Developing test systems for multi-modules hardware designs

Mikhail Chupilko
Institute for System Programming of RAS
Moscow, Russia
chupilko@ispras.ru

Abstract—**The paper proposes the approach of creating test systems for complex hardware designs. The designs can be subdivided into modules and verified separately. The proposed architecture of separated verification systems and the way to combine them into a complex test system are based on simulation-based verification of hardware designs. The components of test systems are connected in a TLM-like way that allows to use high-level model of commutation based on messages and thereby to simplify merging of several test systems into a test system for the complex component.**

Keywords-complex hardware designs; simulation-based verification;combination of test systems

I. INTRODUCTION

The importance of *hardware verification* has been urgent for many years. There are several techniques to conduct it but there is still not unified solution. *Hardware designs* are developed by means of languages of hardware behavior description (hardware description languages, HDLs, e.g., Verilog [1]). Even relatively simple *modules* (i.e., parts of complex designs or very simple designs) can hardly be checked by means of a code inspection, to say nothing of complicated designs. Therefore, the verification, i.e. checking of designs' behavior and their specification to mutual consistency, is of importance and attention. There are some estimates talking that about 70% total amount of development efforts are spent on the verification [2]. The real practice shows that usually at least half of total design development time is spent on this aim. The code written in HDLs called *HDL-model* can be translated into a *net-list* and then on this base the real device will be created. If no modifications of a net-list take place, the functionality of the produced device will be the same as of the HDL-model. Even if the corrections took place, the equivalence can be checked by means of special tools, e.g. [3]. Therefore, to reveal and correct functional errors is possible at the stage of HDL-model development. It should be noticed that the correction of functional errors on later stages and, in particular, after chip manufacturing requires more efforts and time, because necessity to pass all stages of manufacturing again.

Complicated designs are usually developed by means of *abstraction* and *decomposition* techniques. The common approach is to develop the whole system abstractly and then to create the subparts (modules) more carefully. Usually the test system for the whole design under verification (DUV) is created but it is rather abstract one. As some parts of the system can be critical for the total system behavior, the module-oriented test systems are developed. If these "little" test systems can help to improve the common test system for the whole DUV, it will be good for code reuse and debugging abilities of the common test system.

This work is organized as follows. First, the review of works related to verification of complicated hardware designs is given. In the following section, the architecture of test systems formerly proposed in [4] is described. Next section introduces the architecture of multi-module test system. Section 5 includes case studies. Section 6 concludes the paper.

II. RELATED WORKS

There are two common ways of hardware verification. First, we could use *formal methods*, e.g. to prove satisfiability of a logical constructions in a formal model based on hardware design in model checking [5]. All corrections are made over static hardware designs. They are applicable well in case of module-level verification but their scalability is insufficient [6] to use them in case of the whole designs. To improve the scalability, the hardware designs can be checked dynamically during the *simulation* process. Simulation-based verification allows checking the designs in real cases of their work. Usually, simulation-based approaches possess the high level of scalability and the thoroughness of verification varies according to available resources and time. Below, when we are talking of verification we are meaning only simulation-based verification.

Typical components of the test systems are *test sequence generator* (*stimuli generator*), *reaction checker* (or *oracle*), *test completeness estimator*. The generation of sequence or stimuli can be made manually and explicitly by means of test cases description. Other stimuli generators produce test actions self-automatically requiring manual description of variables set in each stimulus with restrictions for their values. To generate stimuli, special mechanism selects subset of available stimuli, solves constraints in their variables assigning fit numbers to the fields and start stimuli. This approach is called *constrained-driven verification* (*CDV*) [7]. Another well-distinguished way of stimuli generation is *FSM traversing* [8], where states of the FSM are states of system under test and transitions between them are applied operations. The reaction checker always knows the correct behavior of DUV, e.g. utilizing a *reference model*. Test completeness estimator usually works on the base

of *source code coverage* or *functional model code* or *aspects coverage.*

The main subject of the article is the possibility for developing test systems allowing reuse in the multi-module complex design case. To reduce extra problems we suggest using a uniform architecture for all test systems. When merging test systems the question about merging of each component from their structure arises. Consequently, the parts should be intended for easy merging initially, their architecture should provide such possibilities.

