Verification of Electronic Systems

Alberto L. Sangiovanni-Vincentelli* Patrick C. McGeer† Alexander Saldanha‡

1 Introduction

The complexity of electronic systems is rapidly reaching a point where it will be impossible to verify correctness of the design without introducing a verification-aware discipline in the design process.

Even though computers and design tools have made important advances, the use of these tools in the commonly practiced design methodology is not enough to address the design correctness problem since verification is almost always an after-thought in the mind of the designer. A design methodology should on one hand put to good use all techniques and methods developed thus far for verification, from formal verification to simulation, from visualization to timing analysis, but should also have specific conceptual devices for dealing with correctness in the face of complexity such as:

- Formalization, which consists of capturing the design and its specification in an unambiguous, formal "language" with precise semantics.
- Abstraction, which eliminates details that are of no importance when checking whether a design satisfies a particular property.
- Decomposition, which consists of breaking the design at a given level of the hierarchy into components that can be designed and verified almost independently.

These mechanisms can be applied to different classes of designs: from embedded controllers to computers, from microprocessors to digital-to-analog converters. They are not only useful in the verification process but also in the design process *per se* making verification itself unnecessary in some cases. For example, formalization of the design specifications is required for formal verification but it also helps in design transfer between different organizations eliminating the risk of losing knowledge about the design and its specifications, thus making verification before and after the transfer unnecessary. We argue that almost all advances in verification stem from the application of these three basic concepts. However, the application of each of these principles has been performed at localized levels of the design hierarchy leading to a plethora of models, where each tool assumes a different model. Maintaining coordination and consistency of these multiple models has rightly become a designer's nightmare.

This paper is organized as follows: in Section 2 we will review the available verification tools. In Section 3, formalization will be investigated in several contexts. In Section 4, abstraction will be presented with a set of examples. In Section 5, decomposition will be introduced. Finally in Section 6, a design methodology that includes all these aspects will be proposed.

2 CAD Tools for Verification

Traditionally verification has been carried out by reproducing the behavior of the design with an approximate implementation, a prototype, of the design, or with mathematical techniques involving the construction of a model and running a computer simulation. Virtual or real measurements were taken to assess the quality of the design.

Simulation is applied at all levels of a design: an incomplete list of simulators at various levels includes circuit simulation (*e.g.* Spice), switch and transistor level simulation (*e.g.* Cosmos [6]), gate level simulation (*e.g.* Verilog-XL), register transfer level simulation (*e.g.* Verilog-XL) and behavioral or system level simulation (*e.g.* Ptolemy [7]).

In both simulation and emulation, the correctness of the design is asserted only with respect to the inputs provided by the environment or by its model and with respect to the measurements chosen by the designer.

In a more formal approach to design as we advocate in the next section, the set of properties and performance indices are explicitly expressed. Sometimes these properties may be verified by setting up a comprehensive set of experiments or simulation runs, other times it is impossible to have a reasonable confidence that the experiments yield the appropriate answer. Properties such as absence of deadlock and fair access to resources to be verified for communicating processes, require a formal approach to verification. Formal verification is an approach that has been explored for the past 20 years but the level of attention paid by the design community has been raised only recently.

The categories of tools that provide formal proofs of correctness, and a counter-example in the case of an error are as follows: *Equivalence checking* where a combinational logic level design is compared against another design for functional equivalence; *Language containment and model checking* where the system is described as a collection of FSMs and the property to be verified is specified as an automaton and temporal logic formula respectively; and, *Theorem proving* where the verification problem is stated as a theorem and a set of axioms (built-in or user-specified) is used to construct a proof of the theorem by proving a set of intermediate results.

3 Formalization

By a *formal* model of a design, we mean a model with *precise, unambiguous* semantics. Formalization is critical: without a formal model of a design, the very meaning of "verification" becomes fuzzy and problematic.

