ISSN 0361-7688, Programming and Computer Software, 2012, Vol. 38, No. 1, pp. 34–42. © Pleiades Publishing, Ltd., 2012.
Original Russian Text © M.M. Chupilko, 2012, published in Programmirovanie, 2012, Vol. 38, No. 1.

Developing Test Systems for Multi-Modules Hardware Designs

M.M. Chupilko

Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia
e-mail: chupilko@ispras.ru

Received June 3, 2011

Abstract—An approach aimed at creating test systems for complex hardware designs is proposed. 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. Components of the test systems are connected in a TLM-like way (i.e., with the use of a high-level commutation model based on messages), which simplifies merging of several test systems into a test system for the entire complex component.

DOI: 10.1134/S036176881201001X

1. INTRODUCTION

The issues of *hardware verification* stay quite topical for many years. There are several techniques to conduct it, but no unified solution has been created yet. *Hardware designs* are developed with the help of *hardware description languages* (HDLs), such as, for example, Verilog [1]. Even relatively simple *modules* (i.e., parts of complex designs or very simple designs) hardly can be checked by means of manual code inspection, to say nothing of more complicated designs. Therefore, automated verification, i.e., checking the mutual conformance of designs' behavior and their specification is of importance and requires much attention. The existing estimates say that about 70% of total amount of development efforts are spent on the verification [2]. Practice shows that at least half of total design development time is spent on this. A code written in an HDL is called an HDL *model*; it can be translated into a *net-list*, and, then, on its base, a real device is created. If the net-list is not modified manually, the functionality of the produced device will be the same as that of the HDL-model. Even if some corrections took place, the equivalence can be checked by means of special tools, e.g., [3]. Therefore, functional errors can be revealed and corrected even at the stage of the HDL model development. It should be noted that the correction of functional errors on later stages and, in particular, after chip manufacturing requires more efforts and time because of 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 its subparts (modules) more carefully. Usually, test is created for the whole system (we will call it *design under verification*, DUV, or *implementation*), but they are rather abstract. As some parts of the system can be critical for the total system behavior, the module-oriented test systems are developed. If these "small" test systems could help to improve the common test system for the whole DUV, it would be very convenient. It would result in the increasing of the code reuse level and the debugging abilities of the common test system.

This paper is organized as follows. First, a survey of works related to verification of hardware designs is given. In Section 3, architecture of the test systems formerly proposed in [4] is described. Section 4 introduces architecture of multi-module test systems. Section 5 includes case studies. Section 6 concludes the paper.

2. SURVEY OF RELATED WORKS

There are two basic ways of hardware verification. First, *formal methods* can be used, e.g., to prove satisfiability of logical constructs in a formal model constructed on the basis of the hardware design in *model checking* [5]. All the analyses in this case are conducted over static hardware designs. These methods work well in the case of module-level verification, but their scalability is insufficient in the case of really complex DUVs [6]. To improve scalability, hardware designs can be checked dynamically in the course of a simulation process with the use of an HDL *simulator*. Simulation-based verification allows checking DUVs under conditions close to their real operation. Usually, simulation-based approaches possess high level of scalability, and thoroughness of verification varies depending on available resources and time. Below, speaking of verification, we mean only simulation-based verification.

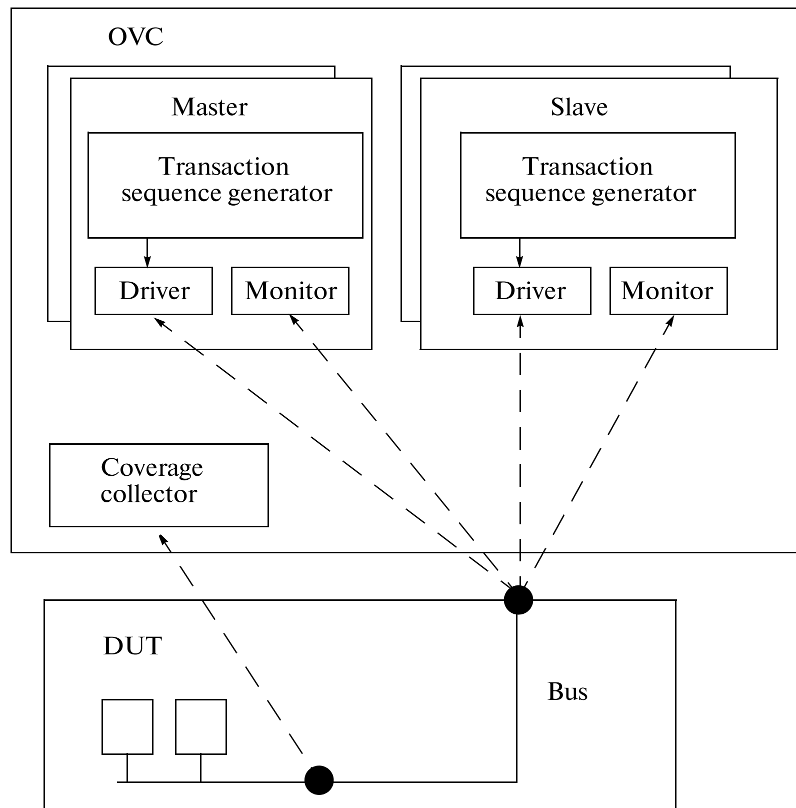

Fig. 1. Open verification component.

