similar to *tests*, the *pair ports* can be found also on the right side of the GCU and can be registered to the GCU. More exactly, they can be extended from the *base port* class and can communicate with the DUT-ports throughout the GCU. In this case, the reusability of the testbench increases also, as both the driver and the monitor are decoupled from the DUT port interface type. The same driver or monitor can be re-used to communicate with different types of ports (e.g. in some cases `tlm` sockets in other cases `sc_fifo<>` ports type). In this case the *test* can provide data to the *driver* and can also take decisions in case one or more ports' state has changed.

The SGEV library implements *pair ports* for all basic SystemC ports: `sc_fifo_out<BASE_DATA_TYPE>`, `sc_in<BASE_DATA_TYPE>`, `sc_out<BASE_DATA_TYPE>`, `sc_fifo_in<BASE_DATA_TYPE>`. There are also *pair ports* implemented for `tlm_target_socket` and `tlm_utils::simple_initiator_socket`[2].

Using BASE_DATA_TYPE as primitive data types (int, bool, etc.) increases the reusability degree, as a larger number of use cases are covered.

C. *Pair Port Interface class*

Each user *pair port* interface must be extended from the *pair port interface* base class. The *pair port interface* base class facilitates grouping and registration of *pair ports* that are bound to the DUT. By grouping the pair ports inside an interface class, the SGEV verification environment's portability to another platform is increased.

At port registration, both the address space and the interrupt address are set. Also, a flag is set establishing if the port will be accessed or not by an embedded software. For a certain interface it is recommended that the address spaces for all ports are set relatively to address zero. Later on, the *pair port* interface can be registered to the GCU at a different offset address. For example, if there are two UART entities inside the same platform, the *pair port interface* is instantiated twice and each interface is bound to one UART module; then the two interfaces are registered to the GCU at different offset addresses.

An example of a *pair port interface* with two *pair ports* is presented below. Port `reg_init_sockt` is registered at [0, 0xFD] address space and there is no address associated to the interrupt. This port will be accessed (*`IF_CPU`*) by embedded software, but the second port, `in_serial`, will not be (*`IF_NOT_CPU`*):

```
//Uart module class – port interface.

class uart: public sc_module
{
public:
   // TLM Socket (BUS Interface )
   tlm::tlm_target_socket<32>  p_reg_targ;
   sc_fifo_out<unsigned char>  p_serial_out;

   SC_HAS_PROCESS(uart);
    uart(const sc_module_name & _n):
       sc_module(_n)
       , p_serial_out("p_serial_out")
       ...
     {  ... }
      ...
};

//Pair port interface for uart module.

template<class B_DT>
class if_uart_pair_ports: public
SGEV_if_base<B_DT>{
 public:

    SGEV_init_socket<32,B_DT>    reg_init_sockt;
    SGEV_sc_fifo_in<unsigned char,B_DT> in_serial;

    //constructor
    if_uart_pair_ports(sc_module_name & n):
       SGEV_if_base<B_DT>(n)
      {
         //register the initiator port
         SGEV_REGISTER_PORT(reg_init_sockt,0,
                            0XFD, ADDR_NOT_USED,
                           IF_CPU);
         SGEV_REGISTER_PORT(in_serial ,
                            1 , ADDR_NOT_USED,
                            1, IF_NOT_CPU)
       }

   };
```

It is recommended for *pair ports* to be grouped depending on the interface or communication interface they serve. For example, a DUT with one part of the ports communicating to a CPU (register access and interrupts) and one part communicating with an UART serial interface.

D. *The General Central Unit class*

The main role of the *General Central Unit* (GCU) is to facilitate the tests' communication with the DUTs through *pair ports* registered to it. The test will have visibility to all the *pair ports* that are registered to the GCU.

The *General Central Unit's* main features are:

- provide the functions for *pair port interface* or single port registration; the *pair ports* are added to an internal *base port* list; this list adapts then to the environment and increases with the number of added base ports; it represents the mapping address of the ports;

- implements an interrupt vector list; this list changes dynamically based on the interrupt call routines that are registered inside tests; if during simulation no callback routine has been registered to any *base port*, the interrupt list remains empty;
- implements read (BASE_DATA_TYPE) and writes (BASE_DATA_TYPE) functions, inside which the data is routed to the corresponding port, based on the address;
- implements the means (used by tests) to register interrupt callback routines on pair ports events;
- provides synchronization mechanisms with the SystemC Scheduler.