There is a broad range of potential formalizations of a design, but most tools and designers describe the behavior of a design as a relation between a set of inputs and a set of outputs. This relation may be informal, even expressed in natural language. It is easy to find examples where informal specifications resulted in unnecessary redesigns! In our opinion, a *formal model of a design* should consists of the following components:

- A set of explicit or implicit equations which involve input, output and possibly internal (state) variables.
- A set of properties that the design must satisfy given as a set of equations over design variables (inputs, outputs, states).
- A set of performance indices which evaluate the quality of the design in terms of cost, reliability, speed, size, etc., given as a set of equations involving design variables.
- A set of constraints on design variables and on performance indices specified as a set of inequalities.

*University of California - Berkeley CA
†Cadence Berkeley Laboratories - Berkeley CA
‡Cadence Berkeley Laboratories - Berkeley CA

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

Note that there are cases where one or more of the sets described above may not be given or may be implicitly stated, although we do not advocate this practice.

Such formalisms have strong advantages. First and foremost, virtually all come with an algebra, permitting to manipulate the design precisely. With a formal model, the effect of an operation or transformation on a design is always well defined. The transitions along the design flow can be smooth and with few or no special conditions that are design-dependent. At every stage of the design flow, one can also argue about the correctness of the intermediate result or final implementation because of the existence of this underlying formal model. Second, the division into equalities and inequalities neatly separates verification and optimization problems. Finally, the notions of *abstraction* and *decomposition* have precise, easy-to-understand definitions over systems of equations.

However, formally describing a design is not enough to set the stage for a correct verification process. Any design is part of a larger system. The interface between the design and its environment is through input and output signals. Hence, to verify the behavior of the design a model of the environment must be given. The environment defines the input domain. Often, the environment is not known precisely. Thus, the model may often be given in terms of distributions over intervals or of stochastic equations. Often, verification fails to give correct answers when the model of the environment has not been precisely defined.

We can now summarize our discussion with the following statement:

The formalization *of a system composed of a design and its environment consists of a closed system of equations and inequalities over some algebra.*

In the following subsections, we supply some examples of formalisms. Our analysis will progress from more detailed representations to more abstract ones.

3.1 Continuous waveforms and differential equations

When the design is at the transistor level, often the properties to be verified involve the precise knowledge of the current and voltage waveforms. In this case, the appropriate representation for the design variables is a continuous function over time. The design equations are given as a set of ordinary differential equations which combine the input-output relationships for each device with the interconnection equations. Note that the input-output relationships in this case are defined implicitly by the circuit equations. The input-output behavior is obtained by solving numerically the set of differential equations. (This is what a circuit simulator such as Spice does). The properties could be expressed as an inequality which should be satisfied over a time interval of interest. In this case the environment is modeled by the input waveforms. Often the environment interferes with the intended behavior of the system through noise. Noise is then an input that may be described in terms of stochastic differential equations.

3.2 Discrete waveforms and wave equations

The *discrete waveform* formalism is intended to retain the precision over the time domain of the continuous formalism while relaxing the range to a discrete set. It arose in attempting to formalize the delay models in hardware description languages, and, simultaneously, in efforts to solve the so-called *false path* problem in timing analysis. Until this formalism was devised in the early 1990's, a variety of crude formalisms had been used to capture the notion of time, and, in particular, the interaction of timing and function for the purposes of timing analysis. These were usually some combination of Boolean algebra and graph-theoretic concepts, and were generally fairly unsatisfactory. Both standard Boolean algebra and graph-theoretic approaches to timing may be viewed as approximations to this general theory.

The primitive objects in the waveform formalism are *discrete signals*, which are total functions from continuous time onto a finite lattice which obey a *quasi-continuity property*: a signal can't change values without first going through the "uncertain" value.

The operators on the discrete waveform space are any functionals which take in one or more discrete signals and produce a discrete signal. A simple gate in a technology library is an example of such a functional. The Boolean function of the gate, together with its associated *delay model*, forms an *evaluation rule* on wave space, or, put more simply, it tells how this gate takes waveforms on its input pins and produces a waveform on its output pin.

