ISSN 0361-7688, Programming and Computer Software, 2007, Vol. 33, No. 5, pp. 272–282. © Pleiades Publishing, Ltd., 2007.
Original Russian Text © V.P. Ivannikov, A.S. Kamkin, A.S. Kossatchev, V.V. Kuliamin, A.K. Petrenko, 2007, published in Programmirovanie, 2007, Vol. 33, No. 5.

The Use of Contract Specifications for Representing Requirements and for Functional Testing of Hardware Models

V. P. Ivannikov, A. S. Kamkin, A. S. Kossatchev, V. V. Kuliamin, and A. K. Petrenko

Institute for System Programming, Russian Academy of Sciences, ul. B. Kommunisticheskaya 25, Moscow, 109004 Russia
e-mail: {ivan, kamkin, kos, kuliamin, petrenko}@ispras.ru

Received January 9, 2007

Abstract—Contract specifications in the form of pre- and postconditions are widely used in software engineering for formal description of interfaces of software components. On the one hand, such specifications are convenient for the developers since they can easily be attached to the system architecture. On the other hand, test oracles verifying conformance of the behavior of the target system to the specifications can automatically be generated from them. In the paper, it is suggested to use contract specifications for representing requirements and for functional testing of hardware models developed in languages such as VHDL, Verilog, SystemC, SystemVerilog, etc. An approach to specification of such systems is proposed and compared with the existing methods of hardware specification. An experience of its practical use is described. The approach is based on the UniTESK testing technology developed at the Institute for System Programming.

DOI: 10.1134/S0361768807050039

1. INTRODUCTION

It is impossible to imagine modern world without electronic devices. Mobile phones, digital cameras, and portable computers have become integral attributes of human life. Special devices control operation of household appliances, on-board equipment of planes and spacecrafts, medical life support systems. Almost all these systems rely on semiconductor hardware, namely, integrated circuits consisting of millions of microscopic transistors, which implement the required functions by conducting and transforming electrical current.

To make sure that the hardware works properly, i.e., implements the required functions, functional testing is used. The requirements for the quality of hardware system testing are very high. This is explained not only by the fact that all information and control systems (including those that are very critical to errors and faults) rely on the hardware systems, but also by economic factors. Unlike in the case of software, where correction of an error is relatively inexpensive, an untimely detected hardware error may require product replacement, which is associated with great expenses. For example, the known error in the implementation of the FDIV instruction in microprocessor Pentium[1] [1] consisting in incorrect division of some floating-point numbers cost Intel 475 million dollars [2, 3]. On the other hand, limitations on the testing period are rather severe. It is important not to overextend the process and to release the product in due time, while it is in demand.

How to develop a quality product in due time using limited resources? Currently, electronic devices are designed with the use of high-level modeling languages, which considerably speed up the design process owing to the automated translation of the hardware description at the *register transfer level* (RTL) into hardware description at the *gate level.* Such languages are referred to as *hardware description languages* (HDL), and the models constructed with the help of these languages, HDL or RTL models.[2] The hardware description languages make it possible to considerably improve the efficiency of the development; however, they do not guarantee the lack of errors, so that functional testing remains topical and important.

Given complexity of modern integrated circuits[3], it is impossible to construct an acceptable set of tests in reasonable time. Hence, automated test development technologies are required. Currently, development of such technologies and tools supporting them became a separate branch of *electronic design automation* (EDA) industry, called *testbench automation.*

The basic task of testing is verification of whether the system behavior corresponds to the requirements imposed on it. Such verification can be automated only if the requirements for the system are represented in a *machine-readable form.* Such a representation of the requirements is referred to as *formal specifications*, or simply *specifications.*

[1] Pentium is the trademark of several generations of the microprocessor family x86 manufactured by Intel since March 22, 1993.

[2] It is these models that are subject of study in this paper.

[3] The number of transistors in modern integrated circuits reaches hundreds of millions, and, according to Moore's law, this number doubles every 18–24 months.

In this work, we consider specifications of a specific form, the so-called *contract specifications*. Contract specifications and the design process based on them, Design-by-Contract (DbC), were introduced in 1986 by B. Meyer in the context of software development [4, 5]. The central metaphor of the approach was borrowed from business. Components of a system interact with one another based on mutual *obligations* and *benefits*. If a component provides the environment with some functionality, it may impose a *precondition* on its use, which determines an obligation for the client components and a benefit for it. The component also guarantees execution of a certain action with the help of a *postcondition*, which determines an obligation for it and a benefit for the client components.

The modern style in the design of complex hardware systems is similar to the component approach in software engineering. There appear libraries of ready-to-use components, called *intellectual property cores* (IP cores), which can be *reused* in different projects [6]. Currently, the creation of really complex hardware systems focuses basically on the integration of ready-to-use blocks [7]. Note that, under such an approach, functional testing plays a very important role, since these blocks can be used in quite different environments [7].

