Specification-Driven Construction of Testbench Checkers for RTL Models of Synchronous Parallel-Pipeline Hardware

Alexander Kamkin
Institute for System Programming of the Russian Academy of Sciences
25, A. Solzhenitsyn Street, Moscow, 109004, Russia
E-mail: kamkin@ispras.ru

***Abstract*— In this paper a method for constructing testbench checkers for RTL models of synchronous parallel-pipeline hardware is considered. The method is based on description of control flow graphs of the design's operations and on formal specification of the microoperations with the help of Hoare triples. The method allows to specify and to automatically construct testbench checkers for complex synchronous designs with control flow branching, parallel starting operations, etc.**

***Keywords*— simulation-based verification, formal specification, testbench automation, assertions, co-simulation**

I. Introduction

Functional verification is one of the most difficult and labor-intensive activities in hardware design process. According to statistics, ensuring the functional correctness of hardware consumes about 70% of the design efforts [1]. In this connection, it is clear why semiconductor industry is so interested in verification automation. The usage of automated approaches reduces verification effort, shrinks time-to-market, and, simply, saves money.

Two main approaches to functional verification of hardware designs are based on formal and simulation techniques [2]. It is well known that formal methods are exhaustive, but do not scale well, while simulation-based approaches are scalable, but are not exhaustive [3]. A reasonable compromise is provided by so-called semi-formal methods combining formal specifications and simulation.

In this work a semi-formal approach to testbench automation is suggested. A testbench is an environment used to verify the design correctness via simulation. A typical testbench has three key components: a stimulus generator, a response checker, and a coverage tracker. The stimulus generator creates input stimuli to the design under verification. The response checker estimates the correctness of the design behavior. The coverage tracker evaluates the test completeness.

The paper introduces the method of automated specification-driven construction of response checkers. It is applicable to synchronous parallel-pipeline designs with complex organization. The method is based on description of the operations' control flow and on formal specification of the microoperations with the help of pre- and post-conditions. The work is a part of research on testbench automation being done at Institute for System Programming of the Russian Academy of Sciences (ISPRAS) [4], [5].

The rest of the paper is organized as follows. Section II describes the existing approaches to response checking including self-checking tests, assertions, and co-simulation. In Section III the suggested method is considered. This section consists of four parts which describe how specifications look like, how they are interpreted, how connection between specifications and an implementation is organized, and, finally, how testbench checkers are constructed. Section IV is a case study. Section V concludes the paper.

II. Response Checking Approaches

Modern digital hardware has rather complex organization including pipelining, branching of control flow, etc. Our goal is to automate construction of testbench checkers for such kinds of designs. Nowadays, there are three basic ways used for response checking: self-checking tests, assertions, and co-simulation.

A. Self-Checking Tests

Self-checking tests are an approach to testbench organization in which each stimulus (test case) is encoded with a procedure of checking for expected results [6]. In this approach, each test case should perform response checking during and at the end of the test. This has certain disadvantages. First, it is hard to write test cases that make checks after sophisticated test sequences. Second, test cases require maintenance during the design process to keep up with the design changes. Finally, the self-checking approach suffers from incomplete checking, because each test case checks only few aspects of the design behavior.

B. Assertions

Assertions are statements about a design's intended behavior, which must be verified [7]. In such approach, checks are detached from stimuli and injected into RTL code (or written in separate files). There is no need to have hand-written test cases that check for specific results. Instead, automatic test generation can be used for the design verification. Assertions usually state the most critical or the most obvious properties.

Therefore assertion-based checking usually lacks for completeness. In the case of built-in assertions, it is impossible to write checks until RTL model of the design is developed.

C. Co-Simulation

Co-simulation is a method of response checking in which an independent executable model (reference model) is used along with the target RTL description [6]. The two models are co-simulated using the same test sequences and their results (execution traces) are compared for equality. Every mismatch is tracked down to discover which of the models is incorrect. The usage of a reference model allows to generate tests automatically. Making two models agree for all test sequences is a difficult task, which in many cases is tantamount to writing two RTL models.

D. Analysis

Let us analyze weaknesses of the approaches described above. Self-checking tests lack for high level of automation and for completeness of checking. Furthermore, they are hard to maintain. Usage of assertions is a perfect solution for checking a few numbers of properties, but is not suitable for gap-free checking of complex designs. Co-simulation does not suffer from incompleteness, but it requires development of a detailed executable model to be used as a reference. This is a labor-intensive and time-consuming task.

