VERIFICATION MODEL STRUCTURES FOR DIGITAL SYSTEMS DESIGN

SERGEY L. FRENKEL

The Institute of Informatics Problems,
Russian Academy of Sciences,
Vavilova 44,2, 117333, Moscow, Russia.
E-mail: slf-ipiran@mtu-net.ru

Abstract Exponential computational complexity of digital systems formal verification algorithms excludes any possibilities of full-automatic verification of complex digital systems. On the other hand, the informal design simulation is also impractical time-consuming. Probably, the possible outcome is to form a verification strategy which, on one hand would combine both approaches, and on the other hand would include a guide to issue verifications algorithms-and-tools appropriate for a given design. It implies a characterization of both verification algorithms and design process. In fact, it means a structurization of various models of design, which are used both explicitly and implicitly during design verification activity.
This paper, relying on the previous experience in testability design planning [1] as well as corrent publications in formal verification areas considers some possibilities of planning of digital systems verification activity to achieve high degree of functional verification.

Key words: formal verification, design verification, digital circuits simulation.

1. INTRODUCTION

Designers of complex digital systems (ASIC, application-specific/general-purpose microprocessors (MP)), etc.) need validation methods and tools to guarantee a perfect design before a process of its manufacturing is started. Errors detected after start of fabrication lead both to added production costs and delay the product. This delay may be very critical issue of market control. For example, some data in [2] shows that loss due to late marketing for 10-15 weeks may be up to half million USD.

This validation is performed mostly as a "verification", checking if a system design is correct with respect to a specification (which is understood here as an initial description of aimed design on a given represenation level (e.g., finite –state machine, register-transfer (RTL), or gate level).

The traditional and the most common method of the verification is verification via simulation. The alternative is so-called "formal verification" [3]. However, both these approaches have some drawbacks of high computational requirements. Thereby, the complexity of simulation-based methods is due to the large number of test vectors needed to manifest all functional issues, and the complexity of the formal verification of large designs is due to very large state spaces, which cannot be handled even by such techniques as implicit state space traversal.
For example, in sequential circuits verification a central problem is the reachability analysis. In this activity, the properties to be checked by an automatic verification have to be reachable from the start state. Reachability analysis is the task of finding this set. If a system is represented as a finite-state machine (FSM), reachability analysis corresponds to a traversal of the state transition graph of an FSM, that as it is well-known may contain billions of nodes [3].

Strictly speaking, the same situation from the point of view of automatic (synthesis-directed) and simulation methods interaction takes place in other areas of Electronic CAD activities, first of all in test pattern generation (TPG). In this area a test designer also has to consider a trade-off between the exponential complexity of automatic test pattern generation (ATPG) (true synthesis) and the necessity to use various simulation tools (a "synthesis through analysis") to check if an input test vector ("candidate to test") provides detection of a fault considered. In fact, this methodology changing means the change of *design specification model*. While the design specification for ATPG consists merely of the

 ISBN 3-936150-25-7

circuit description, the simulation-based TPG requere also explicit input vector set description.
In other words, the reasonability of using either design methodologies depends on the suitable test design cost, which depends on labor force cost, equipment cost, time-to-market etc. A choice of ways of the design goal achieving can be considered as a design strategy planning. Various design testability measures may be used as the cost function during the strategy planning [1]. As it has been shown in this work, any relevant design cost function should be in monotony dependence on any testability measure, thereby, any fault coverage measure is a functional of the testability ones (with given TPG methods). Also note, that ATPG practicability depends greatly on the used fault model. Definition of well-known stuck-at-fault model in sixties [4] has led to ATPG performance increase dramatically. This is because of the considerable decreasing of the considered faults set. So, such issue should be studied for formal verification (FV) activity, namely what kind of bugs detecting model could be more reasonable.

In this paper we consider some factors of design functional verification cost together with various aspects of verification models and design features.

Let us emphasize that one should distinguish between the *verification algorithms development* activity and design functional verification activity *on the whole*.
In the first case the computational complexity of verification algorithm will serve as an indicator of practicability, while (besides the algorithm complexity) a complex cost function (labor cost, equipment cost, design tools cost, time-to-market) is a reasonable indicator in the second case. Thereby, a verification algorithm properties are only part of factors of design verification cost.

So, let us consider what kind of means may a designer use to control mentioned above factors planning his design verification activity.