A major problem which had to be overcome was the synchronization with the SystemC scheduler. This synchronization is needed especially by embedded software. Register pooling mechanism (continuously interrogating a register) is often used inside a while-loop until its value is changed. In such cases, if synchronization with the SystemC scheduler is not performed, the software will most probably enter into an infinite loop. The SystemC library implements a non-preemptive scheduler that runs at most one thread at a time. If the thread does not give the control back to the scheduler, the simulation will block inside this thread[1].

The GCU port accesses (read, write or objects registering) perform the synchronization with the SystemC scheduler. Sequencing the register accesses is also ensured, each access being performed in a different delta cycle.

It is recommended that DUT-ports binding (to *pair ports*) and *pair ports* registration (to GCU) be grouped inside a testbench environment class.

A code example of the testbench environment:

```
template<class B_DT>
class tb_uart_env {
public:
    //GCU instance
       SGEV_general_cu<UC_DATA_TYPE>  i_gcu;
private:
       uart                          i_uart;
       if_uart_pair_ports<B_DT>  if_pair_ports;

       tb_uart_env (const char* name):
           i_if_pair_ports("if_pair_ports")
       {

           //Bind the UART tlm target socket
           //to pair port intiaotor socket.
           i_uart.p_reg_targ.bind
                   (if_pair_ports.reg_init_sockt);

           //Bind the Uart serial output to
           //input pair port in_serial.
          i_uart.p_serial_out
                   (i_if_pair_ports.in_serial);

       }
       void end_of_elaboration(){
              // Register the interface to GCU.
              i_uc.register(if_pair_ports);
       }
   };
```

E. User-Defined Tests

Tests will access the *pair ports* throughout GCU read/write functions. Each test must be extended from the base test class provided by the SGEV library. The tests are split in two types: the first type is used only for embedded tests (IF_CPU tests) while the second type (IF_NOT_CPU) can be used for any desired implementation (including embedded tests). The embedded tests are allowed to access only the *pair ports* previously registered with IF_CPU flag. In case of an attempt to access a port having IF_NOT_CPU set, an error is generated. This measure is taken to avoid problems when embedded software is ported to a processor. At each test instantiation the reference of the GCU instance (used to access the *pair ports*) is transmitted. The tests will be able to access the *pair ports* registered to the transmitted GCU reference.

There are macros in place to map the interaction with the GCU. User explicit function calls through general central instances are not needed.

The following example shows a test for general behavior (not for embedded software):

```
// Test handle serial out port from i_uart.
// "main_io":function name, from where the test
//            starts to run
// test_out_uart: test class name.

#define UART_SERIAL_OUT_INT_ADDR 0
#define UART_SERIAL_OUT_ADDR  0

SGEV_TEST_NOT_CPU(main_io, test_out_uart)
  //Receive data from UART
  //This function will be called when a

  //char has been written to
  //i_uart.p_serial_out port.
  void rx_uart_int(){
     unsigned int data;
     //Read data char from
     //i_uart.p_serial_out port
     rd_from_addr(UART_SERIAL_OUT_ADDR, data);
     //Print to console the received char
       cout<<(char)data;

  }

  // main function
  void main_io(){
     // Register call back interrupt function
     //for i_uart.p_SerialOut port
     register_int_func (UART_SERIAL_OUT_INT_ADDR,
                                  &TYPE_ENV rx_uart_int);
  }
};
```

F. *Embedded software*

Embedded software can be written with the help of the SGEV library. Communication with DUT-ports is performed by means of read or write operations to a given address. This allows embedded software development. In the SGEV environment the embedded software runs on the host computer.

Usually, inside embedded software a register is referenced using a name (e.g. UART_DR, I2CSTAT etc.). The *SGEV* environment allows referencing of ports using a given name. The SGEV library provides a register object class which can be used in case of *register alias* naming.