Since the waveforms are static (though infinite) objects, algebraic operations over waveforms are well-defined; since the gate delay model plays the role in this algebra of an operator, it is an algebra effectively parameterized by the delay model; hence it can be used with a variety of delay models.

Though the wave algebra was only formalized quite recently[15], it has been implicit in a number of tools for some time. Though the semantics of most hardware description languages are difficult to capture in either an equational or denotational sense, the "hardware subset" of most HDL's is pretty much defined over a space of waveforms. Augustin[1] first devised an approximation to general discrete waveform theory to analyze the timing models of VAL and VHDL.

3.3 Binary and multi-valued logic variables and Boolean equations

When time is neglected and switch-level logic is not considered, signals can be represented with static variables that take values on a discrete set. In particular, two-valued variables have been the cornerstone of digital design. On two-valued variables, the input-output relationship of each gate can be represented as a Boolean equation.

The Boolean equation formalization is critical because it is the formalism with which we have the most experience. Thirty years of intensive algorithm development have given us good tools for logic synthesis (finding a smallest set of equations equivalent to a given set), equation solution (find an input vector such that all the equations are satisfied), and equation evaluation (evaluate a given set of Boolean equations very quickly). For this reason, Boolean equations are the very heart of modern synthesis, verification, and simulation technologies, and casting a problem as a system of Boolean equations is a major step in solving it.

However, Boolean algebra is not restricted to operate on binary variables. For fifteen years multi-valued logic technology has been developed into a high art, and most of the familiar two-valued logic technologies are easily extended to multi-valued logic. Logic synthesis technology, binary decision diagrams, and rapid function evaluation technology are examples of those technologies which translate easily from the binary to the multi-valued domain. For this reason, two-valued logic signals are now regarded as *encoding* multi-valued signals.

3.4 System Representations

Designs should be entered into a formal framework possibly supported by tools as early as possible in the design process. Formalization at the system level is crucial for real advances in verification. The terminology "system" means different things for different people, here we consider the notion of an electronic system embedded in an environment to which the system has to react with some constraints on the time of response. Embedded real-time reactive systems are in this category. Such electronic systems contain several components from sensors, to data-processing subsystems, from analog circuitry to actuators. Most of them are implemented using a set of existing programmable components such as DSPs and micro-controllers. In this case an implementation is a combination of hardware and

software components, where the software components "customize" the standard programmable parts for the application. The richness of implementation choices makes the use of a unified model for a "system" almost impossible today. Hence, an unambiguous representation has to reflect the heterogeneity of the components of the design as well as the fact that these components interact among themselves in many different ways. Hence, at the system level of abstraction a design and its environment can be represented as a set of communicating entities that may or may not be based on the same model of computation. A definition of this concept is given in [14]. Each entity is described by a set of algebraic equations involving discrete variables and as such can be treated with Boolean algebra techniques. The communication mechanism can also be described by a set of algebraic equations. Time may or may not enter in the description of the entities or of the communications among them.

A process network is a structural model that consists of a set of nodes *representing* processes *and/or* sub-networks, *and a set of* edges *representing* communication links*. Its semantics are determined by: a node model and a communication model.*

Communication among processes involves *writing* and *reading* information among partners. We distinguish between two basic mechanisms, depending on whether a single piece of information (written once) can be read once or many times: *Destructive read* and *Non-destructive read*. Communication can be:

1. *Synchronous* if the write and read operations *must* occur simultaneously, and
2. *Asynchronous*, otherwise. In this case there can be a finite or an infinite number of *buffer locations*, where the information is stored between the instant in which it is written and the instant in which it is read.

Note that non-destructive read is generally asynchronous, and implies a buffer with a single cell (the shared variable). The standard distinctions between *uni-directional* and *bi-directional* communication, as well as between *point-to-point* and *broadcast*, are also useful for classification purposes. The Ptolemy environment was the first to allow the simulation of process networks with heterogeneous entities [7].

