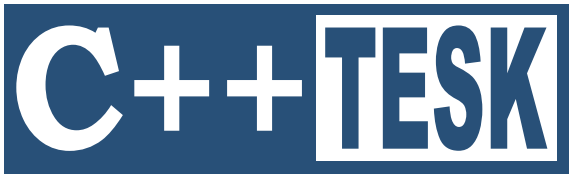

C++TESK Testing ToolKit: **Getting Started**

Version 1.0, 04/09/2013

C++TESK Hardware Extension tool is included into C++TESK Testing ToolKit which can be downloaded from the page http://forge.ispras.ru/projects/cpptesk-toolkit.

Please let us know about your proposals and problems while using C++TESK Testing ToolKit sending them to cpptesk-support@ispras.ru. The forum http://hw-forum.ispras.ru can be also used for such a purpose.

Contents

Introduction

Hardware verification is usually understood as the process of checking *behavior* of hardware on conformity to its *specification*. Such a process can be done formally by means of, e.g., model checking, automatic theorem proving, etc. Also, verification can be done by means of *simulation* of separated hardware modules with the help of *simulator*.

Accounting the complexity of hardware models under verification, the task of automation should have usually been solved before the actual verification. The more processes will be done automatically and the less manual labor will be needed, the more effective check will be made. Without touching upon the formal verification methods, in this course we will focus only on simulation-based verification. Moreover, we will further speak only about one of the existing verification tool, created in the Institute for system programming of RAS. The tool's capabilities allow speaking about it as a powerful and quite modern solution. So, we will speak about using *C++TESK Testing ToolKit* (or C++TESK for short).

C++TESK implements simulation based approach to verification. The main element of the tool is its *core library*, implemented in programming languages C and C++. All core components are arranged in one package and are available at http://forge.ispras.ru/projects/cpptesk-toolkit/files. The tool is designed for creating test systems using C++ for different models of *synchronous hardware* at different *levels of abstraction*. Test systems are created using any means, provided by C++, basing on the approach, macros and classes defined by C++TESK.

When creating test systems for simulation based verification, three main tasks are usually solved. The first one is *test sequence construction*, the second one is *checking of behavior correctness*, and the third one is test completeness estimation. C++TESK allows construction test sequences of two types: selection *random stimulus set* from the previously described stimuli at each simulation cycle or *selective choice* of stimuli based on techniques of *exploration* of implicitly defined *FSMs*. Checking of behavior correctness is made at each simulation cycle by means of *executable reference model*, created by verification engineer at some level of abstraction. External model (e.g., system simulator) can also be used. *Test completeness* is determined either by *the number of testing cycles* for randomly selected stimuli, or on the basis of the information about *completeness of FSM exploration*.

Common scheme of test system is represented in figure 1.

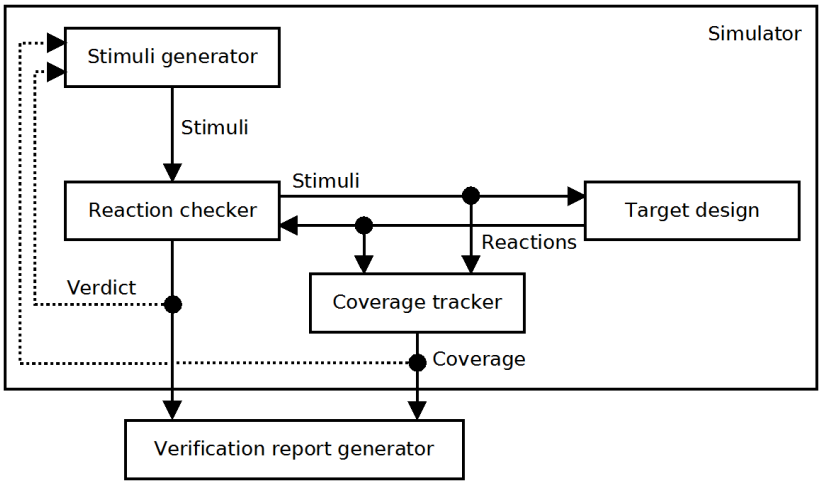

Figure 1. Generalized structure of test system

Let us shortly describe elements represented at the common scheme.

1. *Stimulus generator* is a component making test sequence (stimuli sequence). It is adjusted by test scenarios;

2. *Test oracle* is a component receiving data flows from stimulus generator and target component, sending stimulus flow from generator to target component, estimating correctness of target system behavior;

3. *Target system* is a hardware model, developed at one of hardware description languages (here, in Verilog), receiving stimulus flow and responding to it by reactions which should be checked;

4. *Coverage tracer* is a component grabbing information about reference model functional coverage, which in general affects stimulus generator work (e.g., by information of reached coverage);

5. *Test report generator* is a component making reports (test traces) with information of traversed transitions, reached coverage, found errors, etc.

The following tasks will be overviewed below.

- Analysis of documentation for development of C++TESK test systems;
- Development of test oracles including reference models and their adapters;
- Definition of test coverage;
- Setting up stimulus generator;
- Verification itself.

Documentation analysis for development of test systems

Under *verification* we mean a process of checking observed behavior against specification. In the other words, verification is an establishing of the correspondence between target system behavior and its specification. Therefore, we should have not only the target system, but its *specification* too, being a document written at the beginning of target system development, slightly modified during development process, and containing information of target system functionality.

In reality specification is often very poor or even absent, and target system developers might have written only lists of *input* and *output interface* signals. In this case to conduct verification is difficult as to speak about bug in target system is possible only having information about correct behavior. Verification engineers have to interview target system developers, making list of requirements which are obligatory for target system. When list of requirements (specification) is obtained, verification can be started.

Practice shows that specification (especially, cycle accurate) is convenient to represent in the following ways.

1. by means of *block diagrams* (see figure 2).

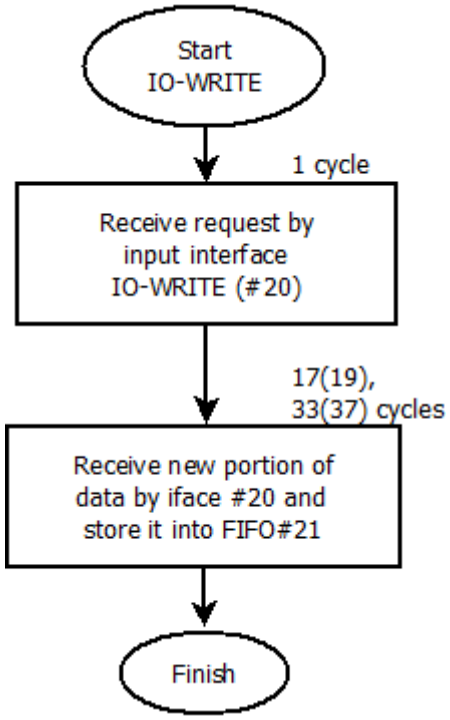

Figure 2. Example of block diagram for single operation

Block diagram allows describing reference model behavior with any proximity to cycle accurateness. Figure 2 contains abstract representation of operation IO-WRITE (write via IO channel), starting from one-cycle request by the 20th interface. At the following cycle, device starts to receive data, taking 17, 19, 33 or 37 cycles depending on data length and presence of mask among sent data. This block diagram is insufficient for usage in reference model development and is supplied with *chart of cycle accurate operation description*, which should be described below. Operation finishes after receiving and saving of all data.

The following notations are used in block diagrams.

- Oval symbols mean points of operation start and stop; start point contains mnemonic designation of operation to be executed in this control flow path;
- Rectangle symbols mean blocks taking exactly one cycle for execution at current abstraction level. Process of reference model functioning is always described inside of such blocks. Block is allowed to be supplied with information about real execution time, other auxiliary information facilitating binding the scheme to reference model.
- Rhomb symbols mean branches in control flow. The symbol should contain condition of branching inside. There should be only one ingoing path and two outgoing paths;
- Forking of control flow is represented by bold point with two or more outgoing paths;
- Merging of several control flow paths is made also in bold point, meaning synchronization of control flow paths (waiting for the slowest one) to proceed to the outgoing path.

