Contract Specification of Hardware Designs at Different Abstraction Levels: Application to Functional Verification

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

***Abstract*— The paper touches upon the issues of functional specification and verification of digital hardware at different abstraction levels. It shows how behavioral models of various degrees of abstraction can be represented by means of the contract paradigm and how contract specifications can be applied to generate test sequences in an automated way. The testing technique under consideration is based on the traversal of FSM derived from specifications. Taking into account that contract specifications are well known to be a high-efficient tool for constructing response checkers and estimating test coverage, we can assuredly report that the contract-based approach is a universal solution for hardware verification at different levels of abstraction.**

I. Introduction

Automation of hardware design verification is feasible only if requirements to hardware are represented in a formal way. This refers not only to formal methods where using mathematical models is one of the integral parts but to simulation-based methods as well. Requirements being represented formally, i.e., in a machine-readable form, are called *formal specifications*. Basing on such specifications one can automatically derive a design model (usually, some kind of automaton) and apply it to functional verification (e.g., to generate stimuli, check reactions, or estimate test adequacy).

When verifying hardware of different size and complexity it is reasonable to use specifications and models of different levels of abstraction. For example, a hardware unit can be specified cycle-accurately, while description of a whole system can be done in a more generalized manner. Choice of an abstraction level depends on the requirements to be specified (they are not always very accurate) and in many cases is conditioned by reusability and maintenance reasons – more accurate specifications are, less reusable they are, and more labor-consuming their maintenance is.

Nowadays, there are lots of specification-based techniques for functional verification of hardware, but there is a lack of unified solutions, which are applicable to various abstraction levels, e.g., contemporary unit and system verification methods are completely different. This paper classifies the modeling levels used in hardware verification and considers how diverse models can be uniformly described with the help of *contract specifications* (i.e., *pre-* and *post-conditions*) and how such specifications can be used to automate simulation-based verification (test sequence generation, in particular).

This work was supported by the RFBR (grant 08-01-00889-).

The testing technique concerned is based on the traversal of FSM constructed from contract specifications. It is obvious that specifications of different size and accuracy produce different automata in terms of determinacy and number of transitions. It is not always possible to make use the derived FSM model due to the huge size or for some other reason. To make the model useable for test generation, one should generalize it. Doing this involves many factors and mostly remains the art of verification. Some of the approaches to construct automata from specifications are described in the paper.

The rest of the article is organized as follows. Section II reviews the papers devoted to specification-based verification of hardware designs. In Section III classification of the abstraction levels used for hardware modeling is given. This section also describes how various models can be represented by means of contract specifications. In Section IV construction of FSM from contract specifications (to be afterwards used for test sequence generation) is considered. Section V gives a concise description of tool support. Finally, Section VI concludes the paper and outlines the directions of our future work.

II. Related Work

There are lots of research and industrial papers on specification-based verification methods. This gives evidence that using formal specifications and models is a right 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. [1] and Mishra et al. [2], [3], [4] use SMV models; Ho

et al. [5] utilize Synchronous Murφ. The main differences between the approaches are concentrated in the following methods: a model construction method (manual development [1], automatic derivation from an RTL description [5], and automatic derivation from specifications [2]) and a test sequence generation method (FSM traversal [1], [5] and model checking [2], [3], [4]).

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 approach, 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 contract specifications.

III. Specification of Hardware Designs at Different Abstraction Levels

When specifying hardware designs we mostly focus on their behavior (functionality). In other words, hardware is considered to be a black box with the given inputs and outputs, and our goal is to specify the input-output relation of the DUV (Design Under Verification).

It is widely recognized that EFSM (Extended Finite State Machine) is one of the most natural formalisms for hardware modeling [6]. It would be recalled that, roughly speaking, EFSM is an automaton with parameterized inputs (stimuli) and outputs (reactions) where state is divided into control and data parts.

In this paper, it is implied that a functional model of hardware is an EFSM-like automaton (formal definitions are not given not to overload the paper). We distinguish the following abstraction levels for hardware modeling (in order of increasing abstractness):

- cycle-driven models:
  - cycle-accurate models;
  - adaptive cycle-driven models;
- event-driven models:
  - ordered events models;
  - unordered events models;
- operation-driven models.

