Promising Directions in Hardware Design Verification

Shaz Qadeer Serdar Tasiran
Compaq Systems Research Center
Palo Alto, CA 94301

Abstract

Ensuring the functional correctness of hardware early in the design cycle is crucial for both economic and methodological reasons. However, current verification techniques are inadequate for industrial designs. Formal verification techniques are exhaustive but do not scale; partial verification techniques based on simulation scale well but are not exhaustive. This paper discusses promising approaches for improving the scalability of formal verification and comprehensiveness of partial verification.

1 Introduction

Functional validation of hardware designs is an important but extremely difficult problem. The two most common approaches to this problem are based on simulation and formal verification. Currently, neither approach is able to handle industrial designs. Simulation can be performed on large designs but it exercises only a small fraction of the possible input sequences and can miss errors. Formal verification is exhaustive but does not scale. As a result, research in this field has focused on increasing both the comprehensiveness of simulation and the capacity of formal verification.

The most widely used formal verification method is model checking [1, 2] where properties of a design are checked by state space enumeration. Current model checkers are limited to hardware designs with a few hundred signals. We present assume-guarantee reasoning [3, 4, 5, 6], a semi-automatic technique for addressing this limitation. In this style of reasoning, a model checking task on a large module is decomposed into a set of model checking tasks on smaller modules. Several large hardware designs [7, 8, 9] have been successfully verified using this approach.

Assume-guarantee reasoning is not completely automatic because it requires the user to provide an abstraction of the implementation and definitions of implementation signals in terms of the abstraction. Hence, this verification method is naturally useful in a design methodology of top-down iterative design refinement. The complexity of modern designs, the need for design reuse, and the prohibitive cost of functional validation has started to push the design process in this direction. Therefore, we believe that assume-guarantee style reasoning will find wide application in the future of hardware design.

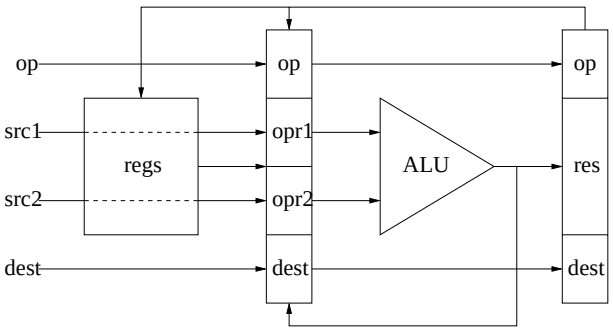
In order to apply assume-guarantee reasoning, specifications and designs need to be expressed formally, with precisely defined interfaces between modules implemented by different designers. Furthermore, this modular structure needs to be maintained during the evolution of the design. This implies a radical shift from the typical design practices of today. Consequently, software simulation continues to be the prevalent means for functional verification. However, simulation driven by random and hand-written inputs has become increasingly unsatisfactory. The resulting validation is not exhaustive enough and gaps in validation are hard and costly to pin-point and improve. In the second half of the paper, we present an approach for improving the quality of validation obtained from software simulation. The techniques discussed are based on coverage analysis, and automatic generation of simulation inputs guided by this analysis.

2 Hardware modules

In this section, we introduce a hardware design that is used to illustrate the techniques discussed in the paper. Figure 1 shows a simple three-stage pipeline. The module `PIPELINE` has a set of input and output signals of finite types. `PIPELINE` has four inputs: the signals `op`, `src1`, `src2`, and `dest` denoting the type, the two source registers, and the destination register of the operation to be performed. The operation `op` is either `AND` or `NOP`. There is an ALU to perform the operation `AND`. The pipeline contains a register file `regs`, and two sets of pipeline latches `pipe1` and `pipe2`. The pipeline latches are records: `pipe1` has fields `op`, `opr1`, `opr2`, and `dest`; `pipe2` has fields `op`, `dest`, and `res`.

The output signals are initialized by the sequence of assignments denoted by **init**. The next values of output signals are obtained by executing the sequence of assignments denoted by **next**. At each step, the next value of an output signal can depend on the values of signals in both the current and the next state. It is illegal for two signals to each depend on the next value of the other signal. In the assignments, an unprimed signal refers to its value in the current state and a primed signal refers to its value in the next state.