Two sets of node models are of particular interest in system design: a control-dominated class based on Finite State Machines (FSMs) and a data-oriented class based on data-flow networks. FSMs in their classical form are not Turing equivalent and hence questions about their behaviors are decidable. However, data manipulations are not easily described. Data-flow representations in their classical form are Turing equivalent, but there are useful restrictions of the model (such as synchronous data-flow) which are not. They are most useful in describing data manipulations.

3.4.1 Communicating FSMs

An FSM is a process whose input/output function can be computed by a finite automaton. The edges of the automaton are labeled with input/output data pairs. A network of FSMs uses broadcast *synchronous non-blocking* communication among the FSMs.

Synchronous languages such as Esterel [3] is a language whose semantic is based on FSM and is of particular interest since it is among the few system-level languages that have unambiguous semantics. The *synchronous hypothesis*, common to all synchronous languages (Lustre and Signal also belong to the class of synchronous languages), states that time is a sequence of instants, between which nothing interesting occurs. In each instant, some events occur in the environment, and a reaction is computed *instantly* by the modeled system. This means that computation and internal communication take no time. This hypothesis is very convenient, because it allows modeling the complete system as a single FSM. This has the advantage that the behavior is totally predictable, because there is no problem of synchronizing or interleaving concurrent processes.

Like all FSM based control-dominated models, data manipulation cannot be done very naturally. Also, having a synchronous model makes it hard to specify components of a system that operates at different rates. Hence Esterel by itself can only be used for process level modeling while the system level modeling of asynchronous communicating processes should be done using another formalism. The perfect synchrony hypothesis simplifies the design process, but also forces the timing constraints to be specified outside Esterel.

3.4.2 Petri nets and data-flow networks

A Petri net is a flat hierarchy. Nodes (usually called "transitions") "fire" by reading from each input and writing to each output. Communication is *asynchronous*, with *infinite buffers* (usually called "places"), with blocking read and non-blocking write. In the pure Petri net model no value is transferred by communications, the only significant information being the possible transition firing sequences.

A data-flow network is similar to a Petri net, but each communication can transfer a value (*e.g.*, integer or Boolean), and buffers have FIFO semantics. Little or no restriction is posed on the function computed by each leaf node in response to a set of communications on its input edges, apart from terminating in finite time. Note that due to the blocking read communication, a node cannot test an input buffer for emptiness. Nonetheless nodes can decide from which input(s) and to which output(s) communications will be performed. The main result concerning data flow networks, which is directly connected with the blocking read communication mechanism, is their *determinacy*. This means that the *sequence* of output values at each node does not depend on the order in which nodes are selected to execute computations and communications, as long as the order does not cause deadlock (*e.g.*, by scheduling a process when one of the buffers it will read from is empty) or starvation (*e.g.*, by never scheduling a process with non-empty input buffers).

4 Abstraction

Abstraction, a most powerful concept, is used in two basic ways: (i) when specifying a design at early stages of the design process to give only the relevant information, (ii) to hide details that are not necessary to assess a particular behavior of the design.

When specifying a design, an important aspect is to express all that is required and known of the design at the moment. Often, designers are forced to put more details than needed to be able to run verification tools. This has the unwanted effect of limiting the design space to be explored or to overcomplicate design exploration. Often, several choices are possible at the early stage of a design and we wish to have a model that leaves a place-holder for these options to be selected later. Nondeterminism is the mathematical abstraction that allows this to happen. Nondeterminism in an FSM setting implies that more than one transition may be taken under the same input. Removing nondeterminism means that the transition relation of the FSM is refined to eliminate the presence of multiple transitions under the same input pattern. This refinement concept is the cornerstone of a design methodology that proceeds from a more abstract model of the design to a more detailed one until an implementation is obtained [13]. Formal verification tools handle nondeterminism so that properties verified on the nondeterministic model still hold after the nondeterminism is refined away thus providing a powerful paradigm for design. Nondeterminism is of great importance to model the environment where the design lives. In this case, nondeterminism reflects the uncertainty about the behavior of the environment. In any verification approach, nondeterministic environment specification is key to obtain a correct answer about the behavior of the system.