For this aim, in Section 2 all principle components of this activity will be outlined. Section 3 describes some well-known tools from the point of view of design specification impact on overall verification cost. Sections 4-5 describe some design properties and possible design decomposition techniques.

2. ABOUT VERIFICATION PROCESS AND ITS COMPLEXITY

Design verification (Model checking [3], in particular) activity in industry uses the following methodology: A verification engineer reads the specification, sets up a work environment and then proceeds to present the model checker with a sequence of properties in order to verify the design correctness A design can be quite large nowadays. As a result the set of properties written and verified becomes large as well, to the point that the engineer loses control over it.

One of the basic questions is: "Have I described enough properties?" [5]. The current solutions consist in manually reviewing of the property set. It is important that the decision if it is possible to verify the correctness (both functional and timing) of a given design depends on many organizational issues. In fact, these issues are determined by the cost (either in money or in labor time terms) of the result obtaining, and, in the end, depend on the verification process planning and organization. Obviously, to provide the verification scenario planning we have to consider and define all features of target system having an impact on the verification algorithm complexity and, correspondingly, on the choice of preferable verification algorithm, some characteristic of design process to guide a verification process, supposing, first of all, that a cost model of design verification process is available.

In general, we can represent amount cost C_V of a design verification as :

$$C_H^M + C_E^M + C_{ad}$$

where index M= {fv, s} reflects one of verification method, namely, either formal verification (fv) or via simulation (s),
C_H^M means a “human” cost factor, which is the cost of various verification models development and manual input data preparing (and, maybe, a software supporting and modification), C_E^M is an equipment cost including, for example, amortization cost, the “machine time” spent up to verification result, a software acquisition cost, power resources costs etc., C_{ad} is any additional expenses. These partial costs depend on the way of verification.

As we try to deal merely with formal methods, the human (or manual) component of the above expenses correspond to a model development and description (ideally, using merely some hardware description language, e.g. VHDL, and design properties (e.g., in terms of some temporal logic [3]) , and, maybe, to some programming activity. This work requires very high qualification of a verification engineer who has to know all modern logical-mathematical techniques (computational tree logic (CTL), model checking, etc.). Time of a verification algorithms execution

depends both on the algorithm and size of the circuit designed. Obviously, C_E is a monotone-increasing function of the time. Formally, the model checking algorithms are linear with respect to number of states, but, the number of the states increase exponentially with number of terms of logical formulas describing the verification conditions. Thus the "machine" cost of formal verification is $C_E^{fv} \sim f_p(N)$, where $f_p(N)$ is a power function (exponential, in particular) of number of variables, describing the verification problem. Note, that even dealing with some components of entire systems e.g., with some buffers of a microprocessors [Biesse01] we encounter with thousands of variables, that leads to huge numbers of states. Correspondingly, several days may be required to check simple properties of such designs even using rather power platforms, e.g. 700 MHz 64-bit Alpha [6].

As for design bugs finding via simulation, then $C_H^M = C_H^R + C_H^F$, where C_H^R stands for random testing (simulation our design under random- generated tests to observe the design bugs)), C_H^F corresponds to cost of so-called "focused" testing, which are some hand-generated tests to cover specific areas of design, not covered by the random tests. Obviously, this activity supposes some involving of the design developers. For example, the tests may be focused to detect some bugs of caching mechanisms, ALU, etc.

However, it should be taken into account that such activity may require to involve many technicians in the simulation process to run hundreds focused tests variants! Although, in general, the computational of computational complexity of simulation is a quadratic relatively to variables number, C_E^S should not be considered as such function, because the simulation of various parts of the design usually is very redundant from the point of view of design bug checked. So, although for separate design components as a rule $C_H^s < C_E^{fv}$, it may be not true for the design as a whole. So, the way out should be based on trade-off between using of simulation and formal verification approach. The table 1 shows a typical example (verification of a memory bus adapter design) of this compromise [7].

Table 1 Design bugs detected with various techniques

Verification techniques	Bugs founds (%)
separate unit simulation	41
formal verification	24
visual design analysis	20
entire chip simulation	15

However, from the point of view of labor cost, an increase of formal verification weight would be very attractively.
Note that besides the computation complexity, simulation-based methods are no longer adequate for complex hardware (HW) designs. Although simulation can catch many design error, part of bugs are frequently sleeping through. Detecting by simulation of every bug resulting from the complex interaction of concurrent event may be very time-consuming task. In particular, in the considered instance, about 40% of bugs that had been found with formal verification, it turned out impractically to find with any simulation tools [7].