III. Suggested Method

Our approach to response checking is based on cycle-accurate formal specification of the design behavior. Each operation of the design is described with the help of a control flow graph having two types of nodes: control nodes and operational nodes. Control nodes specify control flow branching, creation of concurrent threads, etc. Operational nodes describe one-cycle microoperations.

A. Specification of Design

State of the target design is formalized by a set of variables. Among the variables, there are input and output parameters of the operations. All predicates and functions mentioned below are assumed to be defined over that set of variables. Functionality of the design is described for separate operations. Specification of an operation includes its precondition, which restricts situations in which the operation is permitted to be started. If the precondition does not hold, then the result of the operation is unpredictable. In general case, execution of an operation requires several cycles. A one-cycle part of an operation is called a *microoperation*.

Structure of an operation's control flow is described by a directed graph in which four kinds of nodes are admitted: $cond$, $fork$, $join$ (control nodes), and $stage$ (operational nodes). Obviously, control flow graphs must fulfill certain requirements, e.g., out-degree of a $stage$ node is not greater than one, a $cond$ node has exactly two outgoing edges, which are marked by $true$ and $false$, etc. An example of an operation control flow graph is shown on Fig. 1. The graph

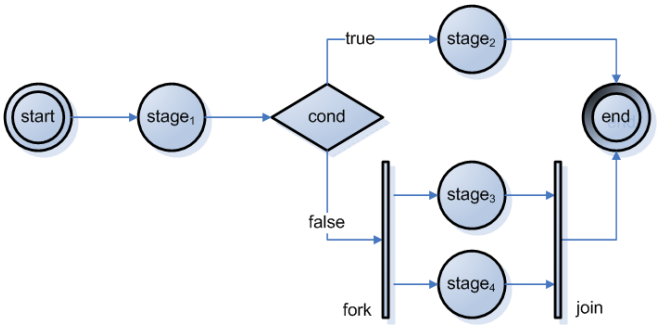

Fig. 1. An example of a control flow graph

on the picture consists of nine nodes: one $cond$, one $fork$, one $join$, and six $stages$ including initial and final stages ($start$ and end).

If in-degree of a stage node is equal to zero, the node is called *initial*. If out-degree is equal to zero, the node is called *final*. An operation is allowed to have more than one initial stage and more than one final stage. Each $cond$ node is supplied with a predicate that represents a condition for choosing control flow direction. Each $stage$ node is described by a *Hoare triple* (*microoperation contract*) $\{P\}C\{Q\}$, where P is a *precondition*, C is a *command*, and Q is a *postcondition* [8]. Semantics of a microoperation precondition is as follows. If the precondition is not satisfied, it does not mean that the result of the microoperation is unpredictable. It simply indicates that the microoperation is interlocked (it will be unlocked, when the precondition becomes true).

To define which operations can be started in parallel, a notion of *execution channels* is used. Each channel is associated with a set of operations that can be executed via that channel. A channel can handle only one operation per cycle. Sets of operations for different channels are able to have a non-empty intersection. Moreover, each channel supports an empty operation nop. Operations associated with different execution channels can be started simultaneously, if that combination of operations is permitted.

Let k be a number of execution channels and $X_1, ..., X_k$ be sets of operations associated with the channels. Possibility of parallel starting of k operations via different execution channels is defined by a Boolean function $sim : X_1 \times ... \times X_k \to \{true, false\}$. If the design has several execution channels, one can define a generalized operation, which is a k-tuple $(x_1, ..., x_k) \in X_1 \times ... \times X_k$ such that $sim(x_1, ..., x_k) = true$. The precondition of the generalized operation is a conjunction of the preconditions of $x_1, ..., x_k$.

Specifications of the described type are called *cycle-accurate contract specifications*. Originally, they were introduced for linear multi-cycle operations to specify pipelined designs [9].

B. Interpretation of Specifications

Execution of a microoperation requires exactly one cycle. Creation of parallel threads, checking branch condition and making decision which branch should be taken are performed