Typical components of test systems are *test sequence generator* (*stimuli generator*), *reaction checker* (or *test oracle*), and *test completeness estimator*. The test sequence generator can be made manually based on explicit description of test cases. Other stimuli generators produce test actions semi-automatically requiring manual description of set of variables in each stimulus with restrictions on their values. To generate stimuli in this case, a special mechanism selects a subset of available stimuli, solves constraints imposed on values of their variables, and starts stimuli. This approach is called *constraint-driven verification* (CDV) [7]. Another well-known way of stimuli generation is *FSM traversing* [8], where states of the FSM are states of the system under test and transitions between them are operations applied. The reaction checker should always know correct behavior of DUV, using, for example, a *reference model*. The test completeness estimator usually works based on *source code coverage* or *functional model code*.

The main subject of the paper is the possibility for developing test systems allowing reuse in the multi-module complex design case. In order to avoid extra problems, we suggest using a uniform architecture for all test systems. When merging test systems, there arises a question of merging their components. Such a possibility should be provided by relatively simple means initially built in the architecture.

Among the most widespread approaches to verification, we selected *Open Verification Methodology* (OVM) [9], which seems to be most suitable for merging test systems. According to OVM, test systems are developed in accordance with the given architecture and subdivision of test system's components into several layers [10]. The test system for each single module is a separate block called *Open Verification Component* (OVC) (see Fig. 1). Each OVC contains basic means for creating a CDV stimuli flow and delivering the flow to the DUV. The stimuli are created by means of a *transaction sequence generator*, where a *transaction* is an *abstract message* containing information about a test situation. The delivery problem is solved by the so-called *transactors*, i.e., components that implement direct and reverse transformation of data between transactions and DUV's wire signals. OVCs can be connected to each other under control of the *united test controller*, or, in other words, *virtual generator* [7]. In this case, all the OVCs will generate stimuli flows, and the developer of the test system may switch off redundant generators connected to unavailable DUV's wires. The OVCs with turned off generators check their target modules correctness according to the indirect stimuli flow from the other DUV's parts.

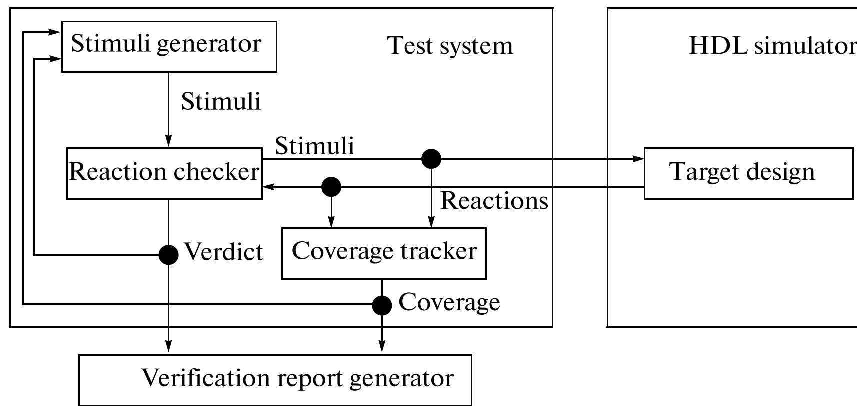

Fig. 2. Architecture of a single test system.

Summarizing the above-said, we may state that test systems made according to OVM satisfy many tasks, especially those arising in verification of connection of several test systems. It should be noted that the OVM methodology is oriented to programming in *SystemVerilog*, so that connection with other languages is possible but is associated with development of intermediate components.

We developed a new approach and presented some its aspects in [4]. That paper touched upon only problems of oracles' development for single test systems. We will briefly review this approach here and discuss it in detail in the next section. Our method utilizes simulation-based verification and implies subdivision of test system into two components: *stimuli generator* and *oracle* (or *reaction checker*). The generator should create a flow of stimuli and pass it to the oracle, which is usually based on a reference model developed on the selected level of abstraction. The test is completed when the generator indicates this in accordance with the generation strategy.