Let us consider some possibility of formal verification (FV) cost reducing.
For this aim we must define and fix, on one hand, various properties of FV algorithms/tools, and on the other hand, various design features affecting the FV cost.

Since this is a combinatorial problem, and as it is well known, combinatorial algorithms may mostly be realized only by the problem description decomposition, we need also to have a characteristics of decomposition ability.

3. VERIFICATION TOOLS AND DESIGN SPECIFICATION ACTIVITY

Since effectiveness of design verification depends on adequacy of logical functions verified representation, it is important that a design tools selected for FV activity would allow the using of various Boolean function representation techniques. Thereby, this representation may depend on both design and requirements specification manner (model).
For example, in [8] the highest level description of a microprocessor is given as an instruction-set specification. At this level the verification may be performed either from actual pipeline design description or representing the stream of executed instructions with a table [8**],** which describes the effect of individual instructions.

However, there are many properties of the pipelined machine and instructions that can be more easily expressed and reasoned about with help of some tables [8]. For example, a *Read After Write dependenc*y between instructions is much easier to represent using our instruction table instead of lifting the necessary information from design.

The basic structure of the design specification for formal verification is Computational Tree Logic

(CTL) [3].For example, well- known VIS package [The release 1.4. of VIS: http://vlsi.colorado.edu/~vis] uses a Verilog front-end and supports fair model CTL checking, language emptiness checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis.

In a program called EMC (Extended Model Checker [9]) the model checking is solved using efficient graph-traversal techniques. Thereby, if the model is represented as a state transition graph, the complexity of the algorithm is linear in the size of the graph and in the length of the formula. However, an explosion in the size of the model may occur when the state transition graph is extracted from a finite state concurrent system that has many processes or components (e.g. dealing with simultaneously-performed six instructions in Alpha processor [10].

The CUDD package provides functions to manipulate Binary Decision Diagrams (BDDs) [http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html], and Zero-suppressed Binary Decision Diagrams (ZDDs represent switching functions like BDDs, however, they are much more efficient than BDDs when the functions to be represented are characteristic functions of cube sets, or in general, when the ON-set of the function to be represented is very sparse. But they are inferior to BDDs in other cases.). The CUDD package can be used in three ways:

- As a black box . In this case, the application program that needs to manipulate decision diagrams only uses the exported functions of the package. The rich set of functions included in the CUDD package allows many applications to be written in this way.

- As a clear box . When writing a sophisticated application based on decision diagrams, efficiency often dictates that some functions should be implemented as direct recursive manipulation of the diagrams, instead of being written in terms of existing primitive functions. - Through an interface. Object-oriented languages like C++ and Perl5 can free the programmer from the burden of memory management.

In the package Almana, (developed at the LaBRI (Universitr e Bordeaux-1)) a *Heuristic methods* based on analysis of the original boolean formula abound is used, and can be subdivided into *static* techniques, that inspect the formula off-line.
Being very popular, these dynamic methods present many problems. The first is that they require that we have already constructed the BDD or some part of it in memory, which is impossible for large systems. A more troublesome problem is that existing techniques are based on *sifting*, which exchanges adjacent variables. Unfortunately, in real systems variables come in *blocks* of related variables, that need to be kept together in the final order or the size explodes.
Note, that the best known BDD-based algorithm for finding an optimal order is of complexity $O(n^3)$, where n is a number of variables.

In tools which are based on Bounded Model Checking [3] accept a subset of the SMV (Symbolic Model Verification)language in which the user can specify a finite state machine and a temporal specification.

Given a bound *k*, BMC outputs a propositional formula which is satisfiable iff there is a counterexample of length *k*. An efficient implementation of the Davis-Putnum technique [11] and PROVER [12] are based on Stalmarck's method to decide propositional satisfiability.

Note, that a lot of modern tools are based on a philosophy of "Satisfability solvers" (SAT)-base model checking, which sometime is considered as an alternative to BDD approach (although, as remarked [13] SAT may be considered as "an interesting complement to model checking with BDDs"). In general, SAT algorithms mission is to decide whether there exists a satisfying assignment for the corresponding formula. Thereby, in spite of mentioned above remark on relationship of BDD and SAT techniques, in [6] was shown that the SAT method for bounded model checking can reduce the verification runtime from days to minutes on real, deep, microprocessor bugs when compared to a state-of-the-art BDD-based model checker.