When creating embedded software, the following restrictions must be considered:

- Assembler code cannot be used because the assembler instructions are processor specific; the code will run on the host computer which most probably will have a different instruction set;
- At interrupt service routine registration TYPE_ENV directive must be used before function name. E.g. : `register_int_func (UART_SERIAL_IN_INT_ADDR, &TYPE_ENV rx_uart_int)`
- For register definitions a number of *define* directives must be used for both register type and register cast; the SGEV_REG_TYPE is used to define the register type while SGEV_REG_CAST_TYPE must be used when cast to register type is needed.

  E.g.:
  ```
  #ifdef IS_SGEV_ENV
     #define SW_REG_TYPE          SGEV_REG_TYPE
     #define SW_REG_TYPE_CAST     SGEV_REG_CAST_TYPE
  #else
     #define SW_REG_TYPE    volatile unsigned int*
     #define SW_REG_TYPE_CAST   (SW_REG_TYPE)
  #endif
  ```
- In case of infinite loops or waiting for a variable to change inside of an interrupt service callback routine, synchronization points must be added. SGEV_NOP define directive must be used for achieving the synchronization with the SystemC scheduler.

  E.g.:
  ```
  #ifdef IS_SGEV_ENV
    #define DO_NOTHING     SGEV_NOP
  #else //
    #define DO_NOTHING
  #endif
  ```

Already written embedded software can be also adapted to run in the SGEV environment. The code must be modified to follow the above described restrictions. The effort to adapt the software depends on how well the code is organized, if lots of synchronization points are needed etc.

The next code represents an embedded software test example:

```
              //file  uart_test_emb.h
//Register type defines. If this file is found in
//SGEV env SW_REG_TYPE is allready defined at this
point
//with SGEV_REG_TYPE
#ifndef SW_REG_TYPE
   #define SW_REG_TYPE     volatile unsigned int *
   #define SW_REG_TYPE_CAST     (SW_REG_TYPE)
#endif

#ifndef DO_NOTHING
   #define DO_NOTHING
#endif

#define Uart_BASE 0x100
#define UART_DR  *SW_REG_TYPE_CAST (UART_BASE + 0x0)
#define UART_FR  *SW_REG_TYPE_CAST (Uart_BASE + 0x18)

void init_uart () {
   SW_REG_TYPE UART_CR = SW_REG_TYPE_CAST
                         (Uart_BASE + 0x14);
   // Enable uart and disable rx and tx
   //interrupts.
   UART_CR  = 0x01;
}

void wr_to_uart (unsigned char value) {
       while ((UART_FR & 0x20) == 0x20);
   // write to UART
   UART_DR = value;
}
void main_uart(){
   unsigned char i = 0;
   //enable uart
   init_uart();
   //write data to uart
   for( i = 0; i<10; i++)
      wr_to_uart(i);

   while(1){
       //synchronize with SystemC scheduler
     DO_NOTHING
   }
}
```

```
            //file  test_uart_sc_emb.h
//Set defines used in SGEV environment
#define SW_REG_TYPE        SGEV_REG_TYPE
#define SW_REG_TYPE_CAST   SGEV_REG_CAST_TYPE
#define DO_NOTHING         SGEV_NOP

SGEV_TEST_CPU (main_uart, test_uart_sc_emb)
     //include the embedded software file
       #include "uart_test_emb.h"
};
```

G. *Main test*

Inside the *main test* the tests and testbenches containing the GCU are instantiated. The *main test* must be extended from the base *main* class. The base class provides mechanisms to run tests in parallel or serial and to stop the simulation when tests are finished. There can be any number of testbenches and *test* instances that have different types as BASE_DATA_TYPE.

A *test* can be launched to run sequentially or in parallel with the following macros:

- SGEV_RUN_TEST_S (&<test_instance_name>);
- SGEV_RUN_TEST_P (&<test_instance_name>);