One of the distinctive features of our approach compared to OVM is that reference models used in reaction checkers can be originally written at a high level of abstraction and, then, can be refined in the course of the DUV development. This is achieved with the help of special techniques, such as *model reactions' arbitration mechanism, DUV reactions' detection mechanism*, etc. After all, in the course of the DUV development, the verification engineers usually develop a *software simulator* for the entire DUV. As they usually do this in C++, it makes sense to use this language in creation of reference models for test systems by reusing parts of simulators. As the approach described in [4] uses C++, it has a certain advantage over OVM in solving the task of system simulator reuse.

3. ARCHITECTURE OF SINGLE TEST SYSTEM

The architecture described in [4] was based on UniTESK technology [11] developed at the Institute for System Programming of RAS. The architecture includes *stimuli generators* (including *FSM-based* ones, in which *irredundant traversal algorithms* are implemented [12]), *oracle* (*reaction checker*), as well as *coverage tracker* and *verification report generator* (Fig. 2). The stimuli generator produces a *sequence of messages* and sends them as a parameter when calling interface operations of the reference model. These calls are called *sending model stimuli*:

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

The reaction checker processes messages and creates *model reactions*. Then, it sends stimuli (which are now called *design stimuli*) to the *target design* and receives its *design reactions*. Next, reaction checker checks correspondence between model and design reactions and returns a *verdict* about correctness of the DUV in the current cycle. The coverage tracker dumps the information about update of *functional coverage* in each cycle. The report generator processes test traces and shows important information about the verification, such as operations called, the functional coverage reached, and the verification result.

The most complex component requiring and allowing reuse is the reaction checker (Fig. 3). The reaction checker supplies its reference model with all necessary functions, which allow the stimuli generator (or other reaction checker) to use the reference model and the model adapter.

Messages sent to the checker are called *model stimuli* (MSs). The generator (or other reaction checker) directs the MS flow to one of the *input interface models*.

Having received the MS, the precondition checkers check if the MS can be started. If the staring requirements are not satisfied at the current state of the functional model, the MS is rejected. Otherwise, the stimulus is supplied to the DUV via *input interface adapters* (on this step, the processed MSs are called