So, basic features of algorithms underlining various verification tools are good basis for their comparison in the framework of a verification procedure planning.

4. SOME EXAMPLES OF TARGET DESIGN PROPERTIES IMPACT

Intuitively, the complexity of the BDD is a function of how much information must be remembered as one passes from one level of the BDD to the next (i.e., from one variable to the next). For example, in [14] a pipeline examples which were verified had approximately $5 * 10^{20}$, states, which puts it far outside the range of model checkers like the one reported in [3]. It required a BDD with 42000 nodes to represent the transition relation. These data are concerned very simple pipelines that perform three-

address logical and arithmetic operations on a register file. The complete state of the register file and pipe registers are modeled. The pipelines in this design had three stages. On the first stage, the operands are read from the register file, on the second stage an ALU operation is performed, and on the third stage the result is written back to the register file. ALU has a register bypass path, which allows the result of an ALU operation to be used immediately as an operand on the next clock cycle, as is typical in RISC instruction pipelines. The inputs to the circuits are an instruction code, containing the register addresses of the source and destination operands, and a STALL signal, which indicates that the instruction stream is stalled.

However, what kind of the circuit's properties enabled such impressive results?

The point is that the information stored from one "bit slice" of the data path to the next was rather small; it amounts to the state of the control bits plus at most the value of the ALU "carry" bit. In particular, this amount of information is not increased as one increases the number of bits, so the BDD becomes deeper, but no "wider".
Although these research [14] are concerned the timing verification, these conclusions are true also for functional verification as in both cases verification algorithms use a Boolean encoding of the elements of the model domain, and represents relations with Boolean decision diagrams.

So, in case the information quantity stored from one "bit slice" of the data path to the next is a system designed characteristics affected the BDD using effectiveness.

5. ABOUT DECOMPOSITION POSSIBILITIES

Let's consider what current state-of-the-art in formal verification may suggest us to decompose design as a way of verification cost reduction.

Mostly a design description decomposition is trying to avoid the state explosion problem .The goal is to verify properties of individual components, infer that these hold in the complete system, and use them to deduce additional properties of the system. It may also be necessary to make assumptions about the environment (that is both other components of the system and various external signals). This approach may be exemplified by Pnueli's assume-guarantee paradigm [15]. A formula is true if whenever M is a part of system satisfying', the system must also satisfy .

Since we consider this problem from the point of view of design tool using, let us consider what kind of requirements the model checking should meet to.

First of all, it must be able to check that a property is true for all systems which can be built using a given component. More generally, it must be able to restrict to a given class of environments when doing this check. It must also provide facilities for performing temporal reasoning. Most existing model checkers were not designed to provide these facilities. Instead, they typically assume that they are given complete systems. A way to obtain a system with the above properties is to provide a preorder on the finite state models that captures the notion of "more behaviors" and to use a logic whose semantics relate to the preorder [16, 17].

Note, that along with design decomposition it can be used also various types of circuit's reduction. For example, the merge buffer, an important component of the Alpha MBox for a next-generation Alpha chip has been considered in [6].The function of the merge buffer is to receive requests to write into memory, and to reduce the trajectory on the memory bus by merging stores to the same physical address. The merge buffer is essentially a large buffer with a very complex policy for reading in entries, merging stores, and writing out stores to the memory. It has about 14 400 latches, 400 primary inputs, and 15 pipeline stages. The pipeline has complex feedback that prevents us from retiming away latches. The original RTL description of the circuit is used as design input.

First of all, the authors tried to reduce the size of the model for verification using standard model checking technology. The idea is to remove portions of the state in the circuit in ways that do not alter the circuit behavior with respect to the properties of interest. After the reductions, the merge buffer has about 40 primary inputs. When the merge buffer is in use, these inputs will be connected to the four subboxes with which the merge buffer communicates. . The final model has about 600 state nodes in the cone of most properties. However, before sending the model to a tool input, it is needed to write down the property of interest in a format that the tool we want to use accepts. Given the model and the property, the verification tool then either produces a failure trace, or tells us that the property is true.