Among well-spread approaches of verification, we selected *Open Verification Methodology* (*OVM* [9]) as the most spread and seemed to be most suitable for our purposes to observe its abilities of merging. Test systems according to OVM are developed in accordance to the given architecture and subdivision of test system's components into several layers [10]. The test system for each single module is named *Open Verification Component* (*OVC*) (see Figure 1). Each OVC contains basic means of creating CDV stimuli flow (*transaction sequencers*, where *transaction* is a *abstract message* containing information about test situation) and delivering the flow to the DUV (so called *transactors*, i.e. components which make direct and reverse transformation of transactions and DUV's wire signals). OVC can be connected to each other when they are put under control of the *united test controller* or in other words *virtual generator* [7]. In this case, all the OVCs will generate stimuli flows. Developer of test system modifies the redundant generators connected with unavailable DUV's wires to switch them off. The OVCs with turned off generators check their target modules correctness regarding to the stimuli flow to the components influence their target modules work. Components checking correctness (or *scoreboards*) continue checking using only available data from DUV. Summarizing, test systems made according to OVM satisfy many tasks usually arising in verification including connection of several test systems. It should be noticed that OVM is oriented to programming in SystemVerilog so that connection with other languages is possible but knotted with development of intermediate components.

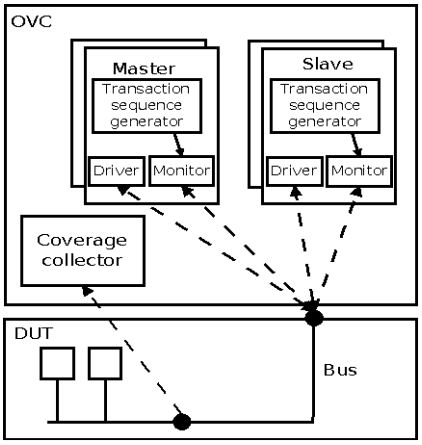


Figure 1. Open verification component

We developed new approach and presented some aspects of it in [4]. That paper touched upon only the problems of oracles' development for single test systems. We will shortly review the approach in this section and thoroughly in the next. The method utilizes simulation-based verification and implies the subdivision of test system components generally into *stimuli generator* and *oracle* (or *reaction checker*). The generator should create a flow of stimuli and apply them into the oracle usually based on a reference model developed on selected level of abstraction. Test is complete when the generator indicates it according to a strategy of generation.

This approach has a distinctive feature when compared with OVM: the reference models used in reaction checkers can be originally written at a high level of abstraction and then can be specified according to a progress of DUV development. To make it possible, there are several techniques in the approach such as model reactions' arbitration mechanism, DUV reactions' detection mechanism, etc. After all, while DUV is developed, the verification engineers usually develop the *software simulator* of the total DUV. As they usually do it by means of C++, to use exactly this language is useful to reuse parts of simulators to create reference models with no extra efforts. As the approach described in [4] uses C++, it has a certain advantage over OVM while task of system simulator reuse is conducting.

III. Architecture of Single Test System

The architecture described in [4] was based on UniTESK technology [11] developed in Institute for System Programming of RAS. The architecture includes *stimuli generators (including FSM-based one)*, *oracle (reaction checker)*, and after all *coverage tracker* and *verification report generator* (see Figure 2). The stimuli generator produces sequence of *messages* and sends them as a parameter while calling interface operations of the reference model. These calls are named sending *model stimuli*:

```
dut.start(&DUT::pop_stimulus, dut.iface1, msg);
```

Reaction checker processes messages and makes *model reactions*. Then it sends stimuli (which now are called *design stimuli*) to the *target design* and receives its *design reactions*. At last, reaction checker checks correspondence between model and design reactions and returns a *verdict* about correctness of DUV at the current cycle. The coverage tracker saves the information about *functional coverage* at the current cycle. The report generator dumps important information about the verification process such as called operations, reached functional coverage and verification result.