2. by means of charts of *cycle accurate operation description* (see table 1).

Table 1. Example of chart of cycle accurate operation description

Stimulus	Branch 0	Microoperation 0	Microoperation 1	Branch 1	Branch 2	Microoperation 2	Microoperation 3
PRE: 1) only one operation of the type may be executed simultaneously 2) **val_wr_data_buff_ nreg_to_IO(o)=1**	**BRANCH:** if val_mask=1 (in stimulus) microoperation 0 is started, else microoperation 1 is started.	**PRE:** -	**PRE:** -	**BRANCH:** if wr64=0 (in stim.), operation is done, else branch 2 is started.	**BRANCH:** if val_mask=1 (in stimulus), microoperation 2 is started, else micro operation 3 is started.	**PRE:** -	**PRE:** -
Set stimulus parameters: **val_wr_data_ from_IO (i)=1** (strobe) **wr_data_ from_IO[15:0](i)={0-7}** Stimulus takes one cycle.		**wr_data_ from_IO[15:0] (i)** contains mask. It is written into **buffer(21)** (with capacity of 4 32- or 64-bytes words). This micro operation is repeated once.	**wr_data_ from_IO[15:0] (i)** contains 2 data bytes. Data are written into **buffer(21)**. This micro operation is repeated 16 times.			**wr_data_ from_IO[15:0] (i)** contains mask for the higher part of 64-byte transmission, being written in **buffer(21).** This micro operation is repeated once.	**wr_data_ from_IO[15:0] (i)** contains 2 bytes of the second part of 64-byte transmission. Data are written in **buffer(21)**. This micro operation is repeated 16 times.
		POST: next cycle after **val** signal **wr_data_buff_ nreg_to_IO (o)** (being written item in (21)) changes to the number of the following free item or to 8 if buffer (21) is full	**POST:** next cycle after **val** signal **wr_data_buff_ nreg_to_IO (o)** (being written item in (21)) changes to the number of the following free item or to 8 if buffer (21) is full			**POST:** -	**POST:** -
INPUT: **val_wr_data_from_IO**(i) - strobe **wr_data_from_IO[15:0](i)** includes - [15:3] - reserved; - [2] **A5** – write to the high 32-bytes of 64-byte item flag; - [1] **wr64** – 64-byte transmission flag; - [0] **val_mask** – mask flag.		**INPUT**: **wr_data_ from_IO[15:0]** - mask	**INPUT:** **wr_data_ from_IO[15:0]** - data			**INPUT**: **wr_data_ from_IO[15:0]** - mask	**INPUT:** **wr_data_ from_IO[15:0]** - data

Such a table represents target system behavior in cycle-accurate manner. Information in the table corresponds to block diagram and used for the following development of cycle-accurate reference model. The table is created for each operation, performed by target system. The column with stimuli comes first, where information about operation preconditions and set input signals should be written. There are four types of the other columns according to block diagrams: microoperation (one cycle of work at current level of abstraction), branching, forking, joining. The table supposes left-to-right execution of operations: the stimulus runs first, and then the following columns run one by one. Each microoperation contains information about its precondition, correspondent actions of reference model, postcondition to be checked after execution, used input and output signals. Each branch item has a condition and numbers of columns for jumps. Each fork and join items are supplied with numbers of input and output columns.

If cycle accurate reference model is not made in particular case (for example, it is taken from system simulator) or the model is abstract, specification in simple text form is enough. Therefore, if they exist in such a form, any additional representation of specification is not necessary.

Let us proceed to analysis of requirements in context of usage C++TESK. To make C++TESK's components «*reference model*» and «*reference model adapter*», *input* and *output interfaces* should be created. Each being verified unit has both input and output signals, which can be grouped in interfaces by their belonging to certain type of activity: writing or reading. Functionally complete sequence of interface call, leading the unit to some stationary state where the unit can stay infinitely, will be called an *operation.* Notice, that signals like CLK and RST do not usually belong to any interface. Each operation can use several input and output interfaces. Analyzing the documentation, one should let input interfaces contain **only input signals**, and output interfaces contain **only output signals**.

Let's review an example of interface information extraction. Let's take a short specification for microprocessor data hub unit Databox (DB), describing connection between DB and external unit Memory Access Unit (MAU).

Arbiter (13), working with round priority, selects one request from 5 issues of short requests (they may come in parallel) to send into MAU. The pace of transmission is 1 request per 2 cycles in case of 32-byte or 1 request per 4 cycles in case of 64-byte request. The selected short request is stored into DB and is not sent to MAU. MAU has the following interface (see the following table).

Table 2. Interface to MAU

Signal	***Type***	***Semantics***
val_data_reqack_to_mau_03	***O***	*Request validity flag. Request for lower part of 64-byte cache row.*
val_data_reqack_to_mau_47	***O***	*Request validity. Request for higher part of 64-byte cache row*
cop_data_reqack_to_mau[2:0]	***O***	*Operation code:* *«011» - reset register LDB* *«100» - send coherent answer* *«101» - reset register STB* *«110» - send data from register STB* *«111» - send data and reset register STB*
source_reg_data_reqack_to_mau[4:0]	***O***	*Number of register LDB/STB*

According to this documentation fragment, abstract reference model requires input interfaces for all five resources of short requests. More information about interfaces is likely to be found later. Keep on reading: “request is sent to MAU”. MAU is an external to the being tested unit and test system has to control data sent there. To control them, first, they have to be taken; second, restrictions for data should be known. To have these tasks done, component “output interface” being correspondent to implementation output interface is created in reference model. This component is overloaded in reference model adapter (where it is bound to implementation signals) and start solving task of getting signals going from implementation and sending them to controlling component. Let this output interface be named as *iface14*. It will contain signals *val_data_reqack_to_mau_03*, *val_data_reqack_to_mau_47*, *cop_data_reqack_to_mau*, *source_reg_data_reqack_to_mau*. Notice, that subdivision of signals into interfaces is only logical in current version of C++TESK. Only selected signals are allowed in correspondent interface adapters but it is not checked. After finding of interfaces, the rest of documentation is ordered by means of block diagrams or table of cycle accurate operation description if cycle accurate model is needed. If cycle accurate model is not needed or it has been developed, one may proceed to the following chapter.

Development of reference model

We proceed to development of test oracle being a control component. Having received stimuli from stimulus generator, it sends them to target system, receives its reactions and checks them (see figure 1). This checking requires reference values obtained from *reference model*. Test oracle can be said to be a «wrapper» of reference model. We will return to test oracle later, speaking about reference model now. Being used in C++TESK structure of reference model is represented in figure 3.