Let us describe the basic properties of hardware models at each of the abstraction levels given above.

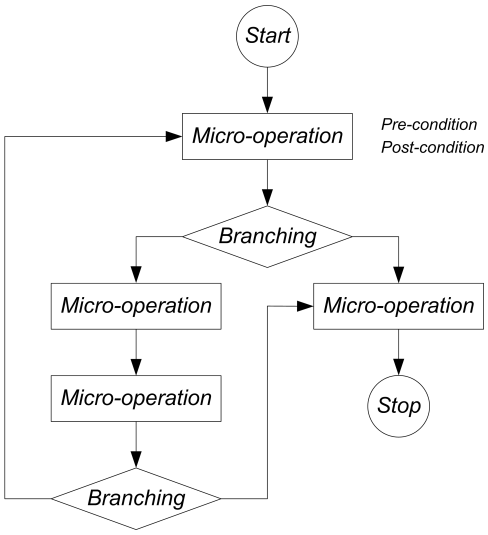

Fig. 1. A control flow graph of an operation.

A. Cycle-Driven Models

Cycle-driven models are the least abstract and most detailed ones. A model is called *cycle-driven* if each of its transitions corresponds to exactly one clock cycle in hardware[1]. To formally describe a cycle-driven model by means of contract specifications, the operations implemented by a DUV should be decomposed into a number of one-cycle *micro-operations* each being described by an individual *contract* (i.e., a *pre-condition* and *post-condition*). It should be noticed that micro-operations of an operation can be connected to each other not only by linear ordering but using more complex control flow relation as well (see Fig. 1, for example).

Summing up, cycle-driven specification of an operation includes its pre-condition (which constrains the situations in which the operation is allowed to start), a set of interconnected micro-operations, and pre- and post-conditions of the micro-operations. Semantics of a micro-operation's pre-condition (which is also called a *guard condition*) differs from the semantics defined for an operation's pre-condition – if a guard condition is not satisfied, it indicates that the micro-operation is interlocked (it will be unlocked, when the guard condition becomes true).

Cycle-driven models are subdivided into two types: cycle-accurate models and adaptive cycle-driven models.

1) ***Cycle-Accurate Models****:* A cycle-driven model is referred to as *cycle-accurate* if for any admissible input sequence it allows *deterministically* identifying (predicting) a DUV's reaction. In other words, it is a self-contained description of the design cycle-by-cycle execution, which can be used *independently* from the implementation (that is the difference between accurate and adaptive models).

In a sense cycle-accurate models are ideal for *co-simulation*. In each cycle a testbench applies an input both to a DUV and

[1]It does not make any serious difference whether a transition or staying in a state corresponds to a clock cycle.

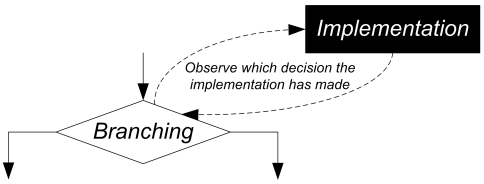

Fig. 2. Using a feedback for choosing a branch.

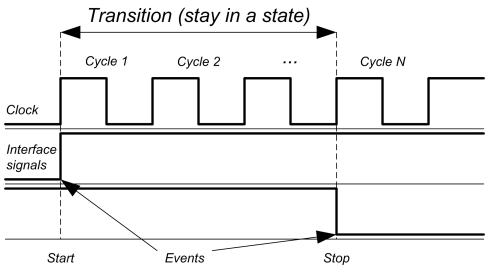

Fig. 3. A transition in an event-driven model.

to a model and compares their outputs for equivalence. There is no problem in determining when to apply a stimulus or when to check a reaction – these actions are performed in each cycle of simulation. However, making a cycle-accurate model is a difficult task which is almost tantamount to writing one more implementation.

2) ***Adaptive Cycle-Driven Models****: Adaptive cycle-driven models* are not as self-contained and deterministic as cycle-accurate models are. Sometimes they are not able to determine a DUV's reaction basing only on the input sequence, but it can always decide which transition to perform observing some outputs of the design. In other words, there is a *feedback* from a DUV to a model (that is why such models are called *adaptive*).