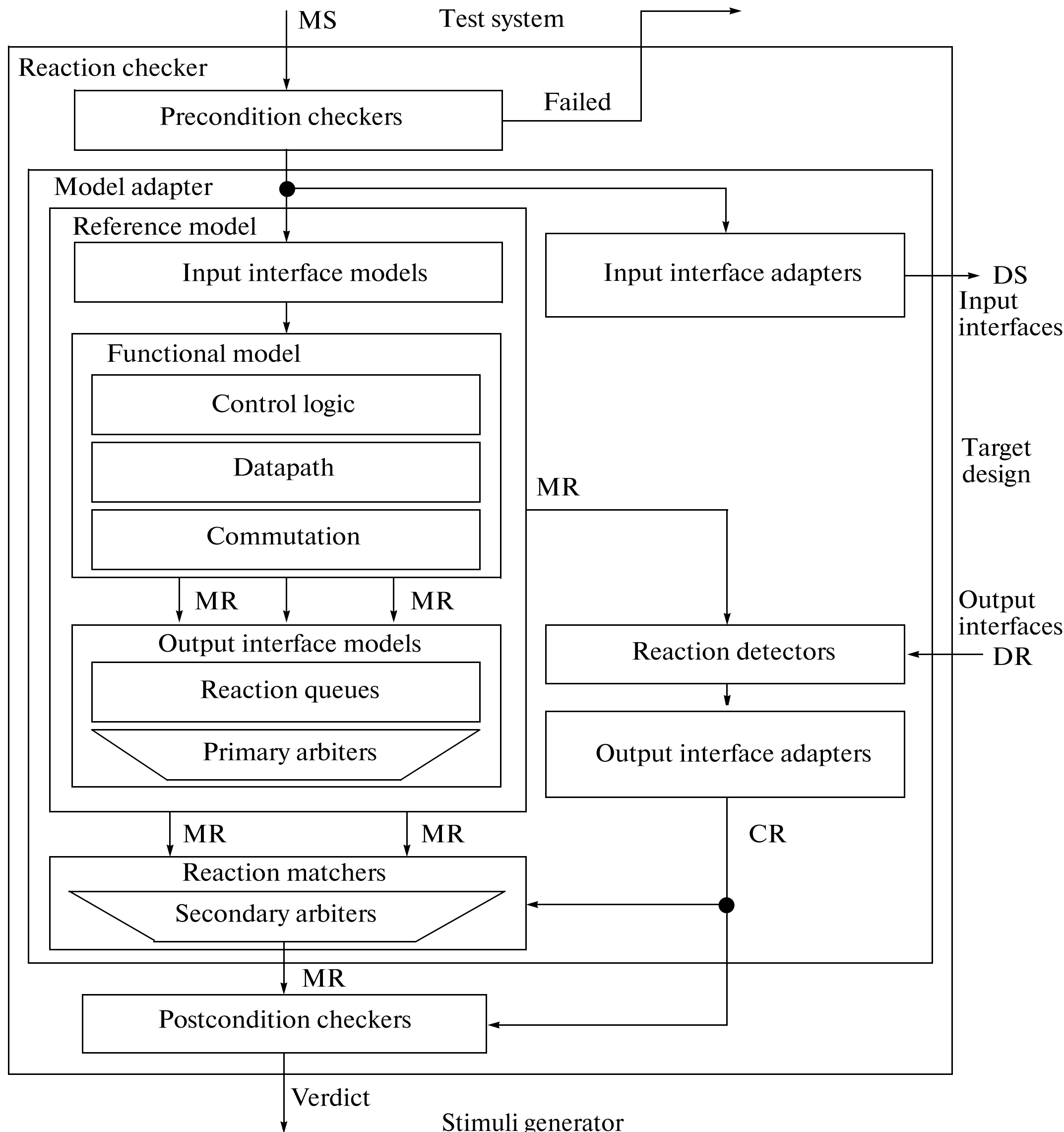

Fig. 3. Reaction checker architecture.

design *stimuli*, DS) and to the functional model via *input interface* models selected by the generator.

The functional model produces model reactions (MRs) 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 MRs and *primary arbiters,* which select an MR subset at the current simulation cycle. The arbiters work according to a strategy selected by the test developer. The MR subset is sent to the reaction detectors to help to recognize DUV's *reactions* (DRs).

The reaction matchers fetch the MR subset from the output interface models and, then, start to wait for the corresponding reactions from the DUV. Additional restrictions can be imposed on this subset by the secondary arbiters and customized by the test developer. There are certain time restrictions on the waiting. If they are violated, the test system shows the *timeout error* and stops working.

When the DRs are found, they are put into one of the *output interface adapters* (the corresponding MR, if found, helps to select the particular adapter). If some

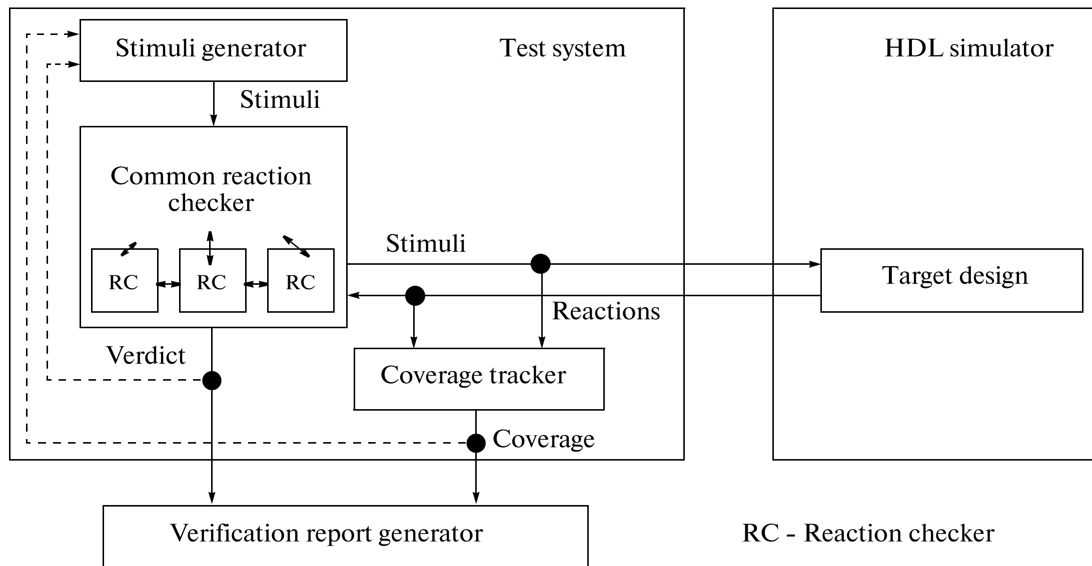

Fig. 4. Architecture of multi-module test system.

DRs do not have the corresponding MRs, the test system shows *unexpected reaction error* and stops working.

The output interface adapters send *checked reactions* (CRs) to the reaction matcher to find the corresponding MR that satisfies the restrictions imposed by the secondary arbiters. After all, the postcondition checkers check equivalence between the corresponding MR and CR. If the MR and CR are identical, the test process goes on. If an error is detected, the test system displays it and stops working.

The test successfully finishes when the stimuli generator have made everything it had been asked to do by the test developer (like visiting all reachable states of the FSM, etc.).