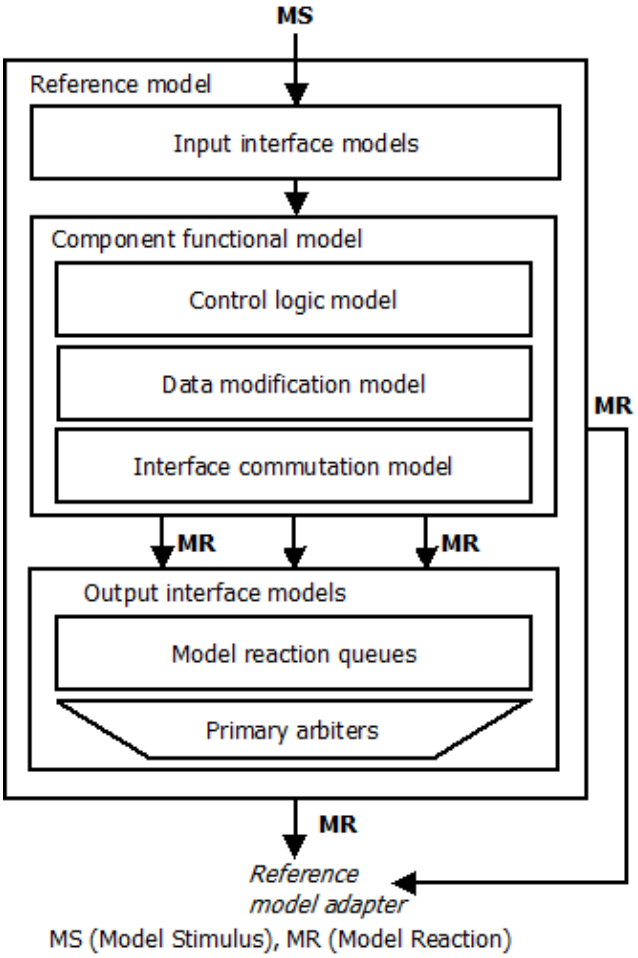

Figure 3. Structure of C++TESK reference model

Reference model contains the following main parts.

- *Input interface models;*
- *Functional model of target system;*
- *Output interface models.*

All data inside of model are carried by messages being instances of class *Message*. It is made for the purpose of unification of interfaces between test system components. Input and output interface models are instances of class *Interface*.

Functional model of system is a part of reference model class but can be written in a separated class. Functional model contains of *control logic*, *data processing*, and *interface commutation models*. First two parts can be implemented in separated classes, but the third part depends on interfaces in reference model class.

The following sub chapter describes message model development. Typically, at least two types of message are used – for input and output interfaces. After introduction of message model, we will return to reference model development.

2.1 Message model

All necessary macros and classes are located in file <**hw/message.hpp**>.

Message model is a class developed by means of macro *MESSAGE(class_name) {}*. Let us place empty constructor and destructor inside of macro's code block. Then let us turn on copying constructor by adding macro *SUPPORT_CLONE(class_name)* after those two functions.

Each message contains one or several *data fields*. These fields do not necessary have the same names as names of DUV wires, but their names are to be convenient for test system developer. In the majority of cases, field names are equivalent to the wire names at certain abstract level, but do not go too far: one "global" field in message for data from all wires is not convenient at all. To include fields into message class, macro *DECLARE_FIELD(name, length)* is used. The length should not exceed 64 bits. This macro defines a variable in message class and creates get and set functions *message_class_object.get_field_name* and *message_class_object.set_field_name* for the variable. Manual initialization of fields is allowed in message constructor as they are available via their names. Although, values do not have to be assigned to the fields manually; this work can be done automatically by macro *RANDOMIZE_MESSAGE(pointer_to_message_class_object)*. Notice, that all explicitly assigned values will be overwritten after macro being called. All fields should be registered in message class constructor by macro *ADD_FIELD(message_class::field_name)*.

Header for message models (*fifo_msg.h*) can look as follows.

```
#pragma once
#include <hw/message.hpp>
namespace cpptesk {
namespace fifo {

MESSAGE(InputData) {
public:
  InputData();
  virtual ~InputData();

  SUPPORT_CLONE(InputData);
  DECLARE_FIELD(data, 8);
};

MESSAGE(OutputData) {
public:
  OutputData();
  OutputData(uint8_t datum);
  virtual ~OutputData();

  SUPPORT_CLONE(OutputData);
  DECLARE_FIELD(data, 8);
};
}}
```

Notice, that directive #pragma once is wide distributed but not fully standard way to ask compiler include the header file into library only once (the other way is to use #ifndef, #define, and #endif).

File with message model implementation (*fifo_msg.cpp*) can look as follows.

```
#include <fifo_msg.h>
namespace cpptesk {
namespace fifo {

InputData::InputData(void) {
  ADD_FIELD(InputData::data);
  RANDOMIZE_MESSAGE(*this);
}
```

```
InputData::~InputData(void) {}

OutputData::OutputData(void) {
  ADD_FIELD(OutputData::data);
}

OutputData::OutputData(uint8_t output_data) {
  ADD_FIELD(OutputData::data);
  data = output_data;
}

OutputData::~OutputData(void) {}
}}
```

2.2 Reference model

Necessary macros and classes are located in file **<hw/model.hpp>**.

Model class is created by macro *MODEL(model_class_name) {}*. Minimally necessary methods include constructor and virtual destructor. Selected in previous parts interfaces are defined in model class by macros DECLARE_INPUT(interface_name) for input interfaces of target system and DECLARE_OUTPUT(interface_name) for output interfaces of target system. Each output interface may be supplied with a method returning availability of the interface by means of standard for interface method *interface_name.isReady()*. Let such methods be declared as *virtual bool isInterfaceNameReady() const*. Notice that availability of interfaces depends on running operations using them: if some operation has been started on the interface, it will be busy till operation stops its execution.

Model also contains definition of operations being sequences of registered interface calls. Being, in fact, operations, model methods are declared by macro *DECLARE_STIMULUS(method_name)* and *DECLARE_REACTION(method_name)*. They are not distinguished, but the first macro is typically used for definition of operations themselves, the second macro is used for operation parts requiring reading of data from output interfaces.

Header of reference model class (*fifo_model.h*) can look as follows.

```
#include <hw/model.hpp>
#include <fifo_msg.h>
namespace cpptesk {
namespace fifo {

MODEL(FIFO) {
public:
  FIFO();
  virtual ~FIFO();
  DECLARE_INPUT(iface1);
  DECLARE_OUTPUT(iface2);
  virtual bool isIface1Ready() const;
  DECLARE_STIMULUS(push_msg);          // operation "push data"
  DECLARE_STIMULUS(pop_msg);           // operation "pop data"
  DECLARE_REACTION(get_pop_msg);       // reaction "read output data"
  void push_item(int data);            // reference model function "push data"
  uint8_t pop_item(void);              // reference model function "pop data"
protected:
  std::vector<uint8_t> fifo;
};
}}
```

Now let us speak a few words about implementation of reference model class. Constructor of the class must contain calls of interface constructor with the only parameter being text description of the correspondent interface. Sensible names are recommended as they will appear in diagnostics messages. Input and output interfaces are registered by means of *ADD_INPUT(interface_name)* and *ADD_OUTPUT(interface_name)* correspondingly.

Let us proceed to description of functions showing interface availability. They use standard for interface class method *isReady* and simply return its value: *return interface_name.isReady()*.

Let us now describe operations. Each operation starts from copying of input message into local variable by means of macro *CAST_MESSAGE(input_message_type)*. Then message may be sent to the interface, set during operation start. If it is sent to an input interface, we are speaking about stimulus start, or about reaction start in another case. If we want to send message to an input interface with standard parameters (i.e., without changing of message, of interface name), this sending can be done by macro *START_STIMULUS(start_mode)*. Start modes are the following.

- *PARALLEL* creates different process for sending and processing of sending results; the process is executed in parallel with operation commands written after START_STIMULUS;

- *SEQUENTIAL* stops execution of operation till stimulus execution is finished.

Being sent message will be *serialized* or turned into sequence of actions to DUV applied via its input wires. This work is done by *serializers* in *reference model adapter* (*see next step*).

Notice: if sending message or used interface are different from those being input parameters of function (what are “those being input parameters” will be clarified later), one should use macro *RECV_STIMULUS(start_mode, interface_name, message_object_name)*. In this case operation has to be started from *START_PROCESS()*, not from *START_STIMULUS(start_mode)*, and to be finished by *STOP_PROCESS()* instead of *STOP_STIMULUS()* (*see later*).