The *main test* can be used as environment for running regressions.

```
template<class B_DT>
class test_uart_sc_main : public
SGEV_test_main_base{
public:
   uart_tb_env<B_DT> i_uart_tb_env;
   test_uart_sc_emb<B_DT> *i_uart_test_emb;
   test_out_uart<B_DT> *i_uart_out_test;
   SC_HAS_PROCESS(test_uart_sc_main);
   test_uart_sc_main(sc_module_name & n):
       SGEV_test_main_base(n)
       ,i_uart_tb_env("i_uart_tb_env")
   {
       //Instantiate the tests
     i_uart_test_emb = new  test_uart_sc_emb<B_DT>
            ("i_uart_test", &i_uart_tb_env.i_gcu);
     i_uart_out_test = new test_out_uart<B_DT>
         ("i_uart_out_test",&i_uart_tb_env.i_gcu);
   }
   void main(){
     //Launch test that receives data from uart
     //p_serial_out and prints to cout
     SGEV_RUN_TEST_PARALLEL(i_uart_out_test)

     //Launch the sc test main that includes
     //the embedded software
     SGEV_RUN_TEST_PARALLEL(i_uart_test_emb)
   }
};
```

IV. SGEV In Comparison With UVM

A comparison can be done, from the user's point of view, between SystemVerilog based UVM and SGEV library. The analysis is done for each component that a user must create for a functional verification environment.

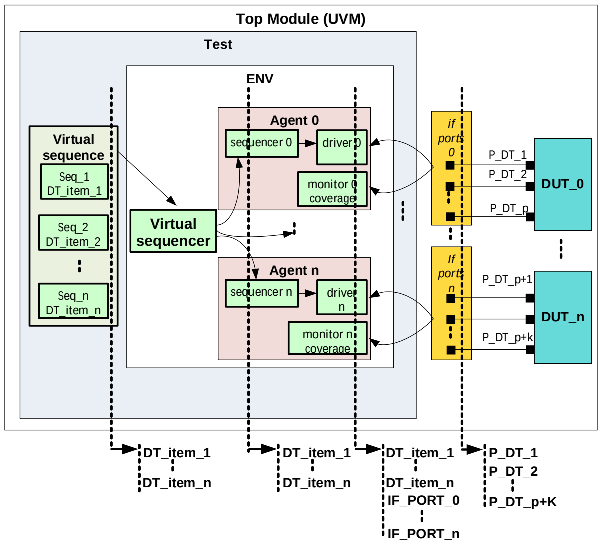

Figure 4 Example of SystemVerilog UVM verification environment

The main components of a verification environment according the UVM methodology, as they are defined in [4]:

- *Data Item (Transaction):* data transferred to the DUT, e.g. packets, bus transactions, instructions;
- *Driver (BFM)*: active component which provides stimulus (data items) by driving the DUT signals;
- *Sequencer*: generates sequences of transactions and send them to the Driver to be applied to the DUT;
- *Monitor*: passive component which samples and checks DUT signals as well as collects functional coverage information;
- *Agent*: higher level entity which contains a driver, a sequencer and a monitor;
- *Environment*: the highest level verification entity, contains agents, additional bus monitors, configurations or other environments;
- *Sequences:* streams of transactions or operations for DUT;
- *Top module*: inside the top module, the virtual interfaces are bound to DUT-instances and the start simulation is triggered.
- *Test:* defines the test scenario.

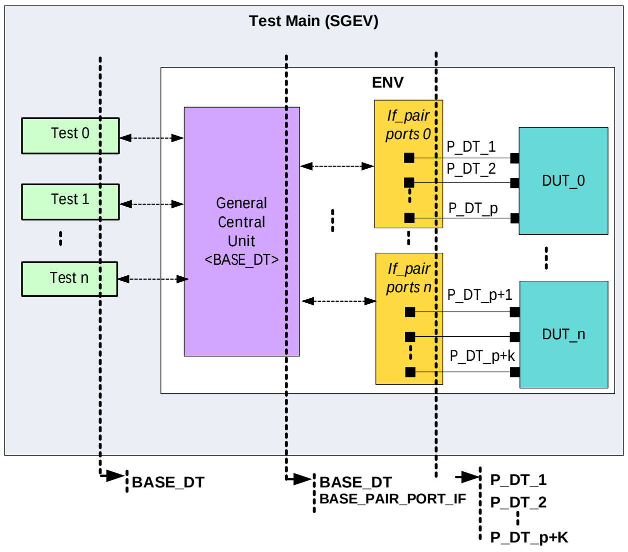

Figure 5 Example of SGEV verification environment

The main components needed to create a verification environment using SGEV (as in Figure 3):

- *Pair Port Interface: contains the pair ports to be bound to DUT; the pair ports are provided by the SGEV library;*
- *TB Environment:* contains the *pair port interfaces*, DUT(s), GCU(s) instances *and* the corresponding connection between elements; the GCU is provided by the SGEV library;

- *Tests:* contain the logic used for stimulating and checking the DUT(s);
- *Main test*: contains the testbench environments and tests instances.