When abstraction is used to simplify and speed the verification process on agiven model, it is just a simpler model of the design, usually specific to the property to be proved. Abstractions are almost always *lossy* – there is less information in the abstraction than there is in the original model of the design.

If we consider our picture of design as a set of equations and inequalities, then an abstraction is viewed as a simpler set of equations and inequalities, describing the same design.

Abstraction is critical – virtually every verification and synthesis tool relies upon some abstraction. However, abstractions improperly done are a road to disaster. In particular, the abstraction must remain faithful to the property being proved – the property must hold in the actual design if it is shown to hold in the abstracted design. An abstraction that is property-preserving is often said to be *conservative* or *homomorphic* with respect to the property.[1]

Here, we give some examples of homomorphic abstractions and their use.

4.1 The "Synchronous" Abstraction

The wave model of Section 3 is extraordinarily detailed – the behavior of every signal at every point in time is captured. This is far too detailed for most properties; we can not prove them over such a detailed model, and almost all of the information is irrelevant anyway. So we assume that the logic will evaluate *fast enough* that we can treat it as evaluating instantaneously, or atomically (the synchronous hypothesis of Section 3.4.1).

This is the central abstraction that underlies formal verification of systems and the fast functional simulation methods which are currently being introduced by the major CAD vendors, and the emulation methods which have been used for some time. Various vendors will impose further restrictions – on clocking methods, for example – but this is more an artifact of the limitations of their tools than any real requirement of the technologies. All that is required is that timing considerations may be neglected. Timing is ignored and the wave space over which the systems are defined is reduced to the space of finite-state machines. Mathematically, this is done by examining all the waveforms of the combinational logic at $t = \infty$. At this point, the waves are constant and Boolean, and may be abstracted as scalars.

Note that this abstraction is only valid when it can be shown that the logic in a design in fact evaluates *fast enough*; thus, this abstraction is conditionally homomorphic. A designer who buys a formal verification system, a cycle simulator, or an emulator, would be well advised to purchase a good timing verifier in the bargain.

4.2 The Functional Abstraction

Hardware emulation systems, most Formal Verification procedures, and new, highly-advanced prototype cycle simulators, actually go further than the synchronous abstraction. These tools really don't need to faithfully reproduce the logic between the latches; all they must do is reproduce the *logic function* of the logic between the latches. Throwing out the logic and replacing it with a more tractable representation is one of the most powerful techniques available to these advanced tools.

The most prominent replacement for the logic network in these functional abstractions is a data structure called the *BDD*, first devised by S. Akers and brought to high art by R. E. Bryant [5]. BDD's have the property that (up to variable order) they are a canonical form for logic functions – an important consideration for formal verification systems. More recently, it has been demonstrated that BDD's offer extremely fast evaluation of logic functions, making them an important tool for cycle simulation.

4.3 The Graph Abstraction

In contrast to the synchronous abstraction, the functional information of a network can be deleted; when this is done, one is left with a directed graph of nodes and edges, with weights on the edges – the weights represent the time required for a value to travel down the wire represented by the edge. This is the abstraction taken by first-generation timing analyzers (those on the market today). This abstraction is homomorphic – the outputs of a circuit will certainly stabilize by the time a graph-based timing analyzer says they will – but has been criticized as too conservative. The conservatism of this abstraction is manifested in the so-called "false path problem". Briefly, the implicit assumption in this abstraction is that an event can travel down any path in the circuit; however, inconsistent values on the inputs to the logic nodes on the path may make this impossible.