4. ARCHITECTURE OF MERGED TEST SYSTEM

The proposed TLM-based approach to the development of single test systems can be used when a test system for the entire DUV is created on the basis of test systems for its separate components. Test systems for components can conveniently be reused when parts of the test systems can be connected to each other through the interfaces they already have. Therefore, the selected TLM-based way of the interface development has certain advantages: TLM is designed just for solving the task of reusing components of the test system as they are, without taking only parts from the components or using a copy-paste method. The development of complex test systems proceeds as follows.

When several test systems are connect to one another, some of them miss their connection with the DUV. We propose to create a common test system that includes all small reaction checkers from the earlier developed test systems to be connected (see Fig. 4). Therefore, input and output interface adapters and reaction detectors should be modified to create connection with other reaction checkers. Fortunately, their separation from the reference model allows us to do this without huge reference model modifications.

When merging the reaction checkers, we create a common test system and place built-in reaction checkers into it. The common test system possesses its own stimuli generator, reaction checker, and coverage tracker. In the development of all these parts, the reuse of some parts of the earlier developed test systems is a good result by itself.

The stimuli generator can inherit scenario functions from the sub generators only if the parts of the model stimuli preparation and the calling of original reaction checkers are separated from each other and located in different functions. In this case, the reaction checkers used can easily be replaced with new ones by means of overload of the corresponding functions

The coverage tracker is a common component for all test systems. To use it, the coverage structure should be described and registered in the tracker. The registration is identical in the cases of single and multi-module test systems. To refresh the coverage information, some functions from the functional models are usually used. Since the old models have been inserted into the common test system, the reuse of the coverage costs nothing. The common test system just calls the required functions at every cycle to collect information about the coverage. It should be noted that coverage structures from single test systems sometimes do not provide important information for the case of combined DUVs; in this case, it is required either to