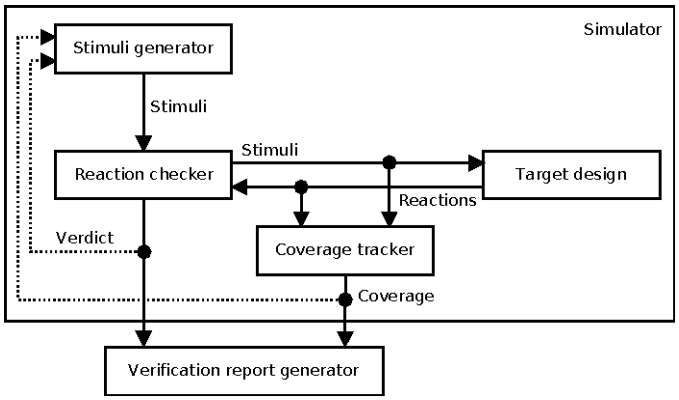
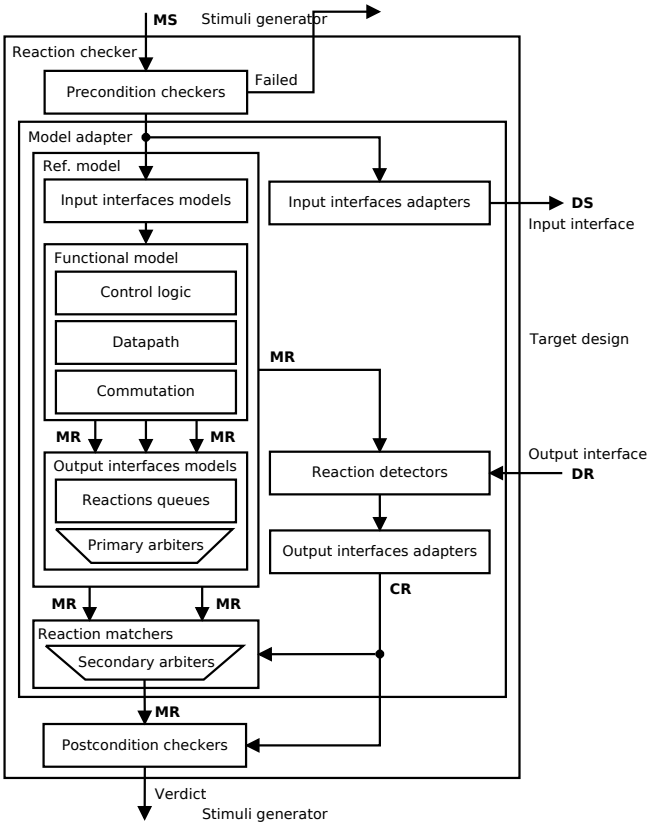


Figure 2. Single test system architecture

The most complex component requiring and allowing reuse is reaction checker (see Figure 3). The reaction checker supplies the reference model with all the necessary functions, which make it possible for the stimuli generator (or other reaction checker) to utilize the reference model and model adapter.



MS - Model stimulus (abstract message)
DS - Design stimulus (cycle- and pin-accurate serialization of MS)
MR - Model reaction (reference message or constraint)
DR - Design reaction (cycle- and pin-accurate series)
CR - Checked reaction (deserialization of DR)

Figure 3. Reaction checker architecture

The messages sent into the checker are called *model stimuli* (***MS***). The generator (or other reaction checker) addresses the **MS** flow to one of the *input interfaces models*.

On having received the **MS**, *pre-condition checkers* check if the **MS** can be started. The **MS**, which staring requirements are not satisfied at the current state of the functional model, is rejected. In the other case they proceed both to the DUV via *input interface adapters* (in this step processed **MS** is called *design stimuli*, ***DS***) and to the *functional model* via *input interface models* set by the generator.

The functional model produces *model reactions* (***MR***) and places them into one of the *output interface models* according to rules included into the functional model.

The output interface models contain *reaction queues* keeping **MR** and *primary arbiters* selecting **MR** subset at the current simulation cycle. The arbiters work according to a strategy selected by the test developer. The **MR** subset is sent into reaction detectors to help recognizing *DUV's reactions* (***DR***).