4.4 Quantification

An important abstraction in formal verification is *projection*, or *quantification.* Briefly, given a system described as a set of equations, and a property to be checked over that system, information about some variables is ignored and the property in question is examined. Physically, the variables in question are often simply deleted from the equations. It is possible to do this so that the abstraction in question is homomorphic – that is, that the property is preserved by the transformation.

An extreme example of this abstraction is the Graph Abstration, given above, where the logic value of *every* variable is forgotten. Similarly, the conservatism associated with the Graph Abstraction is simply the most visible example of the inherent conservatism associated with the quantification abstraction.

Pioneering work on abstraction was done by Clarke and his colleagues, and by others[12, 11, 10].

4.5 Granularity of Time

Though we often discuss time as a continuous variable, in fact it is quantized – at least so far as our mathematics can describe it. More precisely, given *any* extant model of timing behavior of digital systems, one can demonstrate that time is quantized and that the behavior of systems is constant except at integral boundaries of the quantum.

This does permit us to demonstrate a conservative abstraction of a system – we can increase the quantum of time. We ensure that this abstraction is homomorphic by modeling any signal that changes over a quantum as being an uncertain value over the quantum. This is conservative, since properties examined in a digital system must rely monotonically on the stability of signals. The synchronous abstraction, described above, may be viewed as the logical extension of this abstraction – the time quantum is raised to the level of a cycle.

4.6 The Power Abstraction

One interesting abstraction that has arisen of late is the *power* abstraction, which ignores details of function of a system and computes only its power consumption.

Exact power consumption is extraordinarily difficult to compute, since it involves perfect knowledge of the state of every signal in the system at every time. What is worse, *average* power consumption over millions of cycles is desired, since power supplies cycle in the millisecond range while computers operate in the nanosecond range.

The abstraction used is due to Brodersen and Chandrakasan [9]. Average activity on the I/O pins of a block is computed and multiplied by the capacitance and clock frequency, to obtain the power of a block.

4.7 Discussion

Abstraction is essential in a top-down design methodology that progresses through successive refinement of a specification [13]. In this process, refinement is the basic mechanism that maps one level of abstraction into another that is consistent with the previous one. Designers may apply refinement by "hand" and in this case, there should be a set of tools that guarantee that the "hand" refinement is indeed a refinement so that the verification

[1] Note that it is not required that if the property holds in the design then it must hold in the abstraction. While this is desirable, it is not essential – the worst that will happen is that the property will be "proved" false, and the designer will have to investigate this "false negative".

work done at early stages of the design still holds and that the various models of the design are consistent.

In order for the abstraction mechanism in simplifying the verification process to be valid, the abstractions must be homomorphic in nature – that is, they must preserve the property to be proved. Thus the nature of abstraction itself imposes verification obligations on the designer and toolset.

Some abstractions must be done manually. For example, as systems grow more complex, it will be infeasible to simulate an entire chip at the gate level, even with highly advanced cycle simulation capabilities. Designers will thus be forced to create abstracted models of their part, similar to a bus-functional model for a microprocessor, such that the simulation of the abstracted models together may be certified as a simulation of the whole chip. Abstraction is thus necessarily tied tightly to decomposition – the subject of the next section.

5 Decomposition

Decomposition is the process of breaking up a system design into components described at the same level of abstraction. The main goal of decomposition is to allow the design and verification of each component to be performed *almost* independent of each other. There are varying degrees of independence that can be achieved between the components depending on the design and verification problem at hand. Some of these are described below to highlight the differences and similarities in decompositions across problem instances.

5.1 Timing Analysis

Consider a synchronous sequential logic circuit where each memory element (or latch) is enabled by a single clock. Given a set of initial states on the latches and the maximum delay of each gate, the timing analysis problem is to determine the lowest clock period at which the circuit operates correctly. The problem may be decomposed into two: (1) Determine the set of reachable states in the circuit; (2) Find the latest time at which all the outputs of the combinational logic have reached their final stable value and at least one vector that causes this condition.