A *state* of a module M is an assignment to its input and output signals. The set $State_M$ of all states of the module M is exponential in the number of signals. A hardware module M has a set of *initial* states $Init_M \subseteq State_M$ and a *transition relation* $Next_M \subseteq State_M \times State_M$. A state $s \in Init_M$ iff s can be obtained by executing the **init** assignments. The pair $\langle s, t \rangle \in Next_M$ iff t can be




```
module PIPELINE
input op: opType; src1, src2, dest: regIndexType
output regs: regFileType;
       pipe1: pipe1Type; pipe2: pipe2Type
init
  pipe1.op′ := NOP
  pipe2.op′ := NOP
  forall i do regs′[i] := 0
next
  pipe1.op′ := op′
  pipe1.opr1′ :=
     if src1′ = pipe1.dest & pipe1.op = AND
        then pipe2.res′
     elsif src1′ = pipe2.dest & pipe2.op = AND
        then pipe2.res
     else regs[src1′]
  pipe1.opr2′ :=
     if src2′ = pipe1.dest & pipe1.op = AND
        then pipe2.res′
     elsif src2′ = pipe2.dest & pipe2.op = AND
        then pipe2.res
     else regs[src2′]
  pipe1.dest′ := dest′
  pipe2.op′ := pipe1.op
  pipe2.dest′ := pipe1.dest
  pipe2.res′ := pipe1.opr1 & pipe1.opr2
  regs′[pipe2.dest ] :=
     if pipe2.op = AND then pipe2.res
     else regs[pipe2.dest ]
```

Figure 1: Three-stage pipeline

obtained by executing the **next** assignments from s. A state s of M is *reachable* if there is a sequence $s_0, \ldots, s_n$ of states such that $s_0 \in \mathit{Init}_M$ and $s_n = s$ and for all $0 \leq i < n$ we have $\langle s_i, s_{i+1} \rangle \in \mathit{Next}_M$. The set Reach_M is the set of all reachable states of M.

Modules M_1 and M_2 (with different outputs) can be composed by connecting the inputs of M_1 that are outputs of M_2 and the outputs of M_1 that are inputs of M_2. The composition of M_1 and M_2 is denoted by $M_1 \| M_2$.

3 Formal verification

A hardware module is formally verified by stating a property on the design and then checking that the design satisfies the property. The most commonly specified property is an invariant, which expresses a condition on the hardware module that should never happen in a reachable state (or conversely, a condition that should always be true in a reachable state). Formally, an *invariant* is a boolean formula over the signals of the module. The module M satisfies the invariant I if every reachable state of the module satisfies I. Thus, invariant verification on a module can be performed by computing the set of its reachable states. However, this computation is difficult in general since the set of reachable states can be exponential in the number of signals in the module. This exponential growth in the number of states is known as the *state explosion problem*.

An approach to mitigating the state explosion problem is symbolic model checking [10, 11]. Symbolic model checking represents sets of states and the transition relation of a hardware module as a boolean function. Boolean functions are commonly represented as binary decision diagrams (BDDs) [12]. The BDD representation of both Init_M and Next_M is obtained by a syntax-directed translation from a description like that in Figure 1. The BDD representation of I can be computed similarly.

During reachability computation, the set R_n of states reachable in n steps from an initial state is computed iteratively. The set R_0 is just Init_M; the BDD for R_{n+1} is obtained by performing *image computation* using the BDD representations of R_n and Next_M. The set Reach_M is the fixpoint of this computation. Finally, the validity of $\mathit{Reach}_M \Rightarrow I$ (checkable by a BDD operation) ensures that M is satisfied by I. For a number of hardware designs, the BDD representation of the set of reachable states is manageable even though the number of reachable states itself is very large. For such designs, although explicit enumeration of reachable states is infeasible, it is still possible to check invariants using symbolic model checking.

Even with symbolic model checking, invariant verification is currently limited to modules with a maximum of a few hundred signals. To overcome this limitation, another approach is to verify the invariant I on an abstract module N (with smaller number of signals) and show that M is a refinement of N. Then, the module M also satisfies I. A module M *refines* a module N, denoted by $M \prec N$, if every output of N is an output of M, every input of N is either an input or output of M, and every transition taken by M can also be taken by N. The last requirement can be easily checked by symbolic model checking by first computing Reach_M and then checking the validity of

$$\mathit{Reach}_M \wedge \mathit{Next}_M \Rightarrow \mathit{Next}_N.$$

It can be shown easily that if $M \prec N$ and N satisfies an invariant I, then M also satisfies I.