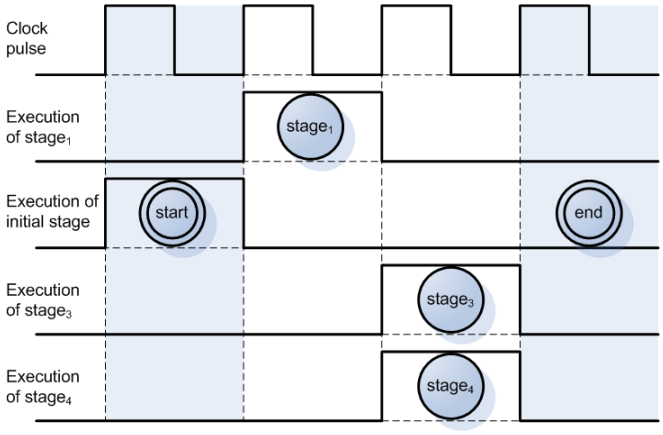

Fig. 2. Cycle-by-cycle execution of the operation

instantly. To interpret specifications, a *set of current stages* is used. At the beginning of simulation the set is empty. On every cycle of simulation, a generalized operation with satisfied precondition is applied to the design. All initial stages of the operation are added into the set of current stages. After that, the set of active stages is calculated among the current stages. A stage is called *active*, if its precondition P is true; otherwise a stage is called *interlocked*. For each active stage the corresponding command C is executed. In theory, the commands of the active stages are executed in parallel. After execution, the postconditions Q of the active stages are checked. Then, the active stages are removed from the set of current stages. If an active stage is not final, then its successive stages are added into the set. A successor relation is defined on the base of the control flow graphs. Given a stage s and an outgoing edge e, define the set of successive stages. Depending on which type of the node the edge leads to, there are four different cases:

- If e leads to a $stage$ node, then that stage is a successive one.
- If e leads to a $cond$ node, then the branching condition is estimated and choice among two alternative edges is made. The same algorithm is applied recursively for the chosen edge.
- If e leads to a $fork$ node, then the algorithm is applied recursively for each outgoing edge of the node. Then, the calculated sets of successive stages are unified.
- If e leads to a $join$ node, then for each ingoing edge the finishing condition of the corresponding thread is checked. If all threads are finished, the algorithm is applied recursively for the outgoing edge of the node; otherwise the set of successive stages is empty (the node will be processed again when another ingoing thread is finished).

C. Connection between Specifications and Implementation

Fig. 2 shows cycle-by-cycle execution of the operation presented on Fig. 1. It is assumed that the operation's stages have no interlocks (the preconditions of all microoperations are identically true) and the branching condition $cond$ is not satisfied. On the first cycle the initial stage $start$ is executed. This stage is responsible for setting the operation's strobe and for assigning the input parameters of the operation to the corresponding inputs of the design. The second cycle is occupied by $stage_1$. At the end of the second cycle the branching condition $cond$ is estimated. Under our assumption, the condition is false. During the third cycle two one-cycle threads $stage_3$ and $stage_4$ are executed in parallel. The last cycle is taken by the final stage end.

To perform simulation-based verification, a testbench should connect specifications and an implementation. For this purpose, one should develop special testbench components: stage drivers, stage monitors, and a mediator. The stage driver is responsible for setting the input signals required by the stage. Usually, input signals (an operation strobe and parameters) are set by initial stages. The driver is executed at the beginning of the cycle on which the stage is carried out. The stage monitor reads the output signals and the internal data affected by the stage and converts them into the specification representation. The information obtained by the monitor is used for checking the stage correctness. The monitor is executed at the end of the cycle occupied by the stage. The mediator reads the output signals and the internal data shared by all operations of the design and converts them into the specification representation. It synchronizes the specification state with the implementation one. The mediator is executed at the end of each cycle of simulation.

D. Organization of Testbench Checkers

Assume that each stage s is specified by a Hoare triple $\{P_s\}C_s\{Q_s\}$ and supplied by a driver D_s and a monitor M_s. In addition, a mediator M is defined. Let S be a set of current stages. To check correctness of the design behavior in response to a certain set of stimuli, a testbench works as follows.

- At the beginning of the cycle:
  - The testbench calculates the set of active stages:
    $$Enabled \leftarrow \{s \in S \mid P_s(\cdot) = true\}.$$
  - Then, it executes the drivers of the active stages in some order. The order is unimportant, because the stages are independent.
- At the end of the cycle:
  - The testbench executes the commands of the active stages.
  - Then, it executes the monitors of the active stages and the mediator.
  - After that, the testbench estimates the design behavior by checking the postconditions of the active stages:
    $$Check(\cdot) = \bigwedge_{s \in Enabled} Q_s(\cdot).$$
  - Finally, it updates the set of current stages:
    $$S \leftarrow \{s \mid s \in S \setminus Enabled\} \cup \bigcup_{s \in Enabled} succ_s(\cdot).$$

Here, $succ_s$ is a function that returns the set of successive stages of the stage s.