Operation description may contain macro *CYCLE()*, telling to the test system that execution of this operation requires one cycle of delay. Operation is finished by macro *STOP_STIMULUS()*.

Parts of operations reading data from output interfaces are subject to separation into specific methods for definition as “reactions” (*DEFINE_REACTION*). In a reaction method reference model says to the test system that it has certain message to be obtained on a given output interface of DUV by means of macro *SEND_REACTION(start_mode, interface_name, message_object_name)*.

Notice: SEND_REACTION works with only output interfaces!

Standard interface and message object may be obtained by means of macros *GET_IFACE()* and *GET_MESSAGE()* (under standard we means those, which are input parameters). Reaction methods (if they are developed) should be called from stimulus method. Best way to call reaction method is to call *reaction_method_name(process, interface_name, message_object_name)*, where process is an implicitly defined operation execution context. To send the context is necessary for joining all operation parts together.

Development of DUV functional model (more precisely, control logic part) is not a specific task in test system development by means of C++TESK. This task is done by means of C++ standard library and that is why detailed comments about it will be omitted.

After all the steps, the following file with model implementation may be obtained (*fifo_model.cpp*).

```
#include <fifo_model.h>
namespace cpptesk {
namespace fifo {

FIFO::FIFO() {
  ADD_INPUT(iface1);
  ADD_OUTPUT(iface2);
}

FIFO::~FIFO() {}

bool FIFO::isIface1Ready() const { return iface1.isReady(); }

DEFINE_STIMULUS(FIFO::push_msg) {
  InputData data = CAST_MESSAGE(InputData);
  push_item(data.get_data());
  START_STIMULUS(PARALLEL);
  CYCLE();
  STOP_STIMULUS();
}

DEFINE_STIMULUS(FIFO::pop_msg) {
  InputData data = CAST_MESSAGE(InputData);
  START_STIMULUS(PARALLEL);
  OutputData outdata = OutputData(pop_item());
  get_pop_msg(process, iface2, outdata);
  STOP_STIMULUS();
}
DEFINE_REACTION(FIFO::get_pop_msg) {
  START_PROCESS();
  SEND_REACTION(SEQUENTIAL, GET_IFACE(), GET_MESSAGE());
  STOP_PROCESS();
}

void FIFO::push_item(uint8_t data) {
  assert(fifo.size() < 16);
  fifo.push_back(data);
}

uint8_t FIFO::pop_item(void) {
  assert(fifo.size() > 0);
  uint8_t data = fifo[0];
  fifo.erase(fifo.begin());
  return data;
}
}}
```

Auxiliary tool VeriTool

Digital devices are developed in *hardware description languages* (HDLs) like Verilog and VHDL. In this paper we use only Verilog language (http://en.wikipedia.org/wiki/Verilog) as one of the most distributed. To connect design under verification (DUV) with test systems developed in different languages (not Verilog), Verilog standard describes special *Verilog procedural interface* (VPI).

To make files of *VPI-environment* (environment is understood to be a set of functions connecting Verilog-simulator running DUV and test system developed in C/C++ in standard to Verilog way), we will use licensed under GPL tool *VeriTool* (http://forge.ispras.ru/projects/veritool/files). VeriTool uses Verilog syntax analyzer built in open source Verilog simulator Icarus Verilog

(http://sourceforge.net/projects/iverilog/). Installation of both tools can be done both manually and automatically by means of script from C++TESK package.

3.1 Command line options of tool VeriTool

Syntax of VeriTool command line looks as follows.

$ veritool [options] input_files

Supported command line options are represented below.

1) --module=*module* sets name of being tested Verilog-module to generate test system components. If this option is absent, the first found module will be processed.

2) --clk=*signal* sets name of clock signal (default value is clk). This option is used only if options --all or --testbench are used.

3) --rst=*signal* sets name of reset signal (default value is rst). This option is used only if options --all or --testbench are used.

4) --rstpos sets high active level of reset signal (active level is supposed to be low by default).

5) --all turns on generation of all test system components including DUV wrapper in Verilog, VPI-adapter, VPI-functions and C-structures with DUV-interface. Usage of this option is equivalent to simultaneous usage of --testbench, --vpi-media, --vpi-systf, and --interface (with default values).

6) --testbench[=*file*] enables generation of DUV wrapper in Verilog and can set the resulting file name (default value is testbench.v).

7) --vpi-media[=*file*] enables generation of VPI-adapter and can set the resulting file name (default value is vpi_media.c).

8) --vpi-media-header=*file* sets the name of header file for VPI-adapter (default value is the name set by --vpi-media, but with extension .h). This option works only if option --vpi-media is set.

9) --vpi-systf[=*file*] enables generation of VPI-functions and can set the resulting file name (default value is vpi_systf.c).

10) --vpi-systf-header=*file* sets the name of header file for VPI-functions (default value is the name set by --vpi-systf, but with extension .h). This option works only if option --vpi-systf is set.

11) --interface[=*file*] enables generation of C-structures with DUV-interface definition and can set the resulting file name (default value is interface.c).

12) --interface-header=*file* sets the name of header file for C-structures with DUV-interface (default value is the name set by --interface, but with extension .h). This option works only if option --interface is set.

13) --destination=*dir* sets an output directory (default value is current directory).

14) --version | -v shows tool version.

15) --help | -h shows information about available options.

3.2 Running VeriTool

To run VeriTool is possible as follows.

$(PATH_TO_VERITOOL)/bin/veritool --clk=clock --rst=reset --rstpos target/src/myfifo.v --vpi-systf=vpi_systf.cpp --vpi-media=vpi_media.cpp --interface=interface.cpp

Notice: names of clk and rst signals should be the exactly same as DUV has. The tool supports only one signals of each semantic in one DUV. Possible names of signals are clock, CLOCK, clk, CLK, reset, RESET, rst, RST etc.

Development of reference model adapter

We keep on developing of test oracle. Now we are to make *reference model adapter (mediator* and *adapter* are synonyms). Adapter is an heir of reference model, providing following means.

- binding with DUV for applying stimuli and receiving reactions;
- listening of DUV output interfaces for «unexpected» reactions;
- matching of DUV and reference model reactions with detailed diagnostics information.

Reference model adapter structure is represented in figure 4.