The refinement check $M \prec N$ still requires the computation of the set Reach_M of the reachable states of M. If done naively, this is no better than checking the invariant I on M directly. However, assume-guarantee reasoning allows the decomposition of refinement checking tasks into subtasks on modules with smaller number of signals. In Sections 3.1 and 3.2, we illustrate the use of refinement checking and assume-guarantee reasoning on the three-stage pipeline module described in Section 2.

```
module ISA
input op: opType; src1, src2, dest: regIndexType
output isaRegs: regFileType
init
  forall i do isaRegs′[i] := 0
next
  isaRegs′[dest′] :=
     if op′ = AND then isaRegs[src1′] & isaRegs[src2′]
     else isaRegs[dest′]
```

Figure 2: Instruction set architecture

3.1 Pipeline verification

Let us consider the pipeline in Figure 1. What does it mean for the pipeline to be functionally correct? One approach is to check various properties about the behavior of the pipeline. For example, we could assert that for any time t, if the pipeline is empty (pipe1.op = pipe2.op = NOP) and the input instruction is $R0 \leftarrow \mathtt{AND}(R1, R2)$, then the contents of register 0 at time $t+3$ is the conjunction of the contents of registers 1 and 2 at time t. While this property is useful, it is difficult to come up with a comprehensive set of properties that provide a full specification of the pipeline.

A different way to specify PIPELINE is to write the instruction set architecture for it. The instruction set architecture for PIPELINE is the module ISA shown in Figure 2. ISA has the same set of inputs as PIPELINE but its implementation is simpler. ISA executes each instruction and updates its register file isaRegs in one cycle, whereas PIPELINE takes three cycles to process an instruction. The conformance between PIPELINE and ISA can be expressed as the requirement that if the pipeline is empty then the contents of the register files in PIPELINE and ISA are identical. This requirement can be formalized as the invariant

$$I \stackrel{\text{def}}{=} \mathtt{pipe1.op} = \mathtt{NOP} \wedge \mathtt{pipe2.op} = \mathtt{NOP} \Rightarrow \mathtt{regs} = \mathtt{isaRegs}$$

The invariant I depends on pipe1.op, pipe2.op, regs, and isaRegs. The signal regs depends on pipe2, which depends on pipe1, which again depends on pipe2 and regs. Thus, the presence of these circular dependencies in PIPELINE ensure that every signal is relevant for the verification of I. Commonly used heuristics for reducing the number of signals relevant to a property like *cone-of-influence* reduction is ineffective in this situation. Although we can check the invariant I on the composition of PIPELINE and ISA, this approach will not scale to modules with a large number of signals. In the next section, we present a method to decompose the verification of I into a set of verification subtasks each of which involves a module with a small subset of the signals in PIPELINE and ISA. The decomposition method works even in the presence of circular dependencies.

3.2 Assume-guarantee reasoning

We would like to construct an abstraction of the composition of PIPELINE and ISA such that it is simpler to verify I on the abstraction. Then we would like to use assume-guarantee reasoning to check that the abstraction is correct. The key insight in coming up with the abstraction is that the computation being performed in the various stages of the pipeline has already been performed in the instruction set architecture. For example, the value of pipe2.res is computed from pipe1.opr1 and pipe1.opr2 in PIPELINE. But, that value is already present in the register pipe2.dest of isaRegs. Similarly, the values of pipe1.opr1 and pipe1.opr2 are present in the registers src1 and src2 respectively of isaRegs. Thus, we have the following abstraction module ABS containing the abstract definitions of pipe2.res, pipe1.opr1 and pipe1.opr2 in terms of isaRegs:

```
module ABS
input pipe1.op, src1, src2, pipe2.op, pipe2.dest
output pipe1.opr1, pipe1.opr2, pipe2.res
  pipe1.opr1′ :=
     if pipe1.op′ = AND then isaRegs[src1′]
     else {false,true}
  pipe1.opr2′ :=
     if pipe1.op′ = AND then isaRegs[src2′]
     else {false,true}
  pipe2.res′ :=
     if pipe2.op′ = AND then isaRegs[pipe2.dest′]
     else {false,true}
```

The definition of pipe2.res says, for example, that the value of pipe2.res is the same as register pipe2.dest in isaRegs if there is a valid operation in the second stage of the pipeline. Otherwise the value of pipe2.res is arbitrary.