Due to the feedback adaptive models can not be used independently from the implementation. Speaking in the terms of contract specifications, micro-operations' pre-conditions and branching conditions can be defined not only over the model variables (control and data state) but over the design's outputs as well (see Fig. 2). The post-conditions do not check the values of the feedback outputs (DUV's outputs the model depends on).

Adaptive models are thought to be a bit more abstract than accurate ones, because they abstract away the way in which feedback outputs are calculated. When using such kind of models there is a tacit assumption that feedback outputs are computed correctly.

B. Event-Driven Models

Event-driven models are the next step in increasing abstractness of the hardware functionality description. The key distinction of that sort of models is that there is no rigid connection between model transitions and clock cycles. Different transitions (or stays in states) can take different amount of time to be completed. Moreover, the same transition being executed several times can take different number of cycles for each execution.

The main concept of event-driven models is an *event*, which is an instantaneous interaction between a design and its environment that can be observed or predicted. If we are talking about verification, an event is an atomic interaction between a DUV and a testbench (e.g., sending of a stimulus, assignment of an output, etc.). A model is called *event-driven* if its transitions are associated with events, not cycles (see Fig. 3).

To specify an event-driven model by means of contract specifications, each operation is decomposed into a number of *sub-operations* (interactions). Further thoughts are very similar to those that are applied to cycle-driven models – all sub-operations are connected into a control flow graph, while each sub-operation is specified by pre-and post-conditions.

Event-driven models are subdivided into two types: ordered events models and unordered events models.

1) ***Ordered Events Models****:* An *ordered events model* is an event-driven model where each transition corresponds to a set of simultaneous events or a linearly ordered set of events happening at different times. It should be noticed that cycle-driven models are a particular case of ordered events models. Indeed, a clock cycle can be considered as a kind of event.

Event-driven models (including models of ordered events) are usually adaptive, because in general case their behavior depends not only on input events (stimuli) but on output events (reactions) as well. To be able to use event-driven models for simulation-based verification, one should develop special testbench components that detect a DUV's reactions and transmit them into a model. Such components are called *catchers*.

2) ***Unordered Events Models****:* In contrast to a model of ordered events, each transition of an *unordered events model* is associated with an unordered set of events. In other words, using such kind of models we know which events have taken place during a transition, but we do not know the linear order of the events (because, for example, some events are not visible or communication medium can change the order).

The distinctive feature of unordered events model is how reactions are checked. Since the events order is not fully known, a testbench tries to create all admissible orders and check the events' post-conditions. This process is called *serialization*. If there is at least one sequence of events such that all the post-conditions are satisfied, the overall reaction is considered to be correct.

C. Operation-Driven Models

No research on hardware verification would be complete that did not consider *operation-driven models*. Such models are the most abstract and described by a vacuous single-state automaton in which each transition corresponds to an operation call. Operation-driven models abstract away from operations' structure (micro- and sub-operations) and cooperative execution of several operations.

Specification of an operation is a classical software contract consisting of a pre-condition (which constrains the situations

in which the operation is allowed to start) and post-condition (which constrains the expected results of the operation).

Operation-driven models are often applied to core-level verification of microprocessors being done with the help of test programs. The same programs are executed on a microprocessor RTL design and its instruction-level simulator (operation-driven model). The results of the two models are compared for equivalence.

D. Unified View on Hardware Specification

It is important that models of various degree of abstraction can be specified in a unified way by means of contract specifications. Contract specifications abstraction/refinement is the topic of a separate research, but we can report that changing abstraction level can be done almost seamlessly. To illustrate the idea, let us consider a simple example. In the example, a operation-driven specification is refined into cycle-driven one by adding timing information without changing the structure of the specification.