6. DISCUSSION AND CONCLUSION

Full-automatic formal verification of complex processors design is a dream of all system designers. Unfortunately, its exponential complexity is well

known, that, it seems, excludes this dream realization, at least for large designs with very large state spaces, which cannot be handled even by techniques such as implicit state space traversal. Obviously, the result of such activity has to be obtained even if system description is so large (either in terms of state space or formulas clauses number) that no formal verification algorithm which could allow to do it Very obvious way to achieve it is a combination of formal and informal (simulation) verification models. Since complex microprocessors systems design verification activity deals, in general, with many optional variants, it should be useful to have a characterization of both verification algorithms and verification process on a whole, which includes the decomposition issues, dividing possible (potential) design bugs classes between formal verification and simulation, final quality analysis etc.

Obviously, we need a guide to provide this hybridization. Following well showed itself conception of coverage analysis, use widely in test pattern generation practice, it would be very attractively to have also similar one for the design formal verification. Some steps towards this notion development are just in progress [Hoscotte99, Chochler01]. As for formal verification, the notion of coverage in functional verification is to cover the entire functionality specification required from the implementation. This notion involves two questions:

-whether we can provide (to take into account) (explicitly or implicitly) all possible input sequence,

- whether the specification contains a sufficient set of properties.

So, the coverage analysis is a search of some dissimilarity between the implementation and specification, which points out a possibility to reduce target design description to enhance the verification possibilities.

Along with coverage characteristic, it is important, not resolved problem is to characterize both formal verification and simulation tools that could be chosen for design verification. They may be characterized by a rate characteristic, e.g. as as number of states per second for formulas. Thereby, on one hand, this rate depends on a way, in which design specification is described, and on the other hand, specification language may determine qualification requirement of personal, affecting the verification cost (e.g., time consuming).

REFERENCES

1. S. Frenkel, Testability Measure as a Test Pattern Generation Cost Function, in *Proc. of* 7th IEEE North-Atlantic Workshop (NATW'98), West Greenwich, RI, USA, 1998, pp.42-50,
2. W.Rosenstiel, Rapid Prototyping, Emulation and Hardware/Software Co-debugging, in "SYSTEM-LEVEL SYNTHESIS" ed. by A. A..Jerraya and J.Mermet, NATO Science Series, Kluwer Academic Publisher, 1998, pp.219-262,
3. E. Clarke 1 , O. Grumberg , and D. Long, "Model Checking", in Springer-Verlag Nato ASI Series F, Volume 152, 1996,
4. R.G. Bennets, Design of Testable Logic Circuits, Addison-Wesley Pub. Company, 1984
5. S. Katz, D. Geist, and O. Grumberg. "Have I written enough properties ?" a method of comparison between specification and implementation. In *10th CHARM*E, *LNCS* 1703, pp. 280–297, 1999,
6. Per Bjesse, Tim Leonard , and Abdel Mokkedem, Finding Bugs in an Alpha Microprocessor Using Satisfability Solvers, in Proceedings of 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, LNCS 2102, p. 454,
7. T.Shlipf et al, Formal verification made easy, in IBM Journal of Research and Development, vol. 41, No 4/5, 1997,
8. Jun Sawada Design Verification of Advanced Pipelined Machines, Doctoral Dissertation, University of Texas at Austin, Computer Science Dept, 1996,
9. M.Clarke, O.Grumberg, D.E.Long: "Model Checking and Abstarction", ACM-TOPLAS, Vol. No. 5, pp.1512- , September 1994.
10.Alpha 21264 Microprocessor Hardware Reference Manual, Compaq Computer Corporation 2000,
11.E.Dantsin et al, Algorithms for SAT and Upper Bounds on Their Complexity, Electronic Colloquium on Computational Complexity, Report No. 12 (2001),
12. M. Sheeran, S. Singh, and G. Stalmarck, Checking safety properties using induction and a SAT-solver. In Formal Methods in Computer Aided Design, 2000,
13. Per Bjesse and Koen Claessen, SAT-based Verification without State Space Traversal 2000, Journal of Global Optimization, , 1{36)
14. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, \Symbolic Model Checking:10 20

States and Beyond," Information and Computation, vol. 98, no. 2, pp. 142-170, 1992.
15. A. Pnueli. In transition for global to modular temporal reasoning about programs. In K. R. Apt, editor, Logics and Models of Concurrent Systems, volume 13 of NATO ASI series. Series F, Computer and system sciences. Springer-Verlag, 1984.]
16. Yatin Hoskote, Timothy Kam, Pei-Hsin Ho, Xudong Zhao, Coverage Estimation for Symbolic Model Checking, DAC'99, p.300
17. Hana Chockler et al, A Practical Approach to Coverage in Model Checking, in Proceedings of 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, LNCS 2102, p. 66.