Given a module M and a subset P of its output signals, the P-slice of M is the module whose output signals are in P and which is obtained by taking the subset of **init** and **next** statements that assign values to signals in P. Thus, if P_1 and P_2 is a partitioning of the output signals of M, then the composition of the P_1-slice of M and the P_2-slice of M is equal to M. Let A, B, and C be the {pipe1.opr1}-slice, the {pipe1.opr2}-slice, and the {pipe2.res}-slice of PIPELINE‖ISA respectively. Then PIPELINE‖ISA can be alternatively represented by $A\|B\|C\|E$, where E is the module with output signals other than pipe1.opr1, pipe1.opr2, and pipe2.res. Let $\overline{A}$, $\overline{B}$, and $\overline{C}$ be the {pipe1.opr1}-slice, the {pipe1.opr2}-slice, and the {pipe2.res}-slice of ABS respectively.

Now we can prove the invariant I in two steps:

$$\frac{\begin{array}{lll} A\|B\|C\|E & \prec & \overline{A}\|\overline{B}\|\overline{C}\|E \\ \overline{A}\|\overline{B}\|\overline{C}\|E & \text{satisfies} & I \end{array}}{A\|B\|C\|E \quad \text{satisfies} \quad I}$$

Finally, assume-guarantee reasoning allows us to decompose the refinement check as follows:

$$\frac{\begin{array}{lll} A\|\overline{B}\|\overline{C}\|E & \prec & \overline{A} \\ \overline{A}\|B\|\overline{C}\|E & \prec & \overline{B} \\ \overline{A}\|\overline{B}\|C\|E & \prec & \overline{C} \end{array}}{A\|B\|C\|E \quad \prec \quad \overline{A}\|\overline{B}\|\overline{C}\|E}$$

Note that the circular dependencies between the pipeline signals is reflected in the three refinement verification lemmas. The abstraction of each of `pipe1.opr1`, `pipe1.opr2` and `pipe2.res` is verified using the abstractions of the other signals. The correctness of the assume-guarantee decomposition is ensured by two requirements. First, two signals in a module cannot have zero-delay dependency on each other. Second, there are no two signals x and y such that x has a zero-delay dependency on y in the module $A\|B\|C\|E$, and y has a zero-delay dependency on x in the module $\overline{A}\|\overline{B}\|\overline{C}\|E$. More technical details can be found in our tutorial paper [13].

We have reduced the verification of the invariant I on the composition of `PIPELINE` and `ISA` into four verification subtasks: one invariant verification lemma and three refinement verification lemmas. In each of these lemmas, the relevant signals (over which reachability computation needs to be performed) as determined by the cone-of-influence heuristic are a small subset of the signals in `PIPELINE` and `ISA`. For example, in the invariant verification lemma, the set of relevant output signals are `regs`, `isaRegs`, `pipe1.op`, `pipe2.op`, `pipe2.res`, `pipe2.dest`, and `pipe1.dest`. The fields `src1` and `src2` of `pipe1` are no longer needed because the definition of `pipe2.res` does not depend on them. Similar reductions are obtained in the three refinement verification lemmas. Each of these lemmas can be checked automatically by a model checker. This decomposition can be generalized to a pipeline of arbitrary depth so that a particular stage of the pipeline is verified using the abstract definitions of other stages on which it depends.

4 Coverage-directed simulation

Methods that combine model checking with compositional decomposition techniques, such as assume-guarantee reasoning, scale well to large designs but are not completely automatic and their use requires changes in the current hardware design process. Other innovations, such as partial-order reduction and approximate reachability analysis, have considerably broadened the scope of completely automatic methods but industrial designs still remain outside their reach. As a result, designers continue to rely heavily on software simulation for functional validation.

However, the conventional approach of simulation driven by random and hand-written inputs has become unsatisfactory due to growing design complexities. The validation quality attained is hard to quantify and labor intensive to improve. This issue is the focus of recently intensified research and development activity. In this section, we present one such approach: coverage-directed simulation.

In comparison to the huge state spaces of industrial circuits the amount of simulation that can be performed is very small. Therefore, careful use must be made of the limited simulation resources, i.e., as many qualitatively distinct input patterns must be simulated as possible. A computationally feasible mathematical formulation of this goal has not been accomplished so far. In the absence of such a formulation, verification coverage metrics [14] have found increasingly wider use as heuristic means to guide the validation process.