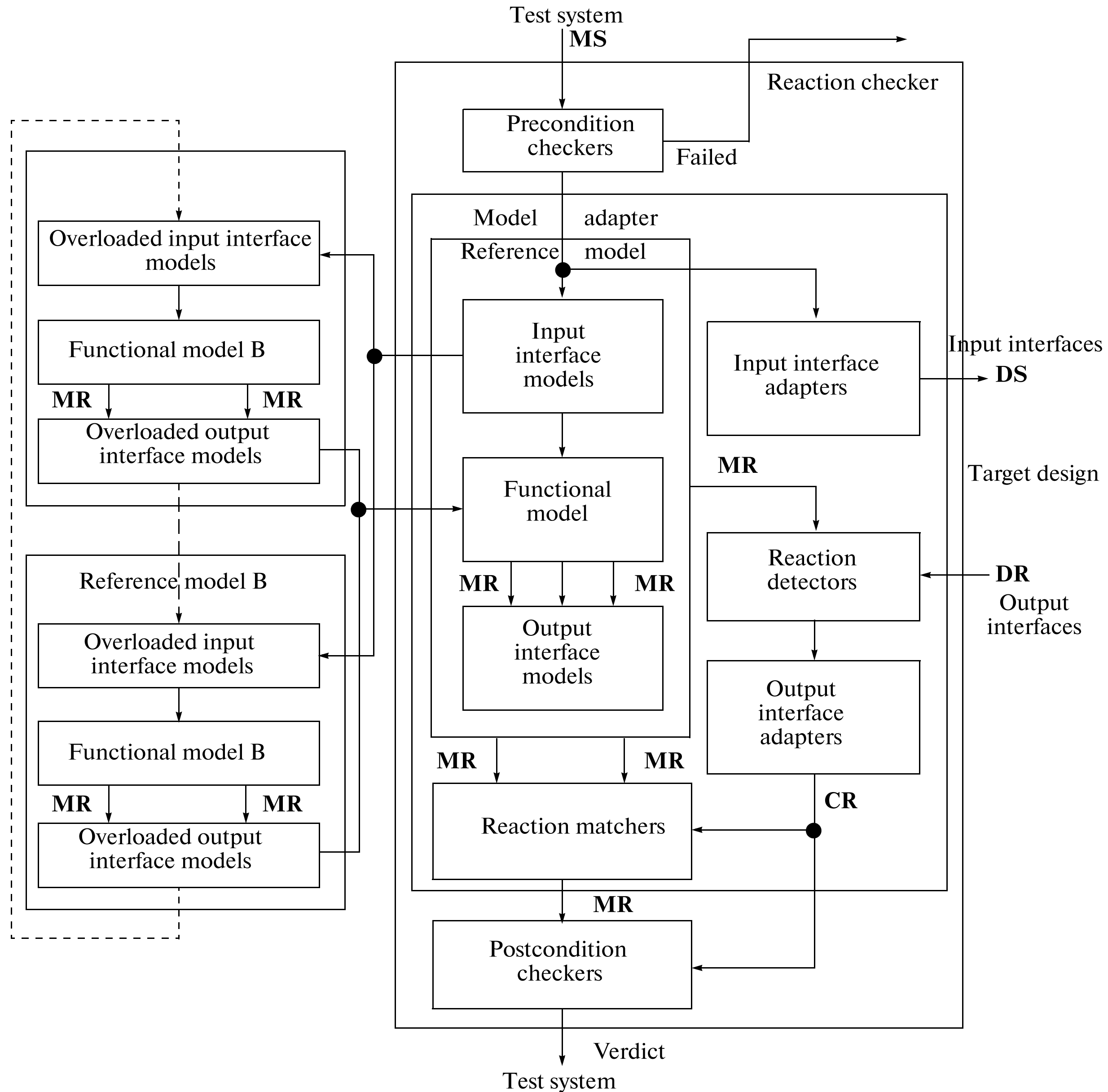

Fig. 5. Architecture of multi-module reaction checkers.

track intersection of the earlier coverage or to create new coverage structures for the whole DUV.

The combined reaction checkers is one of the most difficult parts of combined test systems. The common reaction checker looks like a single reaction checker, but it should use functionality of all included reaction checkers. Only the common reaction checker may change values of DUV's wires, while the sub reaction checkers' input and output interfaces adapters are switched off. The switching them off is possible by means of overloading of methods that send model messages to other reaction checkers rather than to the absent DUV (see Fig. 5).

To facilitate the connection between the reaction checkers, we propose to use *channels*. The channel is a way to connect an output interface model and an input interface model together. To do this, the message from the sender should be translated into a form applicable to the receiver and placed into the latter. This activity can be done by the channels. The channels can also broadcast messages to several receivers. The usage of channels is shown in Fig. 6.

The overloaded input interface models usually contain precondition checkers; therefore, they can check communication protocols between the sub modules. This may help to reveal problems that cannot be detected if the reference model takes input stimuli

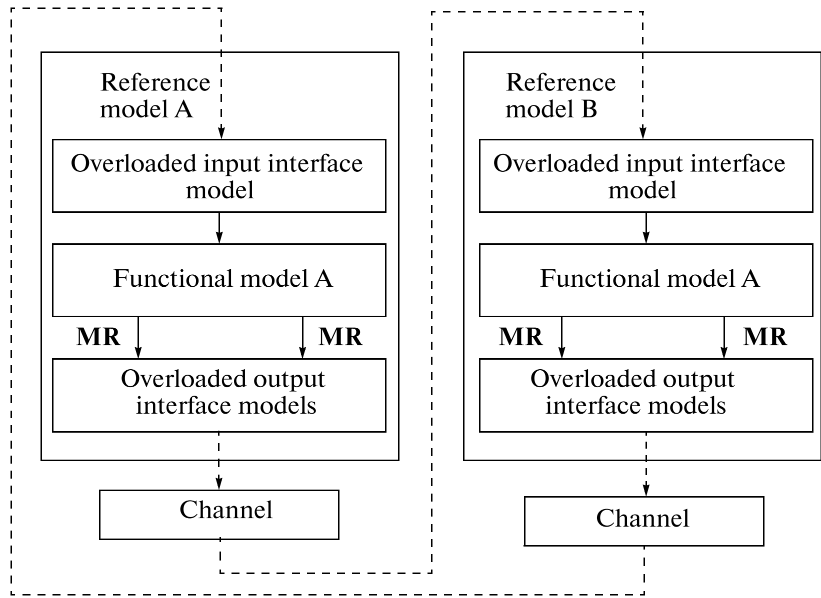

Fig. 6. Usage of channels.

only from the stimuli generator due to a wide range of input variables values. In the case of interconnection, the values assigned are much closer to those in real work situation for the DUV's parts and, at the same time, can be checked.

The approach is supported by the C++-based library and can use some parts of system simulators, which are usually written in C++. Moreover, owing to the accepted architecture, the reference models can be reused in the development of Verilog models as stubs. In this case, input and output interface adapters work in unusual way: input interface adapters take messages from the DUV, send them to the functional model and to the DUV by means of messages via output interface adapters (see Fig. 7).

The common test system controls the reaction checkers built in the DUV. The checkers help the Verilog model developer to speed up development of the hardware model, since the code with the same functionality is usually written quicker in C++ than in Verilog.