A major difference between UVM and SGEV approach is the decoupling of verification environment components from DUT. In SGEV, starting from DUT port interface, the environment components dependency on data types is decreasing to a single data type (base data type – BASE_DT as presented in Figure 5). Because there is only one type of data to be propagated to DUT, different reusable components can be created (e.g. GCU, PAIR PORTs, Test as random data generators etc.). These components can then be used without additional effort.

On the contrary, in the UVM approach (Figure 4) the dependency remains at higher levels. Most of UVM components are dependent on data item type (DT_item_1, … DT_item_n) and/or interface type (IF_PORT_1, IF_PORT_2, … ,IF_PORT_n), hence a reduced portability and reusability than in SGEV. For instance, importing a single sequence to generate data items of different type, implies modification of test bench modules (at least virtual sequence and driver).

Table 1 UVM System Verilog and SGEV library user perspective

Using UVM System Verilog library			**Using SGEV library**
DUT virtual interface			Pair Port Interface
Agent	Driver		-
	Monitor	Checks	
		Coverage	
	Sequencer		
Environment			TB environment
Sequences			Tests
Test			
Top			Main Test

As shown in the Table 1, for a simple verification environment, the user has only to create a reduced number of components in SGEV in comparison to UVM.

For a full verification environment (including functional coverage) the UVM methodology approach is a better choice. But using this methodology for a minimal functional verification of a SystemC module is time consuming and involves many resources.

Usually, for a SystemC loosely timed model, complex bus communications are modeled as TLM (*Transaction Level Modeling*) communication. In these cases there is need for neither complex driver nor monitor implementations. Binding a pair TLM socket to this type of port is enough to establish a communication.

One of the SGEV's advantages is exactly the reduced time needed to get familiar with the environment. As opposite, understanding a complex UVM environment (the mechanism behind, the virtual sequences, etc.) can be a time consuming activity, especially for inexperienced users.

Another major advantage in comparison to UVM is the possibility offered by the SGEV-approach to create embedded software which can then be used during all SoC development process phases.

V. SGEV PERFORMANCE ANALYSIS

In order to evaluate the new method's performance, a use case running in three different environments was analyzed. For each environment a different SystemC platform was necessary. Eclipse[5] and Synopsys' Virtual Prototype Analyzer (VPA)[6] tools were used for simulating the environments.

The presented use case contains embedded software that configures and implements the communication protocol for receiving and sending Ethernet frames to an Ethernet Media Access Controller (Ethernet MAC controller). The embedded software must configure an UART module to which debug messages are sent. The data received by UART is printed to a terminal.

To ease further descriptions, the three used platforms are named as follows: VPA_ETH_MAC, SGEV_ETH_MAC and SGEV_VPA_ETH_MAC.

A. VPA_ETH_MAC platform

This platform contains a SytemC *cortex m3* processor. The embedded software was first written for this platform.

The VPA_ETH_MAC platform runs in VPA tool and the software is compiled for *cortex m3* (the obtained image is used). It comprises the following SystemC modules:

- *cortex_m3*: CORTEX M3 processor model;
- *d_ram*: data code memory;
- *i_ram*: instruction code memory;
- *s_ram*: SDRAM memory (used to intermediate the communication between the processor and the Ethernet controller);
- *eth_mac*: Ethernet MAC controller;
- *uart*: UART module used for printing debug messages;
- *t_display*: used for displaying on the terminal data received from UART;
- *i_intconect_bus*: interconnect bus module (used to interconnect all the platform modules).