Why have we selected contract specifications for our studies? In our opinion, the Design-by-Contract approach is in line with the current trends existing in the hardware design, and, which is also important, contract specifications are very suitable for the testing goals. First, they are convenient for the designers, since they can easily be attached to the system architecture. Second, by virtue of their representation, they aid efforts on creation of *target system correctness criteria* that are independent of implementation. Third, and the most important, they make it possible to automatically construct *test oracles* verifying conformance of the target system behavior to the requirements described in the specifications [8].

The paper is organized as follows. In the next section, hardware models and typical organization of modules of hardware systems are discussed. In the third section, the suggested approach to the representation and verification of requirements based on contract specifications is described. The fourth section contains a brief survey of the testing technology UniTESK and the test development tool CTESK, as well as a description of a method for using the tool for the hardware specification. In the fifth section, the suggested approach is compared with the existing specification methods. The experience of practical use of the approach is described in Section 6. Finally, in Section 7, the conclusions are given, and directions of further studies are outlined.

2. HARDWARE MODELS

Before we describe the suggested approach, we consider specific features of the hardware models developed with the use of languages such as VHDL [9], Verilog [10], SystemC [11], SystemVerilog [12], etc. Knowledge of these features allows us to adequately adapt contract specifications in the form of pre- and postconditions for representing requirements and for functional testing of such systems.

2.1. Specific Features of Hardware Models

Hardware models are systems consisting of several interacting *modules*. Like in programming languages, modules are used for decomposing complex systems into multiplicity of independent or weakly connected subsystems. Each module has an *interface*, a set of *inputs* and *outputs* used for connecting modules with the environment, and an *implementation*, which determines the way the module processes *input signals*, i.e., calculation of values of *output signals* and change of *internal state*.

The processing of input signals by a module is initiated by *events* from the environment side. The events in the hardware models are meant to be any variations of signal levels. Since we consider binary signals, the two following basic event types are distinguished: *positive edge* (*posedge*), change of the signal level from low to high, and *negative edge* (*negedge*), change of the signal level from high to low.[4]

As a rule, each module consists of several statically created *parallel processes*.[5] Each process implements the following cycle: first, one or several events from a given set are waited; then, they are processed; and the cycle is repeated. The set of events waited by the process for processing is called the process *sensitive list*. A process is said to be *passive* if it is in a state of waiting for an event and *active* otherwise.

An important feature of the hardware models is the use of the notion of *time*. Time is modeled by integer numbers, and the physical meaning of the time unit can be specified. To describe cause–effect relations between events occurring during one unit of the model time, the notion of the *delta delay* is used. If there is a delta delay between events, they are executed successively one after another but during the same model time unit.

To run hardware models with the aim of analysis of their behavior, *event-driven simulation* is usually used. In contrast to *time-driven simulation*, where signal values and internal module states are calculated in regular time intervals, in this method, the model is considered only at the moments when some events occur.

[4] Here, we do not consider various undefined values, which are often used in the hardware simulation.

[5] In what follows, they are referred to as *model* processes, to distinguish them from the processes of the operating system.

The *event-driven simulator* operates as follows. At the beginning of simulation, model time is set equal to zero. Next, in the loop, while there are active processes,[6] one of these processes is selected and executed until it becomes passive. After all active processes are executed, the simulator checks whether there are events planned for the current moment plus the delta delay or for the future time moments. If such events exist, the simulator sets the model time equal to the time of the nearest event, implements events planned for this time moment, recalculates the set of active processes, after which the loop is repeated. If there are no such events, the simulation is completed.

2.2. Typical Organization of the Hardware Modules

In what follows, for the sake of convenience, we assume that specification and testing of hardware modules is implemented at the level of separate modules.

Typically, the operation of a module is controlled by a *clock pulse signal*, which will be referred to as *clock signal*, for brevity, or, simply, *clock*. Edges of the cycle signal split continuous time into a discrete set of intervals called *cycles*. What the module has to do during the current cycle is determined by the values of the input signals and the internal module state.

As a rule, some inputs of the module determine the *operation* the module is supposed to perform. These inputs will be referred to as *control inputs.* Other inputs determine *arguments of the operation* and are called *informative inputs.* The modules usually have special operation NOP (*no operation*), which means that no calculations are performed. Operation NOP is usually used for creating time delays between other operations.

The internal module state also consists of two components: *control state* and *informative state.* The control state is used by the module for organizing the process of operation execution. Approaches to the implementation of the module control functions are usually based on finite automata. The informative state represents internal data of the module.

The hardware modules can be organized in different ways. In terms of duration, the operations executed by the modules are classified into *one-cycle* and *multi-cycle* ones. In terms of the operation execution organization, the modules are divided into *modules with one-by-one operation execution, modules with pipeline operation execution*, and *modules with concurrent operation execution.*

Let us consider how a one-cycle operation is executed by the modules. Before a current cycle starts, the environment finds out the code of the operation and arguments at the corresponding module inputs. The execution of the operation by the module starts at the beginning of the cycle. During this cycle, the module performs necessary calculations, changes its internal state, and determines values of the output signals that can be used by the environment starting from the next cycle (Fig. 1).