The code below (we use the language SeC, specification extension of C) describes the functionality of an address translation operation being implemented by a hypothetical TLB (Translation Lookaside Buffer): if a virtual address is invalid, then the output `error` is set to one; if the address is valid but does not belong to the buffer, then the output `miss` is set to one; if the address is valid and it belongs to the buffer, then the output `pa` is set to the resultant physical address.

```
specification void AddressTranslation(VA va) {
  TLB *tlb = getDUV();
  // Pre-condition of the operation
  pre {
    return true;
  }
  // Post-condition of the operation
  post {
    TLBEntry *entry;
    POST(tlb.out.error == !isValid(va));
    if(!isValid(va)) { STOP(); }
    POST(tlb.out.miss == !isHit(tlb, va));
    if(!isHit(tlb, va)) { STOP(); }
    POST(tlb.out.pa == translate(tlb, va));
    STOP();
  }
}
```

To simplify description, the specification uses two special macroses, `POST` and `STOP`. The first of them checks the predicate given and return $false$ if the predicate is not satisfied. The second macros simply returns $true$.

The code below refines the given specification by adding some timing information. This is done with the help of the constructs `CYCLE` and `PRE`. Calling `CYCLE` leads to a one-cycle delay. `PRE` waits until the condition given is satisfied.

```
specification void AddressTranslation(VA va) {
  TLB *tlb = getDUV();
  pre {
    return true;
  }
  post {
    TLBEntry *entry;
    // Post-condition of the micro-operation 1
    POST(tlb.out.error == !isValid(va));
    if(!isValid(va)) { STOP(); }
    // One-cycle delay
    CYCLE();
    // Post-condition of the micro-operation 2
    POST(tlb.out.miss == !isHit(tlb, va));
    if(!isHit(tlb, va)) { STOP(); }
    // Pre-condition of the micro-operation 3
    PRE(tlb.out.ready == 1);
    // Post-condition of the micro-operation 3
    POST(tlb.out.pa == translate(tlb, va));
    STOP();
  }
}
```

IV. Test Sequence Generation at Different Abstraction Levels

The test sequence generation technique we use for verification is based on the traversal of a control FSM derived from specifications. The distinctive feature of the approach is that it does not require explicit FSM representation – instead, the only information it operates with is a traversed part of a state graph, current state, and set of available stimuli. Using such a technique implies that FSM being traversed is deterministic (in some sense) and has a strongly connected state graph.

Specifications of different abstractness lead to different automata in terms of determinacy and number of states. Obviously, it is not always possible to automatically derive a deterministic FSM with a rather small number of states – sometimes a resultant automaton is nondeterministic; other times it is of a huge size.

Thus, one should choose a right abstraction level and given the abstraction level, construct a deterministic automaton (trying to avoid redundant states, of course). This task is not fully automated, but there are several empiric ideas (testing patterns) that we would like to consider. These ideas basically relate to the current state calculation function of the implicit FSM representation.

A. Cycle-Driven Models

*1) **Stage-Based FSM**:* The most usable pattern of FSM description for cycle-driven specifications is as follows. The control state is represented as a set of simultaneously executing micro-operations (see Fig. 4). This approach is called a *micro-operation-based FSM* or a *stage-based FSM* (*stage* is a generalization of micro-operations and sub-operations).

If there are inter-operation *data dependencies* which have an influence on pipeline interlocks (there are nontrivial micro-

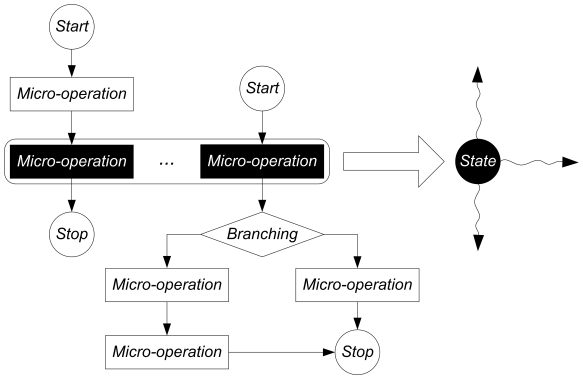

Fig. 4. Constructing a micro-operation-based (stage-based) FSM.

operations' pre-conditions), the pure stage-based approach does not work, because the resultant FSM is most likely nondeterministic. In this case, an extended approach should be used – an automaton state is not only a set of micro-operations, but it also contains some control information describing dependencies between operations. It is also sensible to use *timers* (to make states describe not only a micro-operations but time of their processing as well). More detailed description of the approach is available in [8].