Coverage metrics are associated with a class of structures in the syntactic or semantic description of a design or its specification. A gap in coverage provides a concrete goal for input stimulus generation. Traditionally, input generation to improve coverage has been done by designers and verification engineers through inspection, trial and error. We believe that the greatest near-term improvement to the verification process will come from the development of automatic input stimulus generation tools targeting coverage.

Coverage metrics can be classified into four groups based on the form of the description to which they refer [14].

(i) *Code coverage metrics:* Metrics defined on an HDL description, such as line, branch, expression, or control path coverage.

(ii) *Metrics based on circuit structure:* Metrics defined on a circuit netlist, such as toggle, register-to-register paths coverage, datapath-control interface coverage.

(iii) *Metrics defined on finite state machines:* Metrics defined on a finite state machine that is either extracted from the design or is a manually constructed abstraction of it. Typical metrics are state, transition, and path coverage.

(iv) *Functional coverage metrics:* Metrics that refer to the various modes of functionality or tasks that a design is expected to perform. These metrics often take the form of lists of error prone execution scenarios or transaction sequences.

High code coverage is often a minimal requirement in the validation process, is relatively easily achieved, and gaps in coverage can often be addressed by test writers without much difficulty. Metrics based on circuit structure are good sanity checks. However, in order to address conceptual and design errors, metrics in categories (iii) and (iv) are more appropriate. It is difficult to provide automation for functional coverage metrics; they are often addressed by writing test cases associated with each coverage requirement. We believe that metrics in category (iii) provide a good compromise between relevance to non-trivial design errors and the possibility of providing automation for vector generation. In the rest of this section, we will focus on these metrics and associated vector generation methods.

4.1 State-based coverage

It is impractical to cover the entire state space of an implementation level description by simulation. To ensure that simulation runs explore enough qualitatively distinct regions of the state space, state-based metrics defined on

smaller, more abstract modules called *coverage modules* are used. At each step of a simulation run, the state c of the coverage module is determined uniquely in terms of the state m of the simulated module by an abstraction function f_{abs} such that $c = f_{abs}(m)$. If the coverage module is extracted from the implementation by selecting a subset of the signals, f_{abs} is a simple projection. For hand-written abstractions, a mapping may need to be written to represent f_{abs} in order to measure coverage during simulation. Figure 3 presents the state transition diagram of a coverage module for the pipeline example of Section 2. This module is obtained by extracting the signals `pipe1.op`, `pipe2.op` and the predicate `src1 = pipe2.dest` from `PIPELINE`.

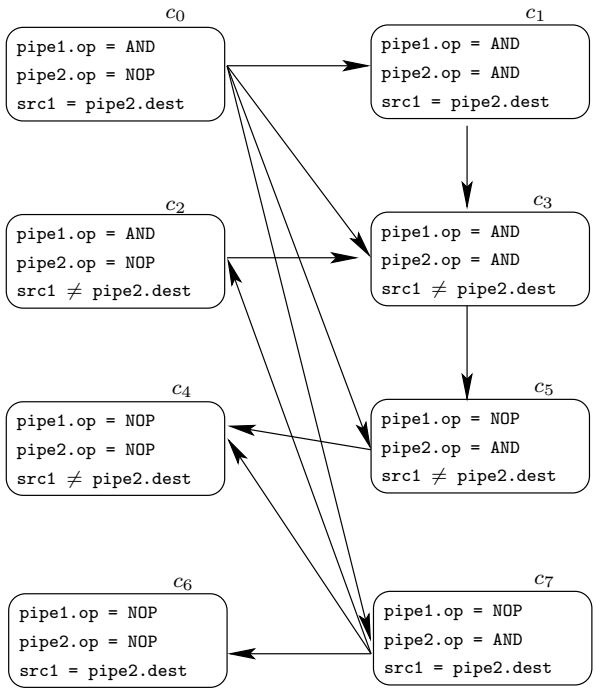


Figure 3: A coverage module for `PIPELINE`. Some transitions are not shown to keep the figure simple.