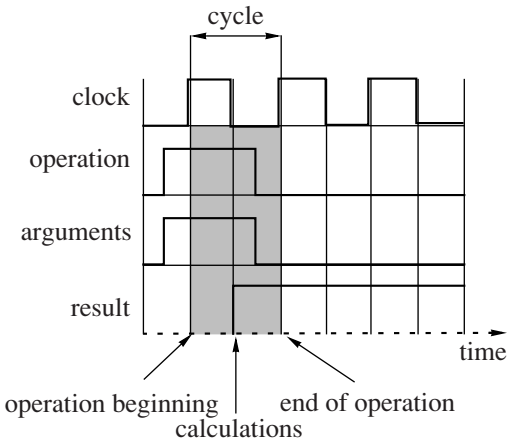

Fig. 1. Time diagram of signals for one-cycle operation.

In contrast to a one-cycle operation, the result of a multi-cycle operation is calculated gradually, cycle by cycle. Let an operation f be executed by a module for n cycles. Then, at each cycle $\tau \in \{1, \ldots, n\}$, the module performs some *microoperation* f_τ, and, after completion of the cycle, the environment gets some partial result. The representation of a multi-cycle operation f as a sequence of microoperations $(f_1, \ldots, f_2)$ is referred to as *temporal decomposition* of f.

Now, let us briefly discuss methods of organization of operation execution. In the modules with the one-by-one operation execution, as can be seen from the name, the execution of a next operation can be started only after complete termination of the previous operation. In the modules with the pipeline operation execution, operations are supplied one after the other not waiting for the termination of the previous operation. In the modules with the concurrent operation execution, several operations can be supplied simultaneously. For modules with the queued and pipeline operation execution, we will use the common name *modules with sequential operation execution*, since, in both cases, the operations are supplied for execution one after the other.

In what follows, we assume that the modules are organized in such a way that simultaneously executed operations do not result in *hazards*; i.e., they do not affect execution of one another. If some operations still may affect one another, the requirements should specify how to supply these operations in order to avoid hazards. For example, the requirement may specify the number of NOP operations inserted between certain instructions.

[6] At the simulation beginning, active processes are those that perform initialization.

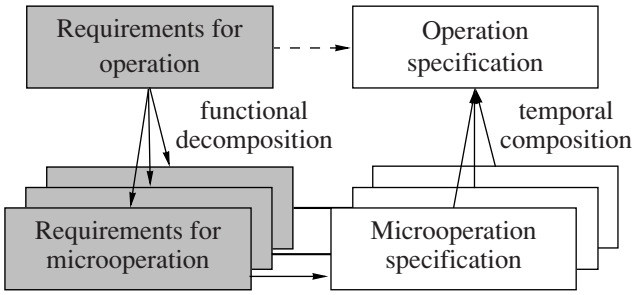

Fig. 2. Construction of a specification for a separate operation.

3. SPECIFICATION AND VERIFICATION OF REQUIREMENTS

As noted in the Introduction, the verification of whether the system behavior conforms to the requirements imposed on it can be automated if the latter are represented in a *machine-readable form.* This representation is called *formal specifications*, or, simply, *specifications*. In this section, we describe the suggested approach to representation and verification of requirements for hardware, which is based on *contract specifications.* First, we consider what kind of requirements for hardware exist.

3.1. Requirements for Hardware Modules

The operations implemented by a module are generally multi-cycle ones; i.e., they are executed by the module during several cycles. There are two basic types of requirements for such operations: *general requirements for operation*, which do not specify cycles when one or another microoperation is executed, and *requirements for temporal operation composition*, which fix cycles during which each microoperation is to be executed.

The general requirements for an operation admit certain freedom in the module implementation. It is not important during which cycle some microoperation is executed; it is important that the result of this microoperation is available for the environment after termination of the operation.

The requirements for the temporal operation composition are more severe. They specify cycles of the microoperation execution. When testing, it usually makes no sense to check whether a microoperation was executed in a certain cycle τ_0. It makes sense to check whether the required values have been assigned to the corresponding outputs of the module after completion of this cycle; it is not important in which particular cycle $\tau \in \{1, \ldots, \tau_0\}$ this was done.

Under such treatment, the general requirements for an operation are a particular case of the requirements for the temporal operation composition; therefore, in what follows, we will not distinguish between these two types of the requirements and simply assume that each requirement is associated with the number of the cycle at the end of which it should be checked.

3.2. Specification of Requirements

The suggested approach to representing requirements is based on the use of contract specifications in the form of pre- and postconditions. In contrast to the classical Design-by-Contract, where contracts are defined at the level of operations, we propose to define contracts for separate microoperations and to obtain the contract for the entire operation by means of *temporal composition* of contracts of separate microoperations (Fig. 2).

Here, under microoperations, we mean some aspects of the operation functionality implemented during one cycle of the module operation, and, under temporal composition of contracts, we mean a specification that, for each microoperation, indicates the number of the cycle at the end of which the corresponding contract is to be executed.