2) ***Resource-Based FSM****:* The other approach is called a *resource-based FSM*. An automaton state represents not currently executing micro-operations but the resources allocated by the micro-operations. This testing pattern abstracts away from the particular operations processing by a design, but it often requires additional work to make an FSM deterministic.

We distinguish some types of resources which hardware designs usually contain: arbiters, FIFO-buffers, RAM-memories, and data transfer channels. Each resource is specified with an individual FSM, and each FSM is described implicitly by defining a way for the resources state calculation. There are some patterns for specifying states. For example, state of an arbiter is often represented as a pair $\langle requests, history \rangle$, where $requests$ are queries having been sent to the arbiter during the previous cycle, and $history$ is a finite sequence of the recent arbiters decisions (it is used to calculate priorities of the requests); state of a FIFO is usually described as a number of entries in the buffer; and so on. Having got known what kind of resources a DUV consists of, one is able to construct a description of a total FSM by concatenating all state calculation functions.

B. Event-Driven Models

The stage- and resource-based approaches can be also applied to event-driven specifications. In this case, sub-operations (instead of micro-operations) are considered. In the stage-based approach, an FSM state is a set of processing sub-operations. In the resource-based approach, an automaton state is a set of the resources allocated by the sub-operations.

C. Operation-Driven Models

Operation-driven specifications are not able to produce an adequate control FSM due to the low informativeness (high abstractness). In this case, combinatorial techniques are used for test generation. The test sequences are generated by systematic enumeration of all feasible combinations of the given operations, test situations (i.e., paths in control flow graphs) and dependencies via shared resources. To reduce number of tests, one can use heuristics, like operation factorization, limitation of the number of dependencies, etc. [7]

V. Tool Support

The suggested approach to specification and test sequence generation is supported by the CTESK toolkit developed at the Institute for System Programming of the Russian Academy of Sciences (ISPRAS) [9]. This toolkit is originally intended for testing software systems written in C, but it has been adapted for verification of hardware designs.

CTESK uses SeC language for development of testbench components. SeC is a C extension, which has additional constructs to define specifications, FSM-based test scenarios, etc. Testbench functionality connected with functional verification of hardware designs is implemented as a library extension of CTESK.

VI. Conclusion

Contract specifications are applicable to a wide rage of hardware including complex parallel-pipeline designs with control flow branching and parallel threads inside individual operations [8]. Their usage allows automating all tasks of simulation-based verification. This simplifies maintenance of functional tests and reduces verification efforts. The important quality of contract specifications is that they can be easily applied to functional verification of hardware at different abstraction levels. Futher we are planning to do a research on smooth changing of specification abstractness. Choosing right abstraction level for verification is a problem which is really hard to formalize. We believe that a tool that could change the abstractness without changing the structure of specifications would be rather useful for verification engineers.

References

[1] S. Ur, Y. Yadin. "Micro architecture coverage directed generation of test programs". Proc. of Design Automation Conference, 1999.
[2] P. Mishra, N. Dutt. "Functional coverage driven test generation for validation of pipelined processors". Proc. of Design, Automation and Test in Europe, 2005.
[3] H.M. Koo, P. Mishra. "Test generation using SAT-based bounded model checking for validation of pipelined processors". Proc. of ACM Great Lakes Symposium on VLSI, 2006.
[4] H.M. Koo, P. Mishra. "Functional test generation using property decomposition for validation of pipelined processors". Proc. of Design, Automation and Test in Europe, 2006.
[5] R. Ho, C. Yang, M. Horowitz, D. Dill. "Architecture validation for processors". Proc. of International Symposium on Computer Architecture, 1995.
[6] A. Petrenko, S. Boroday, R. Groz. "Confirming Configurations in EFSM Testing." IEEE Transactions on Software Engineering, 2004.
[7] A. Kamkin. "Combinatorial model-based test program generation for microprocessors". Preprint of ISPRAS, 2009.
[8] M. Chupilko, A. Kamkin. "Specification-driven testbench development for synchronous parallel-pipeline designs". Proc. of the NORCHIP conference, 2009.
[9] http://hardware.ispras.ru