For the purposes of state-based coverage, a *simulation run* is a path in the state transition diagram of the implementation that starts at an initial state. An implementation state is *visited* if it lies on such a path traversed during a simulation run. A state c of a coverage module C is *covered* if $c = f_{abs}(m)$ for a state m of the implementation module M that is visited during a simulation run. A transition $c \rightarrow \tilde{c}$ of the coverage module is covered if there exist implementation states m and $\tilde{m}$ such that $\tilde{m}$ is visited immediately following m during a simulation run and $c = f_{abs}(m)$ and $\tilde{c} = f_{abs}(\tilde{m})$. A path $c_1, c_2, ..., c_n$ is said to be covered if the transitions $c_1 \rightarrow c_2 \rightarrow ... \rightarrow c_n$ are covered consecutively. In the `PIPELINE` example, executing instructions $R0 \leftarrow$ `AND`$(R1, R2)$, $R1 \leftarrow$ `AND`$(R0, R2)$, $R1 \leftarrow$ `AND`$(R2, R2)$, and `NOP` in that sequence first brings the coverage module of Figure 3 to state c_1 from an arbitrary state, and makes it visit c_3 and c_5. The states c_1,c_3 and c_5, the transitions $c_1 \rightarrow c_3$ and $c_3 \rightarrow c_5$, and the path $c_1 \rightarrow c_3 \rightarrow c_5$ are covered.

4.2 Coverage modules

A coverage module determines the part of the implementation state space that is explored during simulation. The coverage module should be an abstraction of the design, otherwise large parts of the state space of the design might remain unexplored even with full coverage. Such a module is either written manually or extracted from the design by selecting a small number of signals and constructing a module that only refers to these signals.

A manually-constructed coverage module has the advantage of concisely capturing knowledge about the architecture and the intended behavior of the implementation. Even in the simple example of Figure 3, choosing the predicate `src1 = pipe2.dest` as a state variable has kept the state space of the coverage module small. Without this choice, which required knowledge of the pipeline architecture, we would have had to include `src1` and `pipe2.dest` as part of the coverage module state. Manually-constructed coverage modules have been successfully used for coverage analysis and vector generation of industrial designs [15, 16, 17]. A drawback of a manually-constructed coverage module is that it may not be a correct abstraction of the implementation.

A coverage module extracted from the design is desirable because it is a correct abstraction by construction. The selection of the signals to be included in the coverage module can be done manually [18] or heuristically [19, 20]. The heuristics for this selection often make a distinction between control and datapath signals in the implementation and attempt to include only the control signals in the coverage module. Since the control-datapath separation is not always obvious, these heuristics can generate a large number of control signals. Other heuristics [21, 22] have been developed for identifying control signals that change most frequently; these signals are given preference for inclusion in the coverage module.

A desirable property of a coverage module is that complete coverage should guarantee detection of a class of design errors. The identification of necessary and/or sufficient conditions for such guarantees is a largely unexplored area. Gupta et al. [23] impose certain restrictions on signals that can be included in a coverage module. These restrictions guarantee that a transition tour of the state machine of the coverage module uncovers all output and transition errors in the implementation.

4.3 Input sequence generation

Gaps in state-based coverage need to be addressed by generating additional input sequences for driving the implementation module. For example, suppose that a transition $c \rightarrow \tilde{c}$ in the coverage module has not been covered. The input sequence generation task consists of finding an initial state m_0 and a sequence of transitions $m_0 \rightarrow m_1 \rightarrow ... \rightarrow m_n \rightarrow m_{n+1}$ in the implementa-

tion such that $c = f_{abs}(m_n)$ and $\tilde{c} = f_{abs}(m_{n+1})$. Recall that our formal definition of a state also includes inputs. Therefore, the task of finding such a path also includes generating the necessary inputs that enable each transition along the path $m_0 \rightarrow ... \rightarrow m_{n+1}$. The generation of inputs is complicated by the fact that there may be additional restrictions on the inputs to the implementation. Not every combination of inputs is allowed; for instance, the input to a processor has to be a legal instruction. The input to a module that is part of a larger system must be chosen from possible outputs that the surrounding modules can generate. These restrictions on the inputs may be sequential as well as combinational.

Traditionally, input sequences have been generated by choosing random values for input signals. Better input sequences can be generated by biasing certain inputs towards particular values during this random selection. Such techniques use few computational resources but typically will not cover all transitions in a coverage module. ATPG techniques, theoretically more complex but often computationally feasible in practice [24], can be used to generate input sequences to cover the remaining uncovered transitions. Validation of industrial designs [15, 16, 17] typically use a multitude of techniques proceeding from the simple to the complex. Recently, Ho et al. [18] have combined a number of input-generation techniques to achieve validation on designs beyond the reach of formal verification tools.