The process of specification of requirements to a separate operation can be outlined as follows. First, the precondition restricting the situation in which the operation can be supplied for the execution is determined. Based on the analysis of the documentation, *functional decomposition* of the operation into a set of microoperations is carried out. For each microoperation, the postcondition describing requirements to it is defined. Then, temporal composition of specifications is carried out: the postcondition of each microoperation is marked by the number of the cycle at the end of which it should be fulfilled. Thus, the contract of the operation f consisting of n microoperations is formalized by the structure $C_f = (pre, \{(post_i, \tau_i)\}_{i = 1, \ldots, n})$.[7]

For the contract $C = (pre, \{(post_i, \tau_i)\}_{i = 1, \ldots, n})$, we introduce the following notation. The operation precondition (pre) is denoted as pre_C; $post_{C, i}$ denotes the postcondition of the i-th microoperation ($post_i$); $\tau_{C, i}$ denotes the number of the cycle at the end of which the postcondition of the i-th microoperation (τ_i) should be fulfilled; and $Post_C(\tau)$ is the conjunction of the microoperation postconditions marked by the cycle τ, i.e., $\wedge\{post_{C, i} | \tau_{C, i} = \tau\}$.

3.3. Verification of Requirements

After the requirements to the module have been formalized, the verification of the conformance of the module behavior to them can be implemented automatically in the course of testing.

Suppose that, at some time moment t, the module being tested performs m operations $f_1, \ldots, f_m$ that have been supplied for the execution $\tau_1, \ldots, \tau_m$ cycles earlier, respectively ($\tau_i \geq 1$, $i = 1, \ldots, m$). Let $C_1, \ldots, C_m$ be contracts of the operations $f_1, \ldots, f_m$, respectively, and let, at

[7] For demonstrativeness, we do not introduce data model and do not specify signature of the pre- and postconditions.

the moments when the operations $f_1, \ldots, f_m$ were supplied, the preconditions $pre_{C1}, \ldots, pre_{Cm}$ have been fulfilled. Then, to verify correctness of the module behavior at the moment t, it is required to check satisfiability of the predicate $Post_{C1}(\tau_1) \wedge \ldots \wedge Post_{Cm}(\tau_m)$.

Clearly, to verify the conformance of the module behavior to the requirements, it is important to be able to construct "good" test sequences. However, this problem is beyond the scope of this work.

4. UNITESK TESTING TECHNOLOGY

For the base approach in this work, we use the UniTESK testing technology [13] developed in the department of programming technologies of the Institute for System Programming of Russian Academy of Sciences [14]. Typical features of this technology is the use of contract specifications in the form of pre- and postconditions of interface operations and invariants of data types for specifying the requirements, as well as the use of generalized finite-automaton models for constructing test sequences.

4.1. Architecture of the UniTESK Test System

The architecture of the UniTESK test system [15] was developed on the basis of the multiyear experience of testing industrial software of different complexity in different subject domains, which made it possible to develop flexible architecture based on the following division of the testing problem into subproblems:

- construction of a test sequence aimed at achieving the desired coverage,
- construction of a single test action in the framework of the test sequence,
- establishing relationship between the test system and implementation of the target system,
- verification of the correctness of the system behavior in response to a single test action.

Each of these subproblems is solved by means of special components of the test system (Fig. 3). The test sequence and single test actions are constructed by a *test engine* and *test action iterator*. Correctness of the behavior of the target system is verified by a *test oracle*, and the *mediator* establishes relationship between the test system and implementation of the target system. Consider these components in more detail.

The *test* engine is a library component of the UniTESK test system; together with the test action iterator, it is designed for constructing the test sequence. The test engine is based on the algorithm for traversing the state graph of the *generalized finite-automaton model* of the target system (a finite automaton modeling the target system at a certain abstraction level). The test engines implemented in the UniTESK tools libraries require that the generalized finite-automaton model of the target system be deterministic[8] and have strictly connected state graph.

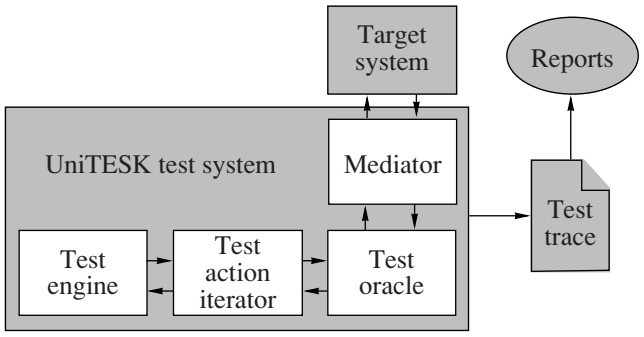

Fig. 3. Architecture of the UniTESK test system.

The **test action iterator** works under control of the test engine and is designed for searching for admissible test actions in each achievable state of the finite automaton. The iterator of test actions is generated automatically from the test scenario, which is an implicit description of the generalized finite-automaton model of the target system.

The **test oracle** estimates correctness of the target system behavior in response to a single test action. It is generated automatically from the formal specifications describing requirements to the target system in the form of pre- and postconditions of the interface operations and data type invariants.

The **mediator** relates abstract formal specifications describing requirements to the target system to the particular implementation of the target system. The mediator transforms a single test action from the specification representation to the implementation representation and the response obtained from the implementation representation to the specification representation. In addition, the mediator synchronizes the state of the specification with the state of the target system.