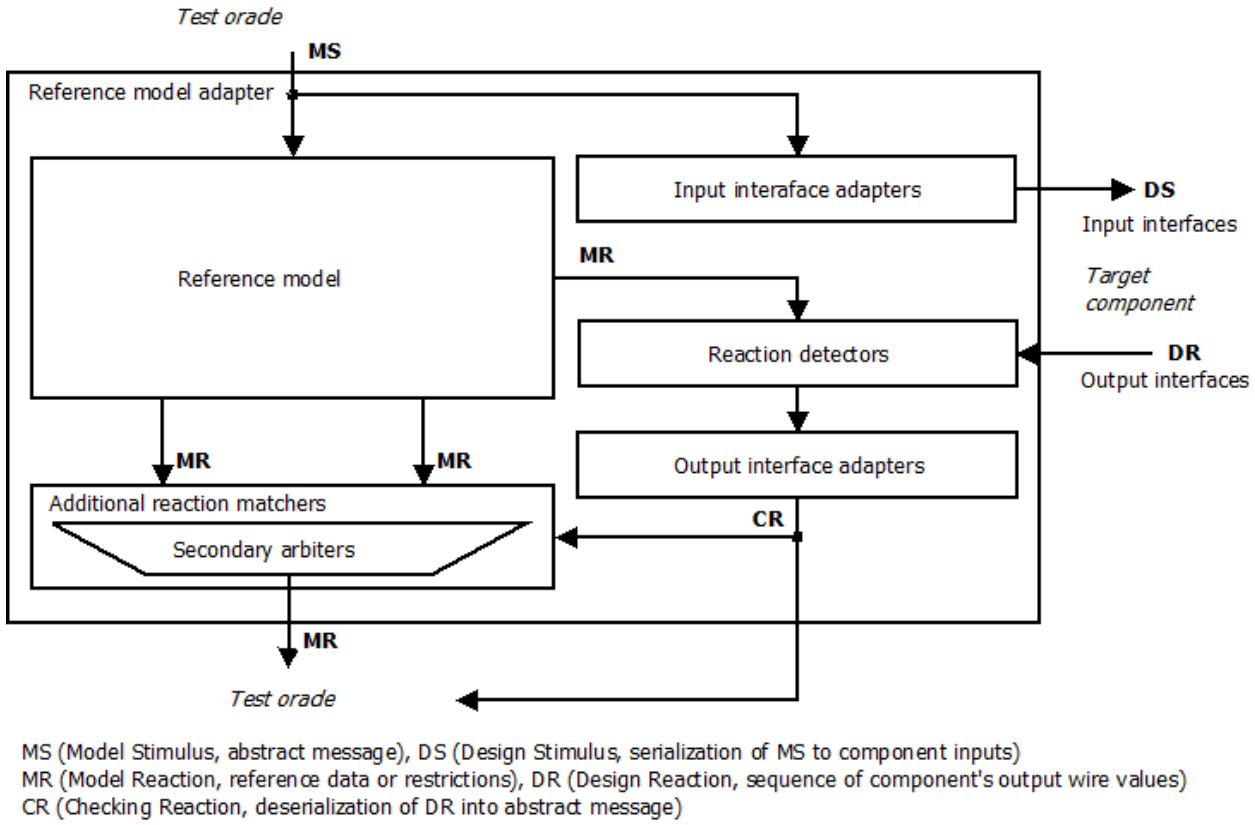

Figure 4. Generalized structure of reference model adapter

General components of adapter are the following.

- inherited reference model;
- *input interface adapters* (or *serializers)* which transform model stimuli into DUV stimuli by converting abstract messages into sequence of DUV input wire assignments;
- *output interface adapters* (or *deserializers)* which transform DUV reactions into being checked reactions by converting DUV output wire values into abstract messages;
- *reaction detectors* which are necessary for tracking of DUV output activity; they are created for each output interface and check DUV reactions by means of model reactions;
- *additional reactions matchers* (*secondary arbiters*) which make final decision of correspondence between DUV and model reactions (if relation 1:1 is not satisfied)

4.1 Development of adapter class

All necessary classes and macros for development of adapter are located in **<hw/media.hpp>**.

Typically, mediator development requires additional structures describing VPI-interface of DUV. Such structures have been already created by means of VeriTool (file interface.h).

Adapter is a heir of reference model class and defined by macro *ADAPTER(adapter_class_name, model_class_name)*. *MEDIA* is an alias of *ADAPTER*. This class at least contains default constructor, virtual destructor and also maximum *timeout of waiting for sent reaction* (let us call it REACTION_TIMEOUT being a simple int constant). Also, adapter class contains overloaded functions showing availability of interfaces.

Macro *DECLARE_PROCESS(serializer_name)* allows definition of a method processing reference model messages sent to a given model interface into sequence of DUV input wire assignments *(the method is called serializer).* Serializers are created for each input interface. Macro *DECLARE_PROCESS(deserializer_name)* allows definition of a method processing DUV messages into message understood by test system. Deserializers send wrapper DUV messages to check against reference model messages. Deserializers are created for each output interface.

In some cases, reference model may send several messages to the same output interface in short time. It may happen that adapter will not be able to match those model messages and received DUV message. In this case, component ordering model reactions (so called *arbiter*) defined by macro *DECLARE_ARBITER(ordering_mode, ordering_component_name)* is used. There are following ordering modes.

- *FIFO* where model messages are ordered according to the time of their sending,

- *Priority* which orders model messages according to their *priority*,

- *Adaptive* which match reference model and DUV reactions by their data.

Output interfaces also support finding unexpected reactions. Reaction is called unexpected if interface adapter couldn't match this reaction with correspondent reference model reaction. It might

happen if model reaction has not been sent or matching method did not recognize correspondence.

Finding of unexpected messages is done by means of standard listeners defined by macro *DEFINE_BASIC_OUTPUT_LISTENER(listener_name, output_interface_object_name, condition)*, where condition is a predicate for message detection.

Also, adapter should contain the following auxiliary functions and variables for testing HDL DUV.

- *virtual void initialize()* prepares test system and DUV for verification;
- *virtual void finalize()* stops DUV and might contain finalizing commands for test system;
- *inputs_t inputs* is a structure with the same fields as DUV input signals (generated);
- *virtual void setInputs()* sets DUV inputs signals to values of fields from inputs;
- *outputs_t outputs* is a structure with the same fields as DUV output signals (generated);
- *virtual void getOutputs()* sets fields from outputs to DUV output signal values at this cycle;
- *virtual void simulate()* gives control flow to simulator for modeling one cycle of DUV.

Macro *VERITOOL_ADAPTER* used instead of ADAPTER creates all these functions and variables automatically.

Result of these actions may look as follows (*fifo_media.h*).

```
#pragma once
#include <hw/veritool/media.hpp>
#include <model.h>
#include <interface.h>
#include <string>

namespace cpptesk {
namespace fifo {

VERITOOL_ADAPTER(FIFOMediator, FIFO)
{
public:
  FIFOMediator();
  virtual ~FIFOMediator();

  const static int REACTION_TIMEOUT = 100;
  virtual bool isIface1Ready() const;
  DECLARE_PROCESS(serialize_iface1);
  DECLARE_PROCESS(deserialize_iface2);
  DEFINE_BASIC_OUTPUT_LISTENER(listen_iface2, iface2, outputs.DO_STROBE);
  DECLARE_ARBITER(Priority, matcher_iface2);
};
}}
```

Let us proceed to implementation of reference model adapter. We are to bind input and output model interfaces to their *adapters (serializers* and *deserializers* correspondingly) by means of macros *SET_INPUT_ADAPTER(interface_object_name, full_adapter_method_name)* for input interfaces and *SET_OUTPUT_ADAPTER(interface_object_name, full_adapter_method_name)* for output interfaces.

Function *setReactionTimeout(timeout)* sets maximum time in cycles which model reaction may wait for correspondent DUV reaction. Here we recommend to use *REACTION_TIMEOUT* defined earlier. If the DUV reaction does not appear in timeout, “missing reaction” error is shown.

Structures inputs and outputs must be initialized in adapter before their usage by means of functions

clear_inputs(&inputs) and *clear_outputs(&outputs)* generated by VeriTool.

Macro *SET_ARBITER(output_interface_object_name, ordering_component_name)* should be used for binding output interface and having been defined message ordering component. Macro *CALL_OUTPUT_LISTENER (full_listener_name, output_interface_name)* starts listener of unexpected reaction on the current interface.

To turn on debug information printing to the console, standard reference model class function *debug(debug_level)* should be called. We will use debug level *DEBUG_ERROR*.

Functions showing whether output interface is free are usually overloaded in reference model adapter but they use functions of super class (of reference model). Implementation of these functions in adapter class may be extended by usage of output signals of DUV if necessary.

Definition of serializers starts from macro *DEFINE_PROCESS(full_serializer_name)*. In the beginning of serializer implementation, reference message can be copied into local variable. To access reference model is possible by means of macro *CAST_MESSAGE(message_class)*. After it, the serializer itself begins by macro *START()*. Function *iface.capture()* "captures" this particular input interface. This capturing is only logical and needed for prevent two or more operations from simultaneous capturing of the same input interface. If necessary, interface state can be checked by means of function isReady* before stimulus sending. Then appropriate structure fields used for setting of DUV input signals is assigned with values carried by reference message's fields. When all the actions have been already applied, the interface should be released by calling *iface.release()*. This calling should be done in advice of one cycle before necessary release of the interface. To make serializer wait for one cycle before following setting DUV input signals is possible by means of macro CYCLE(). Command iface.release() is usually followed by macro STOP() meaning finish of stimulus processing. At least one CYCLE() is required to be between iface.capture() and STOP().