The *reaction matchers* fetch the **MR** sub-subset from the output interfaces models and then start expecting the corresponding reactions from the **DUV**. The restrictions are made by the second arbiters and customized by test developer. There are certain time restrictions for the waiting. If they are violated, the test system shows the *timeout error* and stops working.

When the **DR** are found, they are put into one of the *output interface adapters* (corresponding **MR**, if it is found, helps to select which one). If some of DR does not have the corresponding MR, the test system shows an *unexpected reaction* error and stops working.

The output interfaces adapters send the *checked reactions* (***CR***) into the reaction matcher to find the corresponding **MR** satisfied the restrictions made by secondary arbiters. After all, *post-condition checkers* check equivalence between corresponding **MR** and **CR**. If the **MR** and **CR** are equal, the test process goes on. If there is a problem with messages, the test system shows a given error and stops working.

Test successfully finishes when the stimuli generators makes everything it was asked to do by the test developer (like visiting all reachable states of the FSM, etc.).

IV. Archtecture of Merged Test System

The proposed TLM-based approach to develop single test system can be used while the task is to make test system for a total DUV on the base of test systems for the components of the DUV. To reuse test systems for components is convenient when parts of test systems can be connected to each other be means of the interfaces they have. Therefore, the selected TLM-based way of interface development has a certain advantage: it allows reusing components of test systems as they are, not taking only parts from components or using copy-paste method. To develop complex test system is possible by means of the following steps.

When there are several test systems to connect some of them will miss their connection with DUV. We propose to create common test system and inject all small reaction checkers from earlier developed test systems connected to each other (see Figure 4). Therefore, input and output interfaces adapters and reaction detectors should be modified to connect with other reaction checkers. Fortunately, their separation from the reference model allows us to do without huge reference model modifications.

When merging the reaction checkers, we create the common test system and place sub reaction checkers into it. The common test system possesses its own stimuli generator, reaction checker and coverage tracker. While developing all these parts, to reuse some parts of test systems previously developed would be great achievement.

.

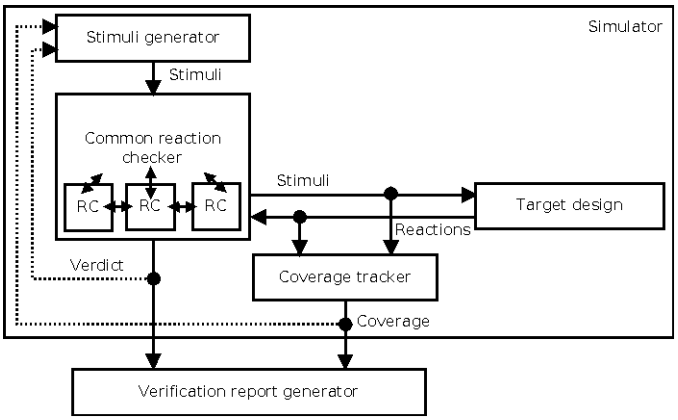


Figure 4. Architecture of multi-module test system

The stimuli generator can inherit scenario functions from sub generators but only if the part of message prepare and call of original reaction checkers were split from each other to different functions. In this case, the reaction checker used can be easily changed to a common by means of overloading the appropriate functions.

The coverage tracker is a common component for all test systems. To use it, a coverage structure should be described and registered in a tracker. The registration is identical in cases of single or multi-module test systems. To refresh the coverage information, some functions from functional models are usually used. Since the models have been inserted into the common test system, the reuse of the coverage becomes free of charge. The common test system just calls all of them at every cycle to make them collect data. It should be noticed, that coverage structures from single test systems in some cases do not provide important information for the case of combined DUV and in this case either cross-coverage is created or new coverage structures for the whole DUV are developed.

The combined reaction checkers is one of the most difficult parts of combined test systems. The common reaction checker looks like the single reaction checker but it should use included reaction checkers functionality. Only the common reaction checker is allowed to change values of DUV's wires while sub reaction checkers' input and output interfaces adapters are switched off. To switch them off is possible by means of overloading to make them send model message not in the absent DUV but to the other reaction checkers (see Figure 5).