The **test trace** reflects events occurring in the course of testing. Based on the trace, it is possible to automatically generate various reports helping to analyze testing results.

4.2. Test Development Tool CTESK

The CTESK tool [13] used in this work is an implementation of the UniTESK concept for the C language. It uses language SeC (specification extension of C), which is an extension of ANSI C, for the development of components of the test system. The CTESK tool includes a translator from SeC to C, test system support library, library of specification types, and report generators.

[8] Except for the *ndfsm* test engine [16], which traverses state graphs for some class of nondeterministic finite automata.

```
// temporal composition of microoperations
void operation_time_comp(…)
{
    Operation *descriptor = create_operation(…);

    // addition of stimulus to the queue of stimuli
    register_stimulus(create_stimulus(time, descriptor)));

    // addition of reactions to the queue of reactions
    register_reaction(micro1_return, tick1, descriptor);
    …
    register_reaction(micron_return, tickn, descriptor);
}
```

Fig. 4.

Components of the UniTESK test system are implemented in the CTESK tool with the help of the following special functions of the SeC language:

• *specification functions* containing specification of the immediate response of the target system to a single test action and definition of the structure of the test covering;

• *postponed response functions* containing specifications of the postponed reactions of the target system;

• *mediator functions* relating the specification functions to the test actions on the target system, as well as reactions of the target system to the postponed response functions;

• *function calculating generalized state*; this function calculates the state of the generalized finite-automaton model of the target system;

• *scenario functions* describing the set of test actions for each achievable generalized state.

In the works [17, 18], it is described in detail how to extend the base architecture of the UniTESK test system to functional testing of the hardware models developed in Verilog and SystemC. These works also give technical details of using the CTESK tool for functional testing hardware models.

4.3. Use of CTESK for Specification of Hardware

The test development tool CTESK provides the designer with rather universal means for specification of systems with asynchronous interface [16]. These means were adapted for specification of the hardware modules. Consider the process of specification development in more detail.

For each operation implemented by the module, the specification function is written, which defines the operation precondition and the structure of the test coverage. The postcondition of the specification function usually returns `true`, since all checks are defined, as a rule, in postconditions of the microoperations.

```
// operation specification
specification void operation_spec(…)
{

   // operation precondition
   pre {…}
   // structure of test    // covering
   coverage C {…}
   // operation postcondition
   // usually returns true
   post { return true; }

}
```

For each microoperation included in the specified operation, the postponed response function defining its postcondition is written. For the sake of convenience, the type of the returned response value should contain operation arguments in order that they could be used in the postcondition:

```
// microoperation specification
reaction Operation* micro_return(void)
{

   // microoperation postcondition
   post {…}

}
```

Then, the *microoperation temporal composition function* is defined. This function, first, adds a stimulus (an operation together with a set of arguments) to the *queue of stimuli* with indication of time required for processing the stimulus (*time*) and, second, for each microoperation, adds the corresponding reaction to the *queue of reactions* with indication of the number of the cycle (relative to the current one) at the end of which the reaction should be checked ($tick_i$), see Fig. 4.

The queue of stimuli contains stimuli executed by the module at the current moment. For each stimulus in the queue, the time during which it has been processing is stored. The queue of reactions contains the microoperations that have not been completed yet. For each microoperation, the time is stored when it supposed to be completed and the reaction can be verified. Time

update is implemented by the *time offset function.* After the time has been updated, *functions of processing the queues of stimuli and reactions* are invoked. The function of processing the queue of stimuli removes completely processed stimuli from the queue. The function of processing the queue of reactions registers postponed reactions for all completed microoperations, which are then deleted from the queue.

To illustrate the syntax of the SeC language, we give a simple example. We consider a device called an 8-*bit counter* (Fig. 5).

The interface of the counter consists of two binary inputs `clk` and `rst` and one eight-bit output register `cnt`. If the level of the signal `rst` is low, the signal edge of the cycle pulse `clk` increases the counter `cnt` modulo 256; otherwise, the counter is assigned zero value.

Figure 6 shows the specification function describing the operation of increasing the counter, i.e., the behavior of the counter in response to the edge `clk` for low level of signal `rst`. Since this operation is simple, the specification is executed without using postponed reaction functions.

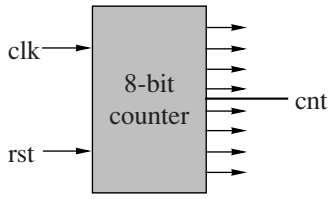

Fig. 5. A schematic of inputs and outputs in the 8-bit counter.

5. COMPARISON WITH THE EXISTING APPROACHES

In this section, the proposed approach is compared with the existing specification methods supported in the modern hardware verification languages (HVL).

Hardware verification languages, among which are PSL, OpenVera, SystemVerilog, and others [19], include constructs from hardware description languages, programming languages, and special verification-oriented means. The latter include, in particular, means for behavior specification, determination of the structure of the test covering, and generation of test data. Note that we compare here only the specification means.