Definition of deserializers starts from macro *DEFINE_PROCESS(full_deserializer_name)*. In the beginning of deserializer implementation, a reference to given by reference model message can be created. This message will contain implementation reaction. Macro *START()* shows beginning of deserialization process. This macro is followed by condition statement allowing to match received implementation reaction and reference model one. This condition is written inside of macro *WAIT_REACTION(match_condition)*. Valid signals are used very often here, but some reference message fields such as address may be used here. Notice that if several implementation reactions satisfy the match_condition, test system will stop with assertion. Therefore match_condition should be enough detailed to differentiate which one really matches to the given reference message. When correspondent reaction is found, some field values of output structure are written into local message. If necessary, macro CYCLE() can make deserializer wait for one cycle before reading the following data. After all data having been read, macro *NEXT_REACTION()* is used, asking test system to proceed to the next registered model reaction at this particular interface (if this reaction exists). At last, macro STOP() shows finish of deserialization process.

Reference model adapter file may look as follows (*fifo_media.cpp*).

```
#include <fifo_media.h>
#include <sync.h>
#include <interface.h>
#include <vpi_media.h>
#include <iostream>

namespace cpptesk {
namespace fifo {
```

```
FIFOMediator::FIFOMediator() {
  SET_DEBUG_LEVEL(DEBUG_INFO, true);
  SET_INPUT_ADAPTER(iface1, FIFOMediator::serialize_iface1);
  SET_OUTPUT_ADAPTER(iface2, FIFOMediator::deserialize_iface2);

  SET_REACTION_TIMEOUT(FIFOMediator::REACTION_TIMEOUT);

  SET_ARBITER(iface2, matcher_iface2);
  CALL_OUTPUT_LISTENER(FIFOMediator::listen_iface2, iface2);
}

FIFOMediator::~FIFOMediator() {}

bool FIFOMediator::isIface1Ready() const {
  return FIFO::isIface1Ready();
}

DEFINE_PROCESS(FIFOMediator::serialize_iface1) {
  InputData msg = CAST_MESSAGE(InputData);
  START_PROCESS();
  CAPTURE_IFACE();
  inputs.WR_STROBE = 1;
  inputs.DI = msg.get_data();
  RELEASE_IFACE();
  CYCLE();
  STOP_PROCESS();
}

DEFINE_PROCESS(FIFOMediator::deserialize_iface2)
{
  OutputData &msg = CAST_MESSAGE(OutputData);
  START_PROCESS();
  WAIT_REACTION(outputs.DO_STROBE);
  msg.set_data(outputs.DO);
  NEXT_REACTION();
  STOP_PROCESS();
}

}}
```

Development of test scenario

Test scenarios are test system integral part. Their main aim is to *describe stimuli* and select *method* of their usage. Test scenarios are used to configure stimulus generation class objects from library of C++TESK. Stimulus generator configured by test scenario is shown in general test system scheme in figure 1 as “stimulus generator”.

C++TESK supports two common ways of stimulus generation. The first one is based on *finite state machine traversing* (this way will be considered in one of the following chapters) and *random stimulus selection* from the list of registered stimuli. Both ways have both advantages and disadvantages. The first one provide better test coverage, being aimed to covering of all states in FSM. The second way requires less labor costs and applies more variable stimuli from the beginning. The first way as a rule is more difficult in development and test running. Following the second way, it is difficult to catch errors appearing after specific sequence of operations applied.

Remember that the at previous steps of test system creation reference model, reference model adapter, test coverage collector have been developed. According to figure 1, we are to develop not only stimulus generator but test oracle as well. The latter is an adapter extended by *checking components* (see figure 5), under which we understand *stimulus precondition* and *reaction*

comparators. Stimulus preconditions are predicates which are satisfied if current reference model state allows to start the particular stimulus. If the precondition is satisfied, stimulus will be started at this cycle, in the other case it will be ignored. Then, according to the test scenario, either new stimulus will be checked, or the time is shifted. Preconditions may be placed inside of test scenario or of reference model used by the scenario. Reaction comparators are automatically created when reference model adapter is created.

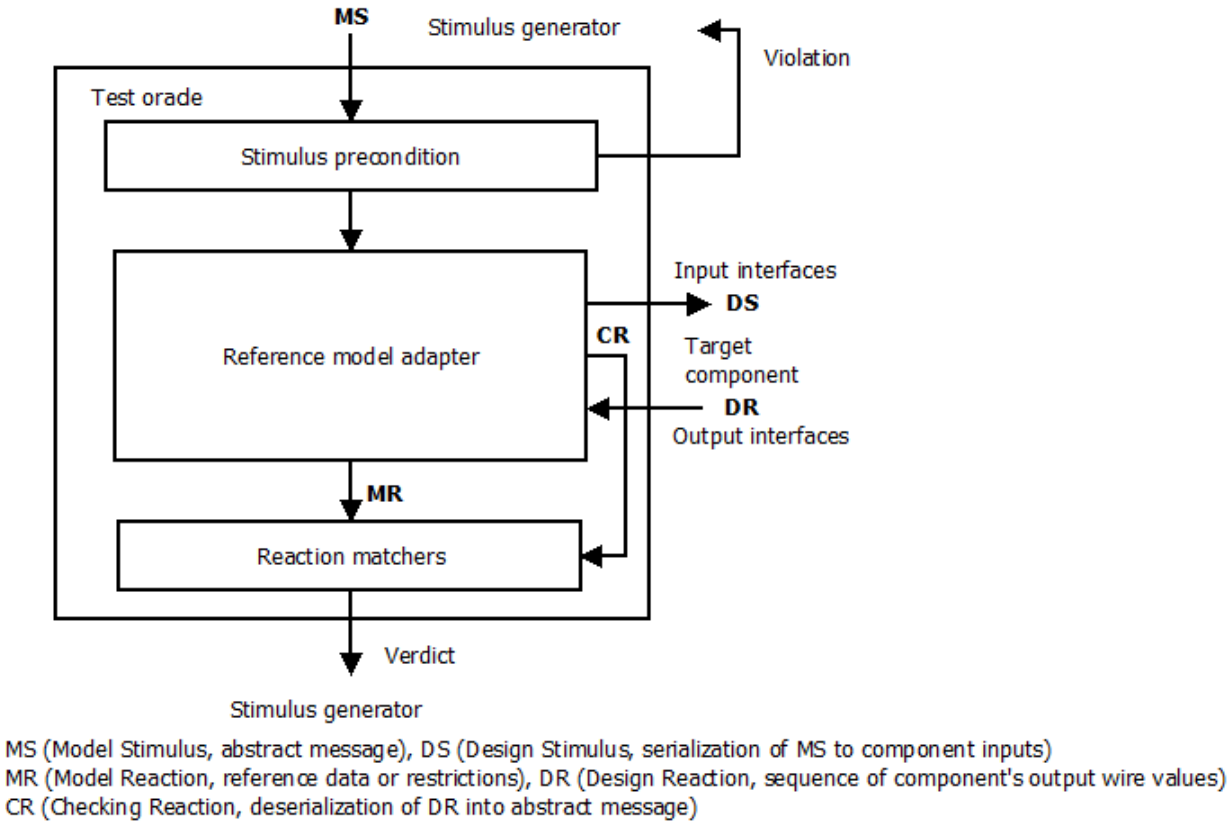

Figure. 5. Test oracle

So, we have all the means to proceed to test scenario development. Scenarios should contain:

- object of reference model adapter class;
- *scenario functions*;
- (optionally) *current state function*.

