Specification-Driven Testbench Development for Synchronous Parallel-Pipeline Designs

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

***Abstract*— In this paper an approach to testbench development for synchronous parallel-pipeline designs is considered. The approach is based on cycle-accurate formal specifications of a design under verification. Specifications include descriptions of control flow graphs of the design's operations and definitions of the microoperations with the help of Hoare triples. The approach allows to automate testbench development for complex synchronous designs with control flow branching and parallel starting operations. The important feature of the proposed method is that specifications are used to perform all tasks of functional testbenches: checking of design correctness, estimation of test completeness, and generation of test sequences. The approach was successfully used in several industrial projects on hardware verification.**

I. Introduction

Functional verification is a well-known bottleneck in hardware design process. Ensuring the functional correctness of hardware consumes about 70% of the design efforts [1]. The situation is only going to get worse as designs grow in size and complexity. There are two different approaches to checking hardware: formal verification and simulation-based verification [2]. It is known that formal methods are exhaustive but not scalable, while simulation techniques are scalable but not exhaustive [3]. A good balance of exhaustiveness and scalability is provided by semi-formal methods, which combine formal specifications (or models) and simulation.

In this work a semi-formal approach to testbench development is suggested. A testbench is an environment used to verify a design 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 approach described in the paper automates construction of the testbench components on the base of formal specifications.

The rest of the paper is organized as follows. Section II describes the existing approaches to semi-formal verification of hardware designs. In Section III the suggested method is discussed. This section consists of five subsections, which describe the method for specifying synchronous designs (A), organization of adapters between specifications and an implementation (B), and usage of formal specifications of the introduced kind for construction of response checkers (C), coverage trackers (D), and stimuli generators (E). In Section IV the CTESK toolkit is briefly described. Section V is a case study. Section VI concludes the paper.

II. Related Work

There are a lot of research and industrial papers on semi-formal verification methods. This gives evidence that specification-driven testbench development is the promising direction for hardware verification. The main question is what kind of specifications and models are preferable. To automate different tasks of testing, distinct types of models are usually used. For example, stimuli generation can be performed on the base of FSM models, while correctness checking can be done by means of temporal assertions. This has a certain disadvantage. Two models require maintenance during the design process to keep up their mutual consistency.

The most of the papers are dedicated to the methods of test sequence generation. Many of them suggest using explicit cycle-accurate models to generate test sequence, e.g., Ur et al. [4] and Mishra et al. [5] use SMV models; Ho et al. [6] utilize Synchronous Murφ. The main differences between the approaches are concentrated in the following methods: a method of model construction (manual development [4], automatic derivation from an RTL description [6], and automatic derivation from specifications [5]) and a method of test sequence generation (FSM traversal [4], [6] and model checking [5]).

Manual development of a model is error-prone, while automatic derivation from an RTL description does not scale well on complex hardware designs. In our opinion, the most promising method of model construction is automated extraction from formal specifications. Speaking about test generation, model checking techniques are not intended for full-scale functional verification. They are aimed to verification of a relatively small number of properties. The most usable way of test sequence generation is based on FSM traversal.

In the suggested method, a model for test sequence generation, so-called generalized FSM model, is almost automatically derived from specifications. The approach uses implicit specifications in the from of pre- and post-conditions and irredundant algorithms for FSM traversal. The distinction feature of the approach is that it does not require two different models for checking design correctness and for test sequence generation. All testing tasks are carried out basing on formal specifications.

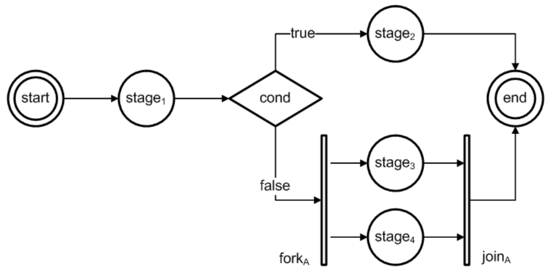

Fig. 1. An example of a control flow graph

III. Suggested Method

Our approach to testbench development 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. Design Specification

State of the design is formalized by a finite set of variables. Among the variables, there are input and output parameters of the operations. All predicates and functions mentioned below are 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 operation's result 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 on the picture consists of nine nodes: one $cond$, one $fork$, one $join$, and six $stages$.

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's contract) $\{P\}C\{Q\}$, where P is a precondition, C is a command, and Q is a postcondition [7]. Semantics of a microoperation precondition is as follows. If the precondition is not satisfied, it does not mean that the microoperation's result 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 it. A channel can handle only one operation per cycle. Sets of operations for different channels are able to have non-empty intersections. 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 $sim : X_1 \times ... \times X_k \rightarrow \{true, false\}$. If the design has several execution channels, one can define a *generalized operation* (*multistimulus*) as 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 $x_1, ..., x_k$.

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 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 the 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).

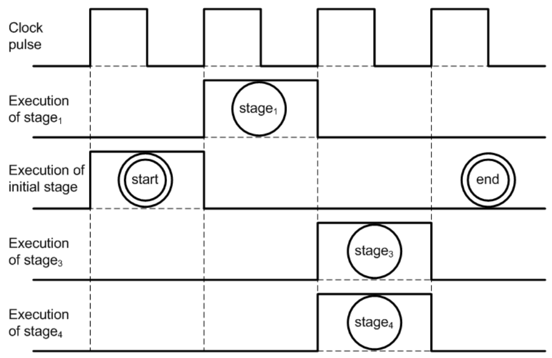

Fig. 2. Cycle-by-cycle execution of the operation

Specifications of the described type are called *cycle-accurate contract specifications*. Originally, they were introduced for linear multi-cycle operations to specify pipelined designs [8].