The behavior specification means used in modern hardware specification languages are based on linear temporal logic (LTL) and/or computation tree logic (CTL) [19]. The languages (at least, in what concerns specifications) originate from ForSpec (Intel) [20] (for languages based on the LTL logic[9]) and Sugar (IBM) [21] (for languages based on the CTL logic). Figure 7 shows interrelation of some hardware verification languages.

Note that the CTL logic is used basically for formal verification of systems. For the simulation and testing purposes, of great interest is the LTL logic. Since all hardware verification languages based on the LTL have similar specification means, the proposed approach is compared with only one of them, namely, OpenVera [22]. This language is supported by many tools and, additionally, is open.

OpenVera was developed in 1995 by Systems Science, and its original name was Vera. In 1998, Synopsys bought Systems Science. In 2001, Synopsys renamed the language into OpenVera and made it open. For specification of behavior, OpenVera provides a special language for formulation of *temporal assertions*, which is called OVA (OpenVera assertions) [23, 24].

OVA operates with *sequences of events limited in time*, for which it is possible to address both the past and future. From simple sequences, one can construct more complicated ones using *logical operations* AND and OR or by means of *regular expressions.* The language has means for combining temporal assertions into *parameterized specification libraries.* Let us illustrate syntax of OVA on the following simple example.

[9] The LTL logic used in ForSpec is called FTL (ForSpec temporal logic).

```
// specification of the counter increase operation
specification void increment_spec(counter_8bit *counter)
    updates cnt = counter->cnt,
            rst = counter->rst
{
    // operation precondition
    pre { return rst == false; }
    // determination of structure of test coverage
    coverage C { return { SingleBranch, ''Single branch'' }; }
    // operation postcondition
    post {  return cnt == (@cnt + 1) % 0xff; }
}
```

Fig. 6.

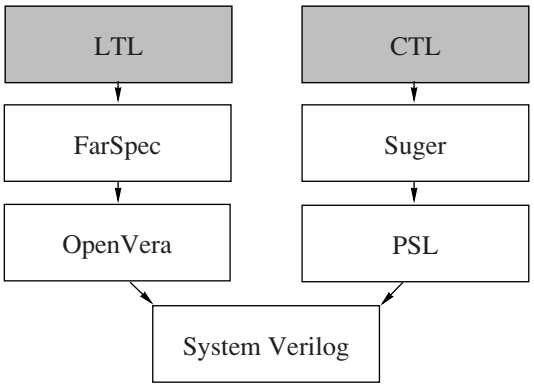

Fig. 7. Interrelation of the hardware verification languages.

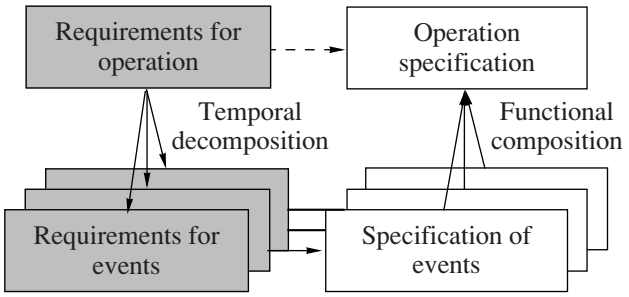

Fig. 8. Construction of a specification in the approaches based on temporal logic.

```
// cycle signal
clock negedge(clk)
{
    // limit counter values
    bool cnt_00: (cnt == 8'h00);
    bool cnt_ff: (cnt == 8'hff);

    // counter overflow event
    event e_overflow: cnt_ff #1 cnt_00;
}

// assertion forbidding
// counter overflow
assert a_overflow: forbid(e_overflow);
```

In this example the event of counter overflow `e_overflow` is defined, and assertion `a_overflow` forbids this event.

Table

Operation	Microoperation	The number of requirements	
		Precondition	Postcondition
Entry reading	–	5	2
Entry writing	–	5	4
Verification of sell availability	–	5	3
Translation of data address	DTLB hit	5	18
	DTLB miss		15
Translation of instruction address	ITLB hit	5	18
	ITLB miss		14
Total number			
5	7	99	
		25	74

In the OpenVera approach, as well as in other approaches based on temporal logics, the focus is placed on *temporal decomposition* of operations. For each operation, its temporal structure—admissible sequences of events and delays between them—is first determined. Then, the predicates describing separate events are defined. Finally, the predicates corresponding to the same time moment are grouped somehow if necessary (Fig. 8).

In our approach, the focus is placed on *functional decomposition* of operations. First, functional structure of the operation, i.e., the set of microoperations, is determined. Then, each microoperation is specified, and temporal specification composition is carried out (Fig. 2).

We assume that the functional structure of an operation is more stable compared to the temporal structure. Hence, the approaches based on the functional decomposition of operations make it possible to develop specifications that are more robust with respect to modifications of the implementation compared to the approaches based on the temporal decomposition.

The advantageous features of the proposed approach are also its clarity and simplicity. The pre- and postconditions are usually simpler than temporal logic formulas and do not require special knowledge from the test designer.

It should be noted that contract specifications use the state model of the target system to describe system behavior. After certain extension, this model can be used for construction of test sequences, e.g., with the help of a graph traversal algorithm, as it was done in the UniTESK testing technology.