To facilitate the connection between reaction checkers we propose to use *channels*. Channel is a way to connect an output interface model and an input interface model together. To do it the message from the input interface should be translated into a form applicable to the output interface to put into it. The channel can also broadcast message into several input interfaces. Usage of channels is given in Figure 6.

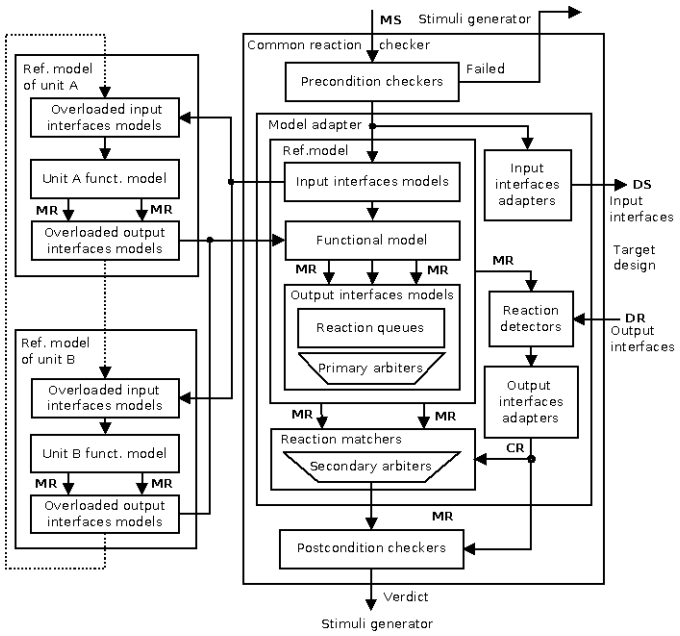


Figure 5. Architecture of multi-module reaction checkers

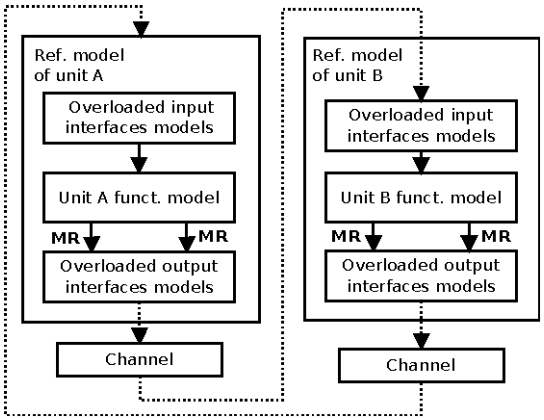


Figure 6. Architecture of multi-module reaction checkers

The overloaded input interfaces models usually contain precondition checkers so that they check protocols of the communications between sub modules. It can help to reveal problems, which can be found if the reference model takes input stimuli only from stimuli generator: the input variables values have wide variance and this variance usually corresponds to the real work situation for the DUV's parts.

The most distinctive feature of the approach, as it has been said before, is that it uses C++ and can use some parts of system simulators usually written in C++. Moreover, the reference models due to their architecture can be reused in development of Verilog-models. In this case, input and output interface adapters do not work as usual: input interface adapters should take the messages from the DUV, process them in functional model and send the messages to the DUV by means output interface adapters (see Figure 7).