Scenario functions create reference model stimuli and apply them to reference model if precondition is satisfied (the latter may be checked inside of reference model). Current state function is necessary if FSM-based stimulus generator is used.

As having said above, C++TESK supports two types of test sequence generation: *FSM-based* and *random* one. The first way will be reviewed later. To generate stimuli randomly, only scenario functions are necessary, current state is not calculated. In this case, registration of scenario functions should be done according to the following convention. The first stimulus in the list should be the one *shifting time* (*cycle of stimulation*), the other ones should not do it. To set up random stimulus generation is possible by means of the following parameters.

- *maximum cycle number*
- *strategy of stimulus application* (*max load*, *min load*, *dynamic load* and *their parameters*)

Usage of random scenario parameters will be considered in detail later. Let us proceed to scenario development.

All necessary classes and macros are defined in **<ts/scenario.hpp>**. Used namespaces are cpptesk::ts, cpptesk::hw, cpptesk::tracer. Scenario class is defined by means of macro *SCENARIO(scenario_class_name) {}*. The following functions are standard for scenario definition.

- constructor and virtual destructor;
- *virtual bool init(int argc, char ** argv)* initializes test scenario;
- *virtual void finish()* finishes test scenario;
- *std::string get_state()* (not used now) returns current state;
- *bool scenario_function_name(IntAbcCtx& ctx)* make reference model stimuli and start them;
- object of reference model adapter class.

Having obtained class is a base class for test scenario classes. It can be inherited to obtain new child scenarios. These scenarios use selectively registered scenario functions defined in base class, use appropriate current state function also defined in base class. Child scenarios usually do not require any methods besides constructor and destructor.

Declaration of test scenario class might look as follows.

```
#pragma once

#include <fifo_media.h>

#include <ts/scenario.hpp>

using namespace cpptesk::ts;
using namespace cpptesk::hw;
using namespace cpptesk::tracer;

namespace cpptesk {
namespace examples {

SCENARIO(ParentScenario) {
public:
  virtual bool init(int argc, char ** argv);
  virtual void finish();
  bool scen_nop(IntAbcCtx& ctx);
  bool scen_push(IntAbcCtx& ctx);
  bool scen_pop(IntAbcCtx& ctx);
  std::string get_current_state();

  ParentScenario(void);
  virtual ~ParentScenario(void);

protected:
  DUTMediator dut;
};

class ChildScenario : public ParentScenario {
public:
  ParentScenario();
  virtual ~ParentScenario();
};
}}
```

After declaration of test scenario class, its methods are to be implemented. Parent class constructor

(if it is not used as a scenario itself) should call only constructor of its super class *ScenarioBase()*. Each child scenario calls its parent constructor and also uses the following function.

setup(string_with_scenario_name, &full_init_function_name, &full_finalizing_function_name, &full_current_state_function_name).

Usage of setup function is limited by the requirement of locating all registered functions inside of one class. After calling setup function, scenario methods can be registered by means of macro *ADD_SCENARIO_METHOD(full_scenario_function_name).*

Scenario functions are methods of parent scenario class. In the beginning of the function, typical local variable should be declared if they are necessary. Then, macro *IBEGIN* starts substantial part of scenario function. IBEGIN is followed by macro *IACTION{}*, having inside all the actions of creating reference message and its sending to reference model. The simplest sending looks as follows. First, stimulus precondition is checked and if it is satisfied, application is made by function *reference_model_object.start(&full_reference_model_stimulus_name, interface_name, message, reference_model_class_name::START_REQUEST_IFACE).* In case of scenario function shifting time, function *cycle* should be used instead of stimulus call: *reference_model_object.cycle()*.

Whatever precondition shows, IACTION block should be finished by macro *YIELD(reference_model_object.verdict())*, getting results of test oracle work at current cycle.

Returning to test scenario methods, in the simplest case, method *init* is usually empty and returns true, method *finish* asks HDL-simulator to stop by calling *reference_model_object.finalize()*. *Current state function get_state* takes some values returning by reference model functions, makes a string basing on them and return it. Typically, developing of such a function is not a trivial task.

Implementation of test scenario methods may look as follows.

```
#include <fifo_scen.h>

#include <tracer/tracer.hpp>

namespace cpptesk {
namespace examples {

bool ParentScenario::init(int argc, char ** argv) {
  return true;
}

void ParentScenario::finish() {
  dut.finalize();
}

bool ParentScenario::scen_nop(IntAbcCtx& ctx) {
  IBEGIN
  IACTION {
    dut.cycle();
    YIELD(dut.verdict());
  }
  IEND
}

bool ParentScenario::scen_push(IntAbcCtx& ctx) {
  IBEGIN
  IACTION {
    if(dut.isIface1Ready() && !dut.is_full()) {
```

```
      InputData msg = InputData();
      dut.start(&DUT::push_msg, dut.iface1, msg, DUT::START_REQUEST_IFACE);
    }
    YIELD(dut.verdict());
  }
  IEND
}

bool ParentScenario::scen_pop(cpptesk::ts::IntAbcCtx& ctx) {
  IBEGIN
  IACTION {
    if(dut.isIface2Ready() && !dut.is_empty()) {
      InputData msg = InputData();
      dut.start(&FIFO::pop_msg, dut.iface2, msg, FIFO::START_REQUEST_IFACE);
    }
    YIELD(dut.verdict());
  }
  IEND
}

std::string ParentScenario::get_state_fifo() {
  std::string state;
  std::stringstream out;
  out << "";
  state = out.str();
  return state;
}

ParentScenario::ParentScenario(void) : ScenarioBase() {}

ParentScenario::~ParentScenario(void) {}

ChildScenario::ChildScenario(void) : ParentScenario() {
  setup("Example scenario", &ParentScenario::init, &ParentScenario::finish,
      &ParentScenario::get_state_fifo);
  ADD_SCENARIO_METHOD(ParentScenario::scen_nop);
  ADD_SCENARIO_METHOD(ParentScenario::scen_push);
  ADD_SCENARIO_METHOD(ParentScenario::scen_pop);
}

ChildScenario::~ChildScenario(void) { };
}}
```

Having been developed test scenario should be bound to stimulus generator. It is done by means of *test repository*. Repository contains of test scenario class objects together with information of selected test engine and its parameters. Each row of test repository should contain different objects of test scenario classes while object names are used for identification of running test scenario. To start test with given name, it should be sent to HDL-simulator as a parameter +scen=*scenario_name* (see file testbench.v generated by VeriTool).

Necessary macros for test repository are defined in file <*utils/testreg.h*>. Repository starts from macro *TEST_REGISTRY_BEGIN*, and finishes by macro *TEST_REGISTRY_END*. Each test scenario is associated with test engine by macro *REGISTER_TEST(test_scenario_object, test_engine_type, default_parameters)*. Parameter test_engine_type must be either *engine::fsm* or *engine::rnd*. The first one means FSM-based generation, the second one means random selection of stimuli. *Default_parameters* allow to set up stimulus generator. The parameter list might include but not restricted by the following ones.

- *-uerr=N, N>0* orders to test until error counter being equal to N;

- *-nt* turns off tracing in format *UTT*;
- *-nt2* turns off tracing in format *UTT2*;
- *--length N, N>0* orders to execute random test scenario exactly N cycles;
- *--parallel* turns on parallel stimuli in random test scenario;
- *--maxload* turns on max stimulus load in random test scenario.

Written in test repository parameters have less priority then those being sent via ARGV (see file testbench.v generated by VeriTool).