6. EXPERIENCE OF PRACTICAL APPLICATION OF THE APPROACH

The proposed approach was applied to testing the *translation lookaside buffer* (TLB) of a microprocessor with MIPS64-compatible architecture [25, 26].

The translation lookaside buffer, which is a part of the majority of modern microprocessors, is designed for caching the *page table*, the table of the operating system that stores the mapping of *virtual* onto *physical pages* of memory. The use of this buffer considerably speeds up translation of the addresses. The buffer is an associative memory with a fixed number of entries. In addition to the address translation interface, it provides an interface for reading and modifying the content of the memory.

Briefly, translation of a virtual address is implemented as follows. If the buffer contains an entry with the desired virtual page number, the corresponding physical address is formed in a certain output register: the number of the virtual page is replaced by that of the physical page, and the offset remains unchanged. Otherwise, the signal indicating that there was a buffer miss is set in one of the outputs of the module.

6.1. Module Structure and Functionality

Consider the structure of the tested translation lookaside buffer. The memory of the TLB consists of 64 entries, which constitute the *joint TLB* (JTLB). In addition, for the sake of the performance improvement, the module contains two additional buffers: *data TLB* (DTLB) and *instruction TLB* (ITLB). The DTLB is used for translation of data addresses, and the ITLB, for translation of instruction addresses. Both buffers contain four entries each; their content is a subset of the JTLB and is updated by replacing the last recently used (LRU) entry.

Each entry of the TLB is conditionally divided into two sections: *section–key* and *section–value*. The section–key includes the *memory segment specifier* (two leading bits of the virtual address, *R*), *number of the virtual page divided by 2* (VPN2), *process identifier* (ASID), *global address translation bit* (*G*), and *page mask* (MASK). The section–value consists of two subsections. Each subsection contains the *physical page number* (PFN_i), *reading permission bit* (V_i), *writing permission bit* (D_i), and the *page* cashing policy (G_i). Which subsection will be used in the address translation is determined by the minor bit of the virtual page number.

The interface of the translation lookaside buffer consists of 30 inputs (16 general-purpose inputs, 3 DTLB inputs, 4 ITLB inputs, and 7 JTLB inputs) and 31 outputs (6 general-purpose outputs, 9 DTLB outputs, 8 ITLB outputs, and 8 JTLB outputs).[10] The module functionality includes operations of reading, writing, checking availability of an entry in the buffer, as well as operations of translation of data and instruction addresses. The RTL – model of the module is developed in the Verilog language and contains approximately 800 lines of code.

[10] When counting the numbers of inputs and outputs, the joint test action group (JTAG) interface, which is a standard interface used for hardware testing, is not taken into account.

6.2. Module Specification Development

The basic requirements for the translation lookaside buffer were obtained from the module designers in a written form. The requirements were elaborated and formalized in the course of discussions with the designers and reading of technical documentation. It should be noted that the requirements formulated by the designers could easily be represented in the form of pre- and postconditions.

At the first stage of specification development, the subject domain model has formally been defined. First, basic notions, such as virtual and physical addresses, memory segment, microprocessor operation mode, and the like, were defined. For these notions, the corresponding data types and auxiliary functions of operations on them—determination of a memory segment by a virtual address, calculation of the offset and the number of an address page, verification of the memory segment access rights for the given microprocessor operation mode, and the like—were introduced. Then, data types for more complicated notions (TLB entry, JTLB, DTLB, ITLB buffers) and the corresponding functions (matching a virtual address to a TLB entry, searching an entry with desired properties, address translation, and the like) have gradually been defined. The result of this stage was a system of types that completely described the module and a set of functions in terms of which the requirements were defined.

The further process was related to structuring the requirements. Each operation implemented by the module was represented as a set of microoperations: one microoperation for each operation of reading, writing, and verification of availability of an entry in the buffer, and two microoperations for the operations of translation of data and instruction address. Text description of the requirements was split into atomic requirements, each of which was formally described as a predicate in terms of the functions defined earlier. Decomposition of the operations implemented by the module into microoperations and distribution of the atomic requirements over operation preconditions and microoperation postconditions are shown in the table.

6.3. Results of Application of the Approach

The project has demonstrated the convenience and comparatively low cost (in terms of working hours) of development of contract specifications in the form of pre- and postconditions for hardware models. It took approximately 2 weeks for one person to develop all specifications consisting of 2500 lines of code in the SeC language. A great portion of the specifications (about 40%) related to definition of the data types and functions of operation on them can be used for formal description of a translation lookaside buffer for any MIPS64-compatible microprocessor. The specifications developed turned out to be modular, well structured, descriptive, and easy to modify (usually modifi-

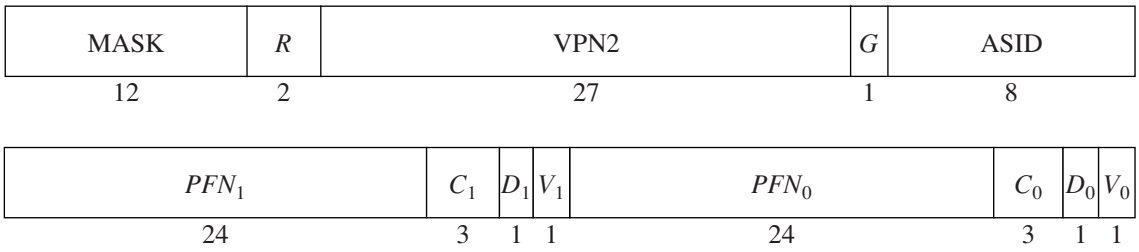

Fig. 9. Structure of a TLB entry (section–key and section–value).

cations are associated with the temporal composition of the microoperations). It should be noted also that the project has revealed about ten errors, including critical ones, in the model implementation.