Summing up, the approach allows one to develop test systems that can be reused as parts of a common test system. These test systems check not only the output data of DUV but also their input data by means of precondition checkers. When the test systems are connected to each other rather than to the DUV, they can check behavior of their neighbors and, thus, the interconnection protocol of DUV's components. Each test system supports special means that allow it to create interconnections, such as interfaces and channels. After all, to reduce time spent on the development of the first version of DUV for system-level verification, the test systems can be inserted into the Verilog code of the DUV when the latter is still under developing. The approach is supported by a C++-based library, so that the test systems developed according to the approach should also be based on C++. This makes it possible to use many C++ means facilitating the test systems development. It should be noted that C++ is usually used in system-level simulators for DUVs, so that parts of the simulators can easily be reused as reference models in the test systems, and vice versa. Finally, the library is compatible with UniTESK approach. This means that it is possible to develop high-quality tests based on FSM traversing even for the system-level case and to distribute tests among clusters of computers [13].

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

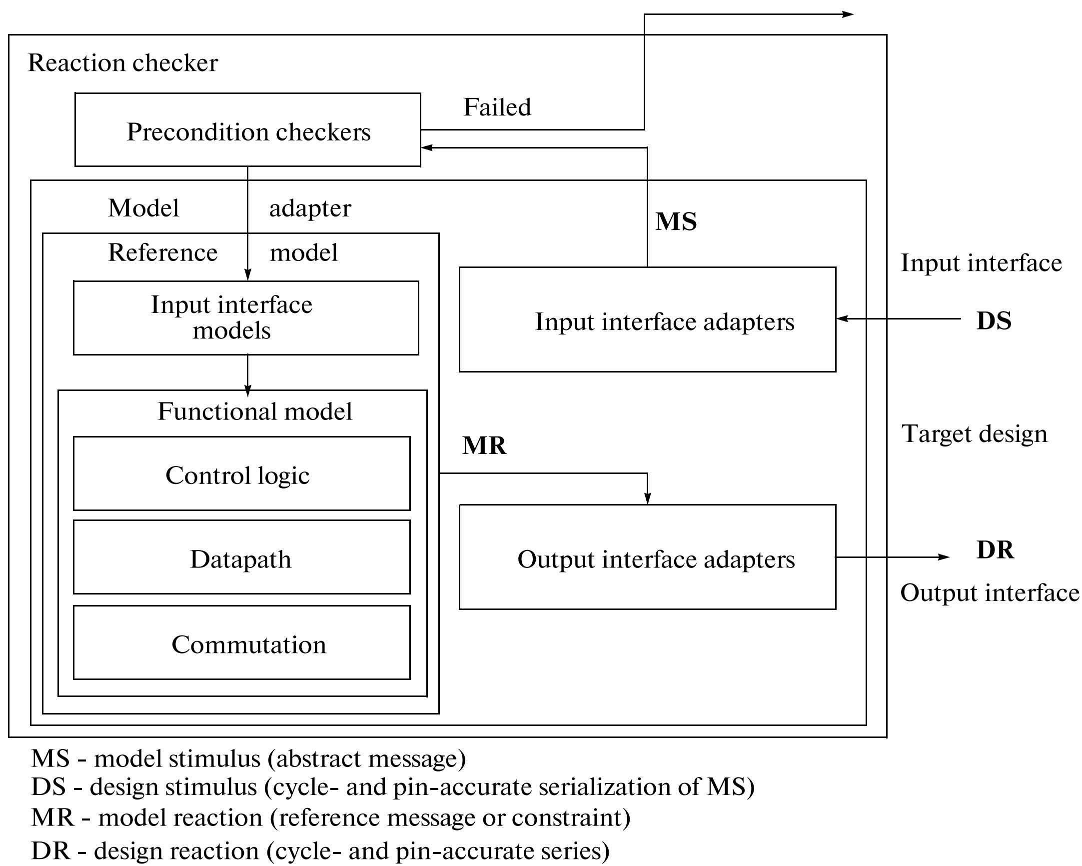

Fig. 7. Architecture of the reaction checker built into a DUV.

5. CASE STUDY

The approach to the single test system development has shown its effectiveness in several projects [4]. The most interesting cases are presented in the table.

The labor costs in the table include verification plan writing, test system development, as well as verification process and debugging of the developed test system. In the case of the non-blocking L2 cache, the costs also include test system maintaining due to permanent modifications in the DUV. The table 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, since it required additional accounting of the permanent changes.

A comparison between our results and the second approach (OVM) could be interesting. We could not do this because of insufficient number of available results of the OVM application. We believe that time required for the development of test systems is about the same in both cases. This assumption is based on the fact that the OVM also utilizes an object-oriented language and the set of test system components looks similar to ours. Nevertheless, our approach provides at the same time additional means of the FSM-based stimuli flow generation. True, we have not analyzed this issue in detail. This is the subject of future studies.