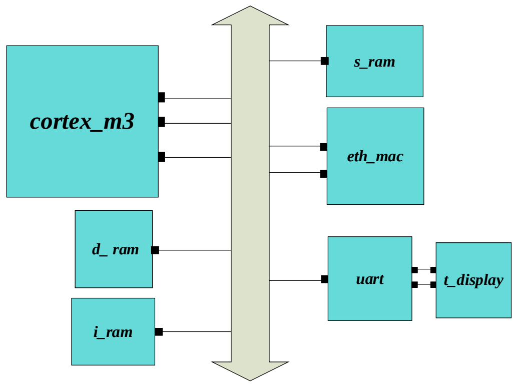

Figure 6 VPA_ETH_MAC platform

The *cortex m3* is connected to *i_intconect_bus* using three ports: p_system_port, p_i_code and p_d_code. The communication with *eth_mac*, *s_ram* and *uart* is performed through p_system_port.

B. SGEV_ETH_MAC platform

The embedded software was first written for VPA_ETH_MAC platform. To use it in the SGEV environment the embedded software had to be adapted. Because in SGEV there is no processor module present and the software runs on host computer, the data ram and instruction ram modules are not needed.

The SGEV_ETH_MAC platform runs in Eclipse tool and uses the C files implementing the software (which runs on host computer). It comprises the following SystemC modules:

- *pair_port_if*: contains three *pair ports* used by the GCU to communicate with *s_ram*, *eth_mac* and *uart* modules; each *pair port* is registered with corresponding address space (used by the embedded software to access these modules)
- *i_GCU*: the General Central Unit to which the the pair port interface is registered; the GCU will also be used as interconnect module;
- *emb_test*: embedded test that contains the embedded software C files; receives the reference of *i_GCU;*
- *s_ram*: SDRAM memory used to intermediate the communication between *emb_test* and the Ethernet controller (*eth_mac*);
- *eth_mac*: Ethernet MAC controller;
- *uart*: UART module used for printing debug messages;
- *t_display*: used for displaying on the terminal data received from the UART
- *i_sram_intercont*: interconnect bus module (used to interconnect *emb_test*, *eth_mac* and s_ram).

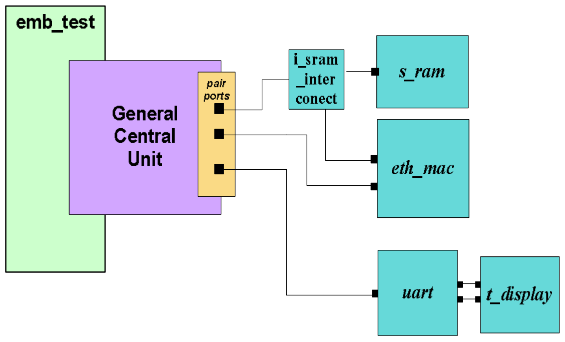

Figure 7 SGEV_ETH_MAC platform

C. SGEV_VPA_ETH_MAC platform

This platform was created by removing the *cortex_m3* processor module from the VPA_ETH_MAC platform and replaced with *pair port* interface, GCU and embedded test from the SGEV_ETH_MAC platform.

The *pair port* interface has suffered modifications: the address space initially associated separately (for s_ram, eth_mac, uart pair ports) is here associated to only one pair port (the *cortex m3* system bus port). This modification was necessary as *cortex m3* uses only one port to communicate with SRAM and peripherals.

The SGEV_VPA_ETH_MAC platform runs in VPA tool and the C files implementing the software (running on host computer) are used. The platform comprises the following SystemC modules:

- *pair_port_if*: contains one *pair port*; in this case the GCU uses a single port to communicate with *s_ram*, *eth_mac* and *uart*; each *pair port* bound to *s_ram* is registered with its corresponding address map (used by the embedded software to access these modules);
- *i_GCU*: the General Central Unit to which the pair port interface is registered; the GCU will also be used as interconnect module;
- *emb_test*: embedded test that contains the embedded software C files; receives the reference of *i_GCU*;
- *d_ ram*: data code memory will not be used
- *s_ram*: SDRAM memory used to intermediate the communication between *emb_test* and the Ethernet controller (*eth_mac*)
- *i_ram*: instruction code memory will not be used;
- *eth_mac*: Ethernet MAC controller;
- *uart*: UART module used for printing debug messages;
- *t_display*: used for displaying on the terminal data received from the UART
- *i_sram_intercont*: interconnect bus module (used to interconnect *emb_test* , *eth_mac* and s_ram).