7. CONCLUSIONS

The contract specifications have originally been proposed for description of interfaces of software components; however, after certain elaboration, they can be used for description of hardware modules. On the one hand, such specifications are convenient for the designers since they can easily be attached to the system architecture. On the other hand, test oracles verifying conformance of the target system behavior to the requirements described in the specifications can automatically be generated from them. Practical verification of the approach in the project on testing a microprocessor translation lookaside buffer has demonstrated convenience of representing requirements to the hardware in the form of pre- and postconditions and shown that the development of the specification requires relatively little efforts.

Currently, we have acquired certain experience of using the UniTESK testing technology and the CTESK tool for specification and testing of hardware models [17, 18]. Our experience shows that some stages of test development can completely or partially be automated. Detailed study of this issue and creation of tools for automated test development are basic directions of future work.

REFERENCES

1. Statistical Analysis of Floating Point Flaw in the Pentium Processor, *Intel Corporation*, 1994.
2. Beizer, B., The Pentium Bug—An Industry Watershed, *Testing Techniques Newsletter (TTN). Online Edition,* 1995.
3. Wolfe, A., For Intel, It's a Case of FPU All Over Again, *EE Times*, 1997.
4. Meyer, B., Design by Contract, *Tech. Report TR-EI-12/CO*, Interactive Software Engineering Inc., 1986.
5. Meyer, B., Applying "Design by Contract", *IEEE Comput.*, 1992, vol. 25, no. 10.
6. Keating, M. and Bricaud, P., Reuse Methodology Manual for System-on-a-Chip Designs, *Kluwer*, 2002.
7. Nemudrov, V. and Martin, G., *Sistemy-na-kristalle. Proektirovanie i razvitie* (Systems-on-a-Chip. Design and Development), Moscow: Tekhnosfera, 2004.
8. Barantsev, A.V., Bourdonov, I.B., Demakov, A.V., Zelenov, S.V., Kossatchev., A.S., Kuliamin, V.V., Omel'chenko, V.A., Pakulin, N.V., Petrenko, A.K., and Khoroshilov, A.V., UniTesK Approach to Test Development: Achievements and Perspectives, http://www.citforum.ru/SE/test-ing/unitesk/.
9. IEEE Standard VHDL Language Reference Manual, *IEEE Std 1076-1987.*
10. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, *IEEE Std 1364-1995.*
11. http://www. systemc.org.
12. http://www. systemverilog.org.
13. http://www. unitesk.com.
14. http://www.ispras.ru.
15. Bourdonov, I., Kossatchev, A., Kuliamin, V., and Petrenko, A., UniTesK Test Suite Architecture, *Lecture Notes in Computer Science* (FME'2002), Springer, 2002, vol. 2391.
16. Khoroshilov, A.V., Specification and Testing of Systems with Asynchronous Interface, *Preprint of Inst. of System Programming RAN*, Moscow, 2006, no. 12, http://citforum.ru/SE/testing/asynchronous_inter-face/.
17. Ivannikov, V.P., Kamkin, A.S., Kuliamin, V.V., and Petrenko, A.P., Applying UniTESK Technology to Functional Testing of Hardware Models, *Preprint of Inst. of System Programming RAN*, Moscow, 2005, no. 8, http://citforum.ru/SE/testing/unitesk_hard/.
18. Kamkin, A., The UniTESK Approach to Specification-Based Validation of Hardware Designs, *IEEE-ISoLA'2006: The 2nd Int. Symp. on Leveraging Applications of Formal Methods, Verification and Validation*, 2006.
19. Edwards, S.A., Design and Verification Languages, *Tech. Report*, New York, Columbia University, 2004.
20. Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M., and Zbar, Y., The ForSpec Temporal Logic: A New Temporal Property-Specification

Language, in *Tools and Algorithms for Construction and Analysis of Systems*, 2002.

21. Beer, I., Ben-David, S., Eisner, C., Fisman, D., Gringauze, A., and Rodeh, Y., The Temporal Logic Sugar, *Lecture Notes in Computer Science*, 2001.
22. http://www.open-vera.com.
23. OpenVera® Language Reference Manual: Assertions, Version 1.4.1, 2004.
24. OpenVera® Assertions. Blueprint for Productivity and Product Quality, 2003, http://www.synopsys.com/products/ simulat-ion/ova_wp.html.
25. http://www.mips.com/content/Products/Architecture/MIPS64.
26. MIPS64T Architecture for Programmers, Revision 2.0, MIPS Technologies Inc., 2003.