The proposed way of merging is a new revealed ability of the basic approach. By now, only some experiments were conducted to estimate the possibility of merging. First, the test system for a simple FIFO module was developed. This took about two man-days, including efforts spent for documenting the project. Then, three such modules were connected to each other. Two of them were input buffers, and the third module became an output buffer. We placed an arbiter between them to select the input buffer for sending data to the output one. The arbiter always selected the first FIFO if it contained any data. To test this cell, we combined three test systems for the original FIFO modules and added functionality of the arbiter in a very simple way: the functional model of the arbiter always read data from the first FIFO if it contained any data and from the second FIFO otherwise. The stimuli generator used original scenario functions like “pop” and “push” with small modification, since, now, two FIFO could only receive data and the third FIFO could only output data. The interface adapters of the sub test systems were overloaded to provide an opportunity of sending messages via input interfaces of the

output FIFO. Initially, the registration of adapters of interfaces looked as follows:

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

where deserializer is a function translating DUV's wires signals into the design messages. Since the output interface iface3 had lost its output status, we overloaded its adapter:

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

The 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 the output FIFO just sends messages to the common reaction checker's output interface model. The creation of the common test system took us about half a day, including time spent on studying merging possibilities. The time requirement for the development of a common test system from scratch is estimated to be about two days, but it is not of big importance. The time to connect sub test systems slightly correlates with complexity of DUV's sub parts. Mostly, it depends on the number of input and output interfaces and efforts required to connect them together. Our estimate is that connection of one input interface to one output interface takes about one hour. This time is spent on the development of a channel between the interfaces to translate their messages. An average DUV's module, by our estimates, consists of about ten input interfaces and ten output interfaces; hence, connection of two average DUV's modules will take one or two man-days, depending on the productivity of the verification engineer.

6. CONCLUSIONS

The proposed approach provides the researcher with a convenient way of development of test systems for separate parts of DUVs and merging test systems with high level of reuse. The main advantages of the approach are verification of the DUV's parts interconnections without need in writing an additional code and special means of facilitating the reuse. The approach is supported by a library in C++, which allows one to take advantage of rich set of tools available in the language in the course of developing test systems. The fact that system-level simulators are usually based on C++ points to the possibility of reuse of parts of these simulators as reference models of the test systems being developed, and vice versa. The approach has two bonuses. First, the test systems can be parts of components under test and, thus, control communications between functional models inside the DUV. This is especially important when the DUV is under development, since it helps to speed up the onset of its verification. The second bonus is that the library supporting the approach has been developed in accordance with the UniTESK technology, which makes it possible to create high-quality tests based on the FSM-traversing and distribute tests among clusters of computers.

REFERENCES

1. IEEE 1364-2005, Verilog Standard.
2. Bergeron, J., *Writing Testbenches: Functional Verification of HDL Models*, Kluwer, 2003.
3. Tool Called Formality by Synopsys, http://www.synopsys.com/Tools/Verification/FormalEquivalence/Pages/Formality.aspx).
4. Chupilko, M. and Kamkin, A., A TLM-based Approach to Functional Verification of Hardware Components at Different Abstraction Levels, *12th Latin-American Test Workshop*, 2011.
5. Clarke, E., Grumberg, O., and Peled, D., *Model Checking*, MIT Press, 1999.
6. Kneuper, R., Limits of Formal Methods, *Formal Aspects Computing*, 1997, vol. 3, pp. 1-000.
7. Iman, S., *Step-by-Step Functional Verification with SystemVerilog and OVM*, Hansen Brown, 2008.
8. Ivannikov, V.P., Kamkin, A.S., Kuliamin, V.V., and Petrenko, A.K., Application of the UniTESK Technology to Functional Verification of Hardware Models, *Preprint of Inst. of System Programming, Russ. Acad. Sci.*, Moscow, 2005, no. 8 (in Russian). http://citforum.ru/SE/testing/unitesk_hard/, 2006.
9. Open Verification Methodology, http://www.ovmworld.org.
10. Kamkin, A.S. and Chupilko, M.M., Survey of Modern Technologies of Simulation-Based Verification of Hardware, *Programming Comput. Software*, 2011, vol. 37, no. 3, pp. 147–152.
11. A. Barantsev et al., UniTesK Approach to Test Development: Achievements and Prospects, *Proc. of Inst. of System Programming, Russ. Acad. Sci.*, 2004, vol. 5., pp. 121–156. http://citforum.ru/SE/testing/unitesk/, 2004.
12. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic Case, *Programming Comput. Software*, 2004, vol. 30, no. 1, pp. 2–17.
13. Demakov, A., Kamkin, A., and Sortov, A., High-Performance Testing: Parallelizing Functional Tests for Computer Systems Using Distributed Graph Exploration, *Proc. of Conf. Cloud Computation. Education. Research. Developments*, 2011.