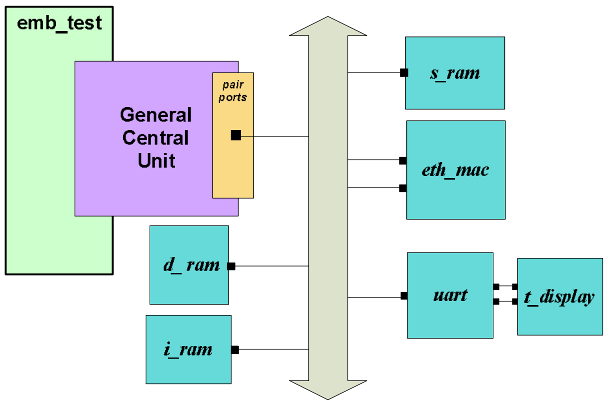

Figure 8 SGEV_VPA_ETH_MAC platform

The *cortex m3* has three bus interfaces: system bus, instruction data bus and data bus. The communication with

SRAM and the other peripherals is performed only throughout the system bus. So, in this case, there is only one *pair port* registered with all the address space. This address space is used by the embedded software to access *s_ram*, *eth_mac* as well as *uart modules*. This is an example showing the big advantage of decoupling the *test* from the ports interface. In one platform (SGEV_ETH_MAC) the test communicates with the design using 3 ports and in the other one (SGEV_VPA_ETH_MAC) using only one port.

In the next table the time execution results are presented for all the three platforms, for different simulation time:

Table 2 **ETH_MAC** Platforms execution results

Simulation time	**Execution time**		
	SGEV_ETH_MAC	**SGEV_VPA_ETH_MAC**	**VPA_ETH_MAC**
5 s	14.05 s	15.48 s	46.64 s
7 s	16.81 s	17.97 s	97.56 s
10 s	18.07 s	19.36 s	171.19 s
20 s	24.55 s	26.17 s	429.73 s
30 s	29.05 s	31.2 s	641.8 s

VI. CONCLUSIONS

As previously shown, a SystemC library was implemented (only the communication with loosely timed models implemented at the moment), and promising results were obtained. The most notable benefits offered by the newly implemented library are as follows:

- faster ramping-up of the verification environment;
- reusable and portable verification environment (unified structure, easily ported tests between hierarchical levels or modules);
- embedded software for the host computer (with a few restrictions) can be written for testing purposes;
- already written embedded software can be adapted and run in this environment;
- SystemC/C++ tests can be used together with embedded tests;
- tests can be reused as embedded software for SystemC processor model;
- *tests* and the *General Central Unit* can temporarily replace a basic SystemC processor model or a SystemC module;
- an environment for regression tests can be created with reduced effort;
- the environment is OSCI compliant as well as vendor independent.

Also, converting any type of communication to a simple set of access functions and interrupt callback routine functions can be easily done without advanced SystemC/C++/OOP knowledge.

Several limitations are predicted though. One such limitation could be the 64 bits of address. To overcome this limitation, for pair ports that are of IF_NOT_CPU type, a code can be associated instead of an address.

A downside of this approach, compared to tools, is embedded software debugging. Most of the tools provide the possibility of software debugging at assembler code level.

Future work for the SGEV library:

- to simplify end-user work, the address map of the IF_NOT_CPU ports type will be generated automatically (in the background). This will not be possible for IF_CPU as these ports are accessed by embedded software running on system having a particular memory map.
- get the SGEV library and SCV[5] library working together. Include random data generators that use SCV library for randomization, constraints and weights for randomization.
- implement pair ports for approximately and exactly timed ports.

REFERENCES

[1] IEEE Standard SystemC® Language Reference Manual IEEE 3 Park Avenue New York, NY 10016-5997, USA 31 March 2006 IEEE Computer Society Sponsored by the Design Automation Standards Committee

[2] OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL

[3] SystemC Verification Standard Specification Version 1.0e Submission to SystemC Steering Group May 16, 2003 Written by the Members of the SystemC Verification Working Group

[4] Universal Verification Methodology (UVM) 1.1 User's Guide May 18, 2011

[5] http://www.eclipse.org

[6] http://www.synopsys.com/SYSTEMS/VIRTUALPROTOTYPING