Developed test repository (*fifo_tests.cpp*) might look as follows.

```
#include <utils/testreg.h>
#include <fifo_scen.h>

using namespace cpptesk::examples::fifo;

ChildScenario scen_rnd;

TEST_REGISTRY_BEGIN
REGISTER_TEST(scen_rnd, engine::rnd, "-uerr=1 --length 10000 --parallel")
TEST_REGISTRY_END
```

Development of functional coverage

Additional possibility helping to improve test process is *function coverage*. Under coverage usually some value measured in absolute or relative numbers is meant. Test coverage is gathered for evaluation of test completeness: the more numbers are, the better DUV is checked.

There are several approaches to gather the coverage. In the introduction to C++TESK we will speak only about *functional coverage* showing number of checked *functional properties* selected by an engineer.

At the moment C++TESK supports only manually selected functional coverage structures. Tracing of their covering is made automatically. Resulted coverage is printed as a report.

Verification engineer should analyze documentation to the DUV to find situations appearing in functional model which must happen during test process. It is useful to ask DUV developers about their vision of test cases. Then engineer creates functional coverage structure for selected situations. This work is rather creative and its results depend on the qualification of engineer.

It should be noticed, that C++TESK does not still allow to describe dynamic situations where some action happens after another, in *n* cycles, etc. Such possibilities will be implemented later.

All necessary macro and classes are declared in **<ts/coverage.hpp>** and **<tracer/tracer.hpp>**.

Coverage class might be a part of reference model class or it may be a different class. In the second case it has to contain a reference to reference model object, constructor and virtual destructor. In both cases object of class CoverageTracker should be declared. This object will keep registered test coverage structures, print reached test coverage in trace of UTT2 format.

Coverage can be set by macro *DEFINE_ENUMERATED_COVERAGE(coverage_id, string_form, ((coverage_element_id, coverage_element_string_form), (…,…), …))*. It declares an object of class Coverage with *coverage_id* name. *Coverage_element_id* is used for identification of all

subcomponents of being defined coverage structure. All string representations are used in report generation. The tool also allows to copy coverage structures, product coverage structures with or without excluded tuples. Macros *DEFINE_ALIAS_COVERAGE(coverage_id, string_form, initial_coverage_id), DEFINE_COMPOSED_COVERAGE(coverage_id, string_form, initial_coverage_A_id,initial_coverage_B_id),DEFINE_COMPOSED_COVERAGE_EXCLUDING (coverage_id, string_form, initial_coverage_A_id, initial_coverage_B_id, {excluded_element_from_A_id, excluded_element_ from_B_id}, ...)* are used for these purposes.

To let coverage information be collected, current state of reference model should be read and used for increasing of an appropriate coverage structure item. The increased values are to be sent into object of CoverageTracker class by means of operator <<. It is easy to do by means of two functions. The first one returns object of Coverage class, selected according to the used reference model parameter. The second function receives value returned by the first function and sends it to the object of CoverageTracker type by means of operator <<.

Header file for coverage class (*fifo_coverage.h*) may look as follows.

```
#pragma once
#include <hw/model.hpp>
#include <ts/coverage.hpp>
#include <tracer/tracer.hpp>

#include <fifo_model.h>

using namespace cpptesk::ts;
using namespace cpptesk::hw;
using namespace cpptesk::tracer;
using namespace cpptesk::tracer::coverage;

namespace cpptesk {
namespace fifo {

class DUT;
class DUTCoverage {
public:
  DUTCoverage(DUT &dut): dut(dut) {}
  virtual ~DUTCoverage() {};
  CoverageTracker coverageTracker;
  DEFINE_ENUMERATED_COVERAGE(DUT_FULLNESS, "DUT fullness",
                                                ((I0, "0 (free)"), (I1, "1"),
(I2, "2"), (I3, "3"), (I4, "4 (full)")));
  DUT_FULLNESS select_coverage_element_DUT_FULLNESS(void) const;
  void update_coverageTracker_DUT_FULLNESS(void);
protected:
  FIFO &fifo;
};

}}
```

Implementation of this class (*fifo_coverage.cpp*) may look as follows.

```
#include <fifo_coverage.h>

namespace cpptesk {
namespace fifo {

DUTCoverage::DUT_FULLNESS
DUTCoverage::select_coverage_element_DUT_FULLNESS(void) const {
  switch(dut.dut_fullness()) {
```

```
    case 0:  return DUTCoverage::DUT_FULLNESS::I0;
    case 1:  return DUTCoverage::DUT_FULLNESS::I1;
    case 2:  return DUTCoverage::DUT_FULLNESS::I2;
    case 3:  return DUTCoverage::DUT_FULLNESS::I3;
    case 4:  return DUTCoverage::DUT_FULLNESS::I4;
    default: assert(false);
  }
}

void DUTCoverage::update_coverageTracker_DUT_FULLNESS(void) {
  coverageTracker << DUTCoverage::select_coverage_element_DUT_FULLNESS();
}

}}
```

If coverage means are created in a separated class, object of the class should be added to reference model class to let binding the reference to the reference model inside of coverage class. To update coverage information kept in coverage class, the second function of the two above is called from time to time. Such “times” might be cycle shifts, certain operation starts, i.e. all events changing reference model state where functional coverage might change and should be evaluated.

Reached coverage is traced into UTT2 format. To analyze trace, special report generator is used which can be started by command *$CPPTESK_HOME/bin/reportgen.sh trace_name.utt2* from the directory with trace (trace name should not contain spaces). The report generator creates directory Reports with reports about testing to be read by usual Internet browser.

Development of FSM-based scenarios

Addition to chapter 5.

To generate test actions, FSM-based test engine can also be used. It implies dynamic creation of **tight connected** finite state machine and its exploration by certain **non-extra state algorithm**. The FSM is defined by current state function and list of stimuli (scenario functions in this context) with their preconditions. Test is supposed to be finished, when all the stimuli are applied in each state, and no error has been found.

Each scenario function is supplied by *iteration context* (see signature of scenario functions: IntAbcCtx& ctx is the context), keeping iterator values. Iterators are constructions for iterative assignment of input parameters of started operations. They let consider stimuli started from the same scenario function but with different iteration variable values to be different stimuli. It is necessary for applying of all input parameter combinations of all stimuli.

Iteration variables are declared in the beginning of scenario functions: int& variable_name = IVAR(a). Macro IVAR() should have inside one letter of Latin alphabet (from a till z). Values of these variables will be kept into mentioned above iteration context.

Let us remind that after initialization of all local variables the pair of macros IBEGIN-*IEND* is written. Inside of the pair IBEGIN-IEND nested into each other cycles by iteration variables may be written, and the last one should have macro IACTION{} inside. Cycle by iteration variable is a usual cycle “for”: for(variable_name = 0; variable_name < 2; variable_name++). Usage of iteration variable allows to define a set of stimuli being different in this parameter, *variable_name*. Notice, that iterators are not always needed. In the simplest examples like FIFO, iterators are not necessary but can be added artificially to increase number of FSM states. Let us use iteration variable into

scenario function nop to increase the number of arcs issuing from states.

```
bool ParentScenario::scen_nop(IntAbcCtx& ctx) {
  int &pseudo_iteration = IVAR(a);
  IBEGIN
  for(i = 0; i < 10; i++)
    IACTION {
      dut.cycle();
      YIELD(dut.verdict());
    }
  IEND
}
```

Current state function may be the following one.

```
std::string ParentScenario::get_state_fifo() {
  std::string state;
  std::stringstream out;
  out << dut.fullness() << ","<< dut.isIface1Ready() << dut.isIface2Ready();
  state = out.str();
  return state;
}
```

To start FSM-based stimulus generation, test engine engine::fsm should be used:

```
#include <utils/testreg.h>
#include <fifo_scen.h>

using namespace cpptesk::examples::fifo;

ChildScenario scen_fsm;

TEST_REGISTRY_BEGIN
REGISTER_TEST(scen_fsm, engine::fsm, "-uerr=1 -nt -nt2")
TEST_REGISTRY_END
```