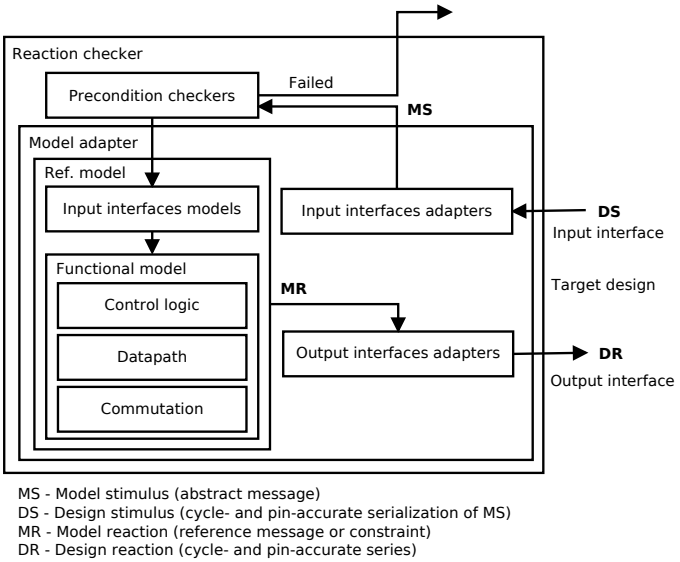


Figure 7. Architecture of reaction checker built into DUV

Common test system controls the reaction checkers built in DUV. The checkers help Verilog-model developer in accelerated development of the model as code in C++ with the same functionality as in Verilog is usually written quicker.

Summing up, the approach allows developing test systems, which can be reused as parts of common test system. These test systems check not only the output data of DUV but input data from, say, stimuli generator (by means of precondition checkers). When the test systems are connected to each other not to DUV, they can check the behavior of their neighbors, i.e. they check the interconnection protocol of DUV's components. The test system supports special means to make the interconnection such as interfaces and channels. After all, to reduce the time spent to make the first version of DUV for system-level verification, the test systems can be inserted into the Verilog code of DUV while the last is still under designing. The approach is supported by a library developed in C++, so that test systems being developed according to the approach, are also based on C++. It gives wide range of opportunities to use all means of C++ to facilitate the development of the test systems. It should be noticed, that C++ is usually used in system level simulators of DUV, so that parts of the simulators can be easily reused as reference models in the test systems and vice versa. At last, the library is compatible with UniTESK approach, which means the opportunity to develop high-quality tests based on FSM-traversing even for the system-level case and to spread test systems among clusters of computers.

V. CASE STUDY

The approach to develop single test systems has shown its effectiveness in several projects [4]. The most interesting cases are generalized in Table 1.

TABLE I. APPLICATIONS OF THE SUGGESTED APPROACH

Design under verification	Depth of verification	Source code, KLOC	Labor costs, man-months
Translation lookaside buffer (TLB)	Up to cycle-accurate	2.5	2.5
Non-blocking L2 cache	Up to detailed-timed	3	6
Northbridge data switch	Up to cycle-accurate	3	3
Memory access unit (MAU)	Up to cycle-accurate	1	1

The labor costs in Table 1 include verification plan writing, test system development, as well as verification process and debug of the developed test system. In case of non-blocking L2 cache the costs also include test system maintaining due to permanent modifications in the DUT. Table 1 shows that time spent to the verification can be roughly estimated to be one man-month per one KLOC. The L2 cache case is an exception from this rule, but it required additional supporting as it has been already said.

A comparison between our results and the second approach (OVM) could be worth knowing. Very little estimations of the OVM application efforts prevented us from doing it. We suggested those spent to the development of test systems with close functionality to be similar to the proposed methodology. It is because OVM also utilizes object-oriented language and the set of test system components looks like proposed. Nevertheless, there is a certain difference between approach given in this article and OVM: our approach provides additional means of FSM-based stimuli flow creation. Actually, this question has not been thoroughly analyzed by us. It is a point of the following research.

The proposed way of merging is a new revealed ability of the basic approach. Only some experiments were conducted to estimate the possibility of merging. First, the test system for a simple FIFO module was developed. It took about two men-days including efforts spent for documenting of the project. Then this FIFO module was multiplied to become three FIFO-modules into one envelop. Two of them were input buffers and the third one was output buffer. We placed between them an arbiter to select which one of the input buffers sends data to the output one. It always selected the first FIFO if it contained any data. To test this combination we combined three test systems for original FIFO-module and added functionality of the arbiter in a very simple way: functional model always reads data from the first FIFO if it contained any data and in other case read the second FIFO. Stimuli generator used original scenario functions like "pop" and "push" but with a little modification as now two FIFO played role of the input and the last one did of the output. The interface adapters of sub test systems were overloaded. Formerly, to make pop and push operations only two interfaces had been used: one input and out output. In case of input FIFOs, input interfaces were not changed (the functionality of setting values to the DUV's wires from sub reaction checkers had been removed before). Output interfaces were overloaded to send messages into output FIFO's input interfaces. Initially, the registration of adapters of interfaces had looked like:

```
CPPTESK_SET_OUTPUT_ADAPTER(iface3,
    FIFOMediator::deserialize_iface3);
```