The testbench checkers organization described above is a general one. There are two different flavors of the method: *hidden state testing* and *open state testing*.

In the hidden state approach, the testbench's monitors and mediator do not read the internal data of the design. The only information that they care about are the output signals. The testbench uses the stage commands to emulate the design behavior (results of the commands are used to check the design's outputs). Hidden state testing allows to abstract away from the implementation details and thereby to increase the reusability of tests.

In the open state approach, the testbench's monitors and mediator read the internal data of the design to synchronize the specification state with the implementation one on each cycle of simulation. If open state testing is used, the stage commands are not needed, because the testbench is able to obtain their results without emulation. The approach improves the quality of verification by increasing the observability of checkers.

IV. Tool Support

The suggested approach to specification-driven testbench development is supported by the CTESK toolkit developed at ISPRAS [4], [5]. This toolkit is originally intended for testing software systems written in C programming language, but it has been adapted for simulation-based verification of RTL models of hardware designs.

CTESK uses SeC language for development of testbench components. SeC is a C extension, which has additional constructs for description of specifications, drivers, monitors, and other components. The toolkit supports the advanced FSM-based techniques for test sequence generation, but this is out of the paper's scope.

Testbench functionality connected with interpretation of cycle-accurate contract specifications and response checking is implemented as a library extension of CTESK. The library emulates behavior of the design's pipeline by checking the preconditions of the current stages, executing the commands of the active stages, and by checking the postconditions of the active stages.

V. Case Study

The suggested method has been used in several industrial projects on verification of microprocessor units (translation lookaside buffer, cache memory, floating-point unit, arithetic and logic unit, and others). The most complex project is testbench development for the L2 cache of the MIPS64-compatible microprocessor.

The cache under verification is a direct-mapped 256 KB buffer that consists of 8192 rows and serves both data and instructions. It implements operations for loading and storing data, for loading instructions, for modifying control information, and some others (total number of the operations is 6). All operations are multistage and comprises 92 microoperations. Many operations contain branching of control flow and repetitive sequences of microoperations. Some operations can be started in parallel (maximum number of parallel starting operations is 3).

Specifications of microoperations were represented in the form of Hoare triples. It should be emphasized that all requirements were cheaply formalized (total number of the non-trivial preconditions, commands and postcondition is about 120). The volume of specifications is about 3 KLOC in SeC. The labor costs of the testbench development including creation of stimuli generators are 6 man-months. In this project we have found 12 functional errors in the design implementation including very critical ones.

Our experience has shown the following advantages of the approach: (i) thoroughness of response checking (testbench checkers take into account all possible situations in operations execution), (ii) acceptable level of labor costs (testbench development consumes about 20-30% of RTL development efforts), and (iii) ease of debugging (if there is inconsistency between specifications and an implementation is found, a testbench knows which particular microoperation violates the requirements).

VI. Conclusion

The method described in the paper is applicable to a wide rage of synchronous hardware including parallel-pipeline designs with control flow branching and parallel threads inside individual operations. The method combines strengths of assertions and co-simulation while minimizing their weaknesses. Like assertions, it uses predicates for specification of the design behavior, but, in contrast to them, the predicates are not scattered. Instead, they are composed into a full-scale reference model of the design. Such kind of model is not hard to develop, as opposed to conventional co-simulation, because the model's core is separated out as a run-time CTESK library. The only input data that verification engineer should provide to the tool are a description of operations' control flow and contracts of microoperations. The suggested approach has been successfully used in several industrial projects on hardware verification. Further we are planning to develop visual tools for simplifying creation of specifications and tests.

References

[1] J. Bergeron. "Writing testbenches: functional verification of HDL models". Kluwer Academic Publishers, 2000.

[2] W. Lam. "Hardware design verification: simulation and formal method-based approaches". Prentice Hall, 2005.

[3] S. Qadeer, S. Tasiran. "Promising directions in hardware design verification". Proceedings of ISQED, 2002.

[4] http://hardware.ispras.ru

[5] http://www.unitesk.com

[6] C.-M.R. Ho. Validation tools for complex digital designs. PhD thesis, Stanford University, 1996.

[7] H.D. Foster, A.C. Krolnik, D.J. Lacey. "Assertion-based design". Kluwer Academic Publishers, 2004.

[8] C.A.R. Hoare. "An axiomatic basis for computer programming". Communications of the ACM, 12(10):576-580,583, 1969.

[9] A. Kamkin. "Contract specification of pipelined designs: application to testbench automation". Proceedings of SYRCoSE, 2007.