5 Conclusions

Ideally, a hardware design project would start with a modular and abstract specification. An implementation would be constructed through successive refinement of the specification while maintaining the interfaces between modules. Global properties can be verified using the specification, and the implementation is guaranteed to satisfy them. The complexity of modern designs, the need for design reuse, and the prohibitive cost of functional validation has started to push the design process in this direction. The compositional verification method based on assume-guarantee reasoning presented in Section 3 could be very useful for such a design methodology.

However, current design practices are very far from this formal top-down design methodology and are likely to remain so in the near future. The coverage module driven simulation methods presented in Section 4 are essential for validating current designs, and providing a bridge to a more formal design discipline. Simulation is unlikely to go away as a validation tool though; abstract models need to be debugged by simulation before formal refinement proofs can be attempted. Coverage guided techniques are extremely valuable for getting the maximum from simulation. We believe that formal techniques together with coverage-based simulation will be essential components of the functional validation toolkit of the future.

References

[1] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logic of Programs*, Lecture Notes in Computer Science 131, pages 52–71. Springer-Verlag, 1981.

[2] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Fifth International Symposium on Programming*, Lecture Notes in Computer Science 137, pages 337–351. Springer-Verlag, 1981.

[3] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[4] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[5] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[6] K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science 1427, pages 110–121. Springer-Verlag, 1998.

[7] Á.Th. Eíriksson. The formal design of 1M-gate ASICs. In G. Gopalakrishnan and P. Windley, editors, *FMCAD 98: Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science 1522, pages 49–63. Springer-Verlag, 1998.

[8] T.A. Henzinger, X. Liu, S. Qadeer, and S.K. Rajamani. Formal specification and verification of a dataflow processor array. In *ICCAD 99: IEEE/ACM International Conference on Computer Aided Design*, pages 494–499. IEEE Computer Society Press, 1999.

[9] R. Jhala and K.L. McMillan. Microarchitecture verification by compositional model checking. In *CAV 2001: Computer Aided Verification*, Lecture Notes in Computer Science 2102, pages 396–410. Springer-Verlag, 2001.

[10] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[11] K.L. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem*. Kluwer Academic Publishers, 1993.

[12] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[13] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *ICCAD 2000: IEEE/ACM International Conference on Computer Aided Design*, pages 245–252. IEEE Computer Society Press, 2000.

[14] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 18(4):36–45, July 2001.

[15] M. Kantrowitz and L.M. Noack. I'm done simulating; now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor. In *DAC 96: Design Automation Conference*, pages 325–330. ACM, 1996.

[16] S. Ur and Y. Yadin. Micro architecture coverage directed generation of test program. In *DAC 99: Design Automation Conference*, pages 175–180. ACM, 1999.

[17] M. Benjamin, D. Geist, A. Hartman, G. Mas, R. Smeets, and Y. Wolfstahl. A study in coverage-driven test generation. In *DAC 99: Design Automation Conference*, pages 970–975. ACM, 1999.

[18] P.-H. Ho, T.R. Shiple, K. Harer, J.H. Kukula, R. Damiano, V.M. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal and simulation engines. In *ICCAD 2000: IEEE/ACM International Conference on Computer Aided Design*, pages 120–126, 2000.

[19] D. Moundanos, J.A. Abraham, and Y.V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47(1):2–13, January 1998.

[20] R.C. Ho and M.A. Horowitz. Validation coverage analysis for complex digital designs. In *ICCAD 96: IEEE/ACM International Conference on Computer Aided Design*, pages 322–325, 1996.

[21] J. Shen and J.A. Abraham. An RTL abstraction technique for processor microarchitecture validation and test generation. *Journal of Electronic Testing: Theory and Application*, 16(1-2):67–81, 1999.

[22] J. Shen and J.A. Abraham. Verification of processor microarchitectures. In *Proceedings of the 17th IEEE VLSI Test Symposium*, pages 189–194. IEEE Computer Society Press, 1999.

[23] A. Gupta, S. Malik, and P. Ashar. Toward formalizing a validation methodology using simulation coverage. In *DAC 97: Design Automation Conference*, pages 740–745. ACM, 1997.

[24] M.R. Prasad, P. Chong, and K. Keutzer. Why is ATPG easy? In *DAC 99: Design Automation Conference*, pages 22–28, 1999.