B. 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 with 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 corresponding cycle. 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.

C. Organization of Response 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, the 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.

D. Test Coverage Description

In the suggested approach to testbench automation, a test adequacy criterion is defined on the base of formal specifications. Test coverage has a hierarchical structure describing different aspects of the design functionality. It consists of three levels of granularity.

Microoperation-level coverage defines test situations on microoperation interlocks (*interlock coverage*) and on functional branches inside individual microoperations (*functional coverage*). In the simplest case, interlock coverage includes two situations, which describe whether the stage is interlocked or not. General-case coverage takes into account a structure of the logical formula that describes the interlock. A typical microoperation has no local branching; therefore its functional coverage consists of only one element.

Operation-level coverage specifies test situations on an operation in terms of paths in the operation's control flow graph. Usually, a control flow graph is acyclic. In this case, it is reasonable to define test situations as paths from the initial nodes to the final ones.

Pipeline-level coverage describes test situations connected with parallel calls of operations (*multistimulus coverage*) and simultaneous execution of microoperations (*control state coverage*). The goal of test generation is to create all feasible combinations of simultaneously executing microoperations. To this effect, one needs to use all possible paths in operations and all possible interlock situations.

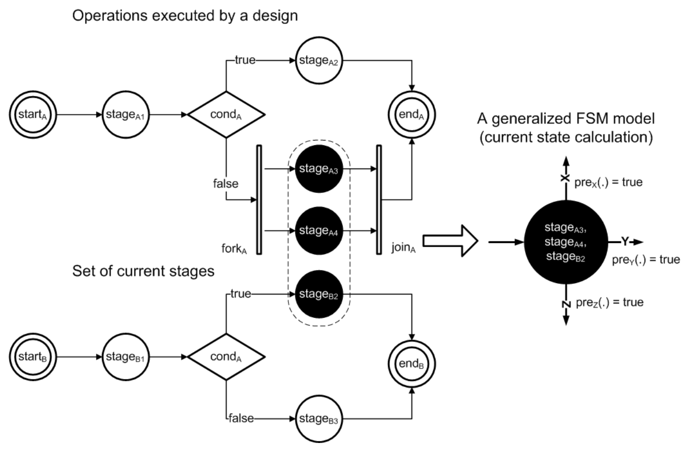

Fig. 3. Construction of an generalized FSM model

E. Test Sequence Generation

To create test sequencies for a design under verification, we use a generalized FSM model of the design. Roughly speaking, a state of genereralized model is idenfied by a set of current stages (see Fig. 3). The state graph of the generalized model is created on the fly. Test generation is complete when all reachable states (all feasible combinations of simultaneously executing microoperations) are visited.

In order to make the model deterministic, a verification engineer should include some additional information into the FSM state. For example, if there are interlocks in the microoperations, one should add a certain information that deterministically determines the interlock conditions; if there are branches in the operations, one should add some information that determines the branching conditions, and so on. This part of the approach is not fully automated.

To compile several operations from different execution channels into a multistimulus, we have a temporary buffer [9]. Every operation except nop is put into the buffer before it is applied to the design. Multiple operations targeted at the same channel are prohibited to be added. When nop is applied, all operations stored in the buffer are applied to the design in parallel, and the buffer is flushed.

IV. Tool Support

The suggested approach to specification-driven testbench development is supported by the CTESK toolkit developed at ISPRAS [10]. This toolkit is originally intended for testing software systems written in C programming language, but it has been adapted for 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. Testbench functionality connected with interpretation of cycle-accurate contract specifications and response checking is implemented as a library extension of CTESK.

V. Case Study

The suggested method and the CTESK toolkit have been used in several industrial projects on hardware verification. The most complex project is the testbench development for L2 cache of the MIPS64-compatible microprocessor.

The cache under verification is a direct-mapped 256 KB cache 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 pipelined operations (total number of the microoperations is 92). Many of them contain branching of control flow. Some operations can be started in parallel (maximum number of parallel starting operations is 3).

Specifications of the microoperations were represented in the form of Hoare triples. It should be emphasized that all requirements were cheaply formalized (total number of the nontrivial preconditions, commands and postcondition is 120). The volume of specifications is about 3 KLOC in SeC. The labor costs of the testbench development are approximately 6 man-months (20-30% of RTL development efforts). In this project we have found 12 functional errors in the design implementation including very critical ones.

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 usage of formal specifications allows to automate the main tasks of simulation-based verification: test sequence generation, checking of design correctness, and estimation of test completeness. The distinction feature of the approach is that the same specifications are used for all tasks of testing. This simplifies maintenance of testbenches and reduces verification efforts. The suggested approach has been successfully used in real-life projects on hardware verification.

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". Proc. of International Symposium on Quality Electronic Design, 2002.
[4] S. Ur, Y. Yadin. "Micro architecture coverage directed generation of test programs". Proc. of Design Automation Conference, 1999.
[5] P. Mishra, N. Dutt. "Functional coverage driven test generation for validation of pipelined processors". Proc. of Design, Automation and Test in Europe, 2005.
[6] R. Ho, C. Yang, M. Horowitz, D. Dill. "Architecture validation for processors". Proc. of International Symposium on Computer Architecture, 1995.
[7] C.A.R. Hoare. "An axiomatic basis for computer programming". Communications of the ACM, 12(10):576-580,583, 1969.
[8] A. Kamkin. "Contract specification of pipelined designs: application to testbench automation". Proc. of Spring Young Researchers' Colloquium on Software Engineering, 2007.
[9] M. Chupilko. "Constructing test sequences for hardware designs with parallel starting operations using implicit FSM models". Proc. of East-West Design & Test Symposium, 2009
[10] http://hardware.ispras.ru