where deserializer is a function which translates DUV's wires signals into message. As the output interface iface3 was not already output one, we overloaded this adapter:

```
   CPPTESK_SET_INNER_IFACE_ADAPTER(FIFOMediator,
fifo0, fifo0.iface3, MM::deserialize_inner_iface0);
```

New deserializer calls the output FIFO fifo2:

```
   CPPTESK_DEFINE_PROCESS(
        MM::deserialize_inner_iface0)
   {
     …
     if(!fifo2.is_full()) {
       …
       fifo2.start(&FIFO::push_msg,
                   fifo2.iface1, data);
     }
   }
```

The output interface model of output FIFO just gave the messages into common reaction checker's output interface model. To create the common test system we spent about half a day accounting the time to research of merging possibility. The time could have been spent for developing of complex test system from scratch is expected to be about 2 days, but it is not the most important. The time to connect sub test systems slightly correlates with complexity of DUV's sub part. Mostly, it depends on amount of input and output interfaces and efforts to connect them together. We have an estimate that to connect exactly one input interface to one output is possible in about one hour. This time will be spent to develop channel between them that will translate messages between channels. The average DUV's part, by our estimates, consists of about ten input and ten output interfaces, so that to connect two average DUV's parts is possible in one-two men-days, according to the productivity of the verification engineer.

VI. CONCLUSIONS

This approach gives us a useful way to develop test systems for separated parts of DUVs and then to merge the test systems with high level of reuse. The main advantages of the approach are the check of interconnection behavior with no additional code and special means to facilitate reuse. The architecture is supported by a library developed in C++ that allows using the great opportunities of the language while developing test systems. The fact that usually system level simulators are based on C++ points to the ability of reusing parts of system-level reference models and reference models of the developed test systems and vice versa. There are two bonuses: the test systems can control communications between functional models inside the DUV as its parts, especially when the DUV is still under construction, and that the library supporting the approach supports UniTESK technology which allows developing high-quality tests based on FSM-traversing and spreading test systems among clusters of computers.

REFERENCES

[1] IEEE 1364-2005, Verilog Standard.

[2] J. Bergeron, Writing testbenches: functional verification of HDL models. Kluwer Academic Publichers, 2003.

[3] Tool called Formality by Synopsys (http://www.synopsys.com/Tools/Verification/FormalEquivalence/Pages/Formality.aspx)

[4] M. Chupilko, A. Kamkin. A TLM-based approach to functional verification of hardware components at different abstraction levels. 12th Latin-American Test Workshop, 2011.

[5] E. Clarke, O. Grumberg, D. Peled. Model Checking. MIT Press, 1999.

[6] R. Kneuper. Limits of Formal Methods. Formal Aspects of Computing (1997) 3: 1-000, 1997.

[7] S. Iman. Step-by-step Functional Verification with SystemVerilog and OVM. Hansen Brown Publishing Company, 2008.

[8] V. Ivannikov, A. Kamkin, V. Kuliamin, A. Petrenko. Application of the UniTESK technology to functional verification of software. http://citforum.ru/SE/testing/unitesk_hard/, 2006 (in Russian)

[9] Open Verification Methodology, http://www.ovmworld.org.

[10] Y. Gubenko, A. Kamkin, M. Chupilko. Comparative analisys of modern technologies of hardware designs test development. http://citforum.ru/SE/testing/hardware_models/, 2009 (in Russian)

[11] A. Barancev et al. UniTesK approach to the software development: achievements and prospects. http://citforum.ru/SE/testing/unitesk/, 2004. (in Russian)