The first problem is solved using standard techniques for reachability analysis in formal verification[16] and logic synthesis [17]. Next, functional timing analysis is invoked using the complement of the reached set as don't cares [15]. Thus, the timing analysis problem on a sequential circuit is decomposed into a functional (timing independent) analysis of the sequential behavior followed by a timing dependent analysis of the combinational behavior.

5.2 Combinational Equivalence Checking

The problem of equality checking of Boolean functions can be solved by comparing the BDD's for the functions built using an initial multi-level description of the functions. An alternate scheme is to use a satisfiability or test generation program (based on classical branch and bound search techniques) to determine if the functions are different. Both techniques have limited applicability and the range of examples on which the techniques can be applied can be significantly increased by using a divide and conquer approach.

Suppose that functions $F(a, b, c, X, Y)$ and $G(a, b, c, P, Q)$ are to be compared - a, b and c are primary inputs and X, Y, P and Q are functions over a, b, and c. The approach in [4] attempts to replace the occurrence of the logic computing the sub-function X by that for P. This replacement is correct if no difference between the functions X and P can be observed at the output of F. This check is performed by first introducing an XOR gate between X and all its fanout - the other input of the XOR gate is connected to P. If the stuck-0 fault on the output on the XOR gate is untestable at output of F, P is a safe replacement for X. Repeating this process from inputs towards outputs leads to F having more and more of its circuit replaced by parts of G.

5.3 Finite State Machines

Most complex designs have compact representations when expressed as a set of intercommunicating processes. For example communication protocols and embedded systems are often expressed as a set of communicating FSMs. When properties are formally checked against this description, both language containment and model checking require to compose the FSMs into a single one. Since the composition has number of states bounded by the product of the number of states of the FSMs in the decomposed representation, we often have a state explosion problem.

Maintaining the representation compact is key to make formal verification applicable. An approach [10] is to automatically extract from each component machine only the behavior relevant to the verification of the given property through abstraction, and compose these extracted subcomponents to represent only the part of the behavior of the entire system needed to verify the property. This approach is powerful when the properties to be verified are *local* i.e., involve only variables that are in a single FSM. Decomposition then is accomplished when properties are either given in local form or they are decomposed into a set of local properties. In [2], an embedded control application to automotive electronics (a shock absorber automatic setting system) is expressed as a set of interacting FSMs and the properties to be verified are successively reformulated and decomposed until they are localized to allow verification in reasonable time. In this approach, the decomposition and abstraction mechanisms interplay to solve the complexity issues and are carried out manually. It is conceivable that in a not too distant future some of these actions could be carried out with the help of tools.

5.4 Cycle Simulation using Decision Diagrams

A recent approach to cycle-based logic simulation employs the use of decision diagrams to achieve orders of magnitude speedup in simulation speed. The approach consists of building the BDD's for the output and latch functions of a circuit, and performing simulation via a sequence of lookups on a tabular representation of the BDD's. One of the key optimization criteria that determines the simulation speed is the number of lookups needed to determine the output function

5.5 Discussion

The examples above illustrate that decompositions may be obtained manually or automatically. For example, the timing analysis problem is a one-time manual decomposition, whereas the decomposition for cycle based simulation is wholly automatic. In contrast, the automatic decomposition for combinational equivalence checking can be substantially aided by manual hints from the user [4, 8].

Depending on the problem being solved, decompositions may be exact or conservative. In the former case, a property holds on each component of the decomposition if and only if it holds on the complete system. In the latter case, the property is guaranteed to hold on the complete system if it holds on on each component of the decomposition, however, the converse need not be true.

6 Conclusion

We summarize the conclusions that naturally follow from our contention that the three concepts of formalization, abstraction and decomposition as cornerstones for a design methodology that could alleviate the problems facing the design community

when the implementation technology allows, and new applications require, ever increasing complexity.

- From the specification, the design process has to progress towards implementation by mapping the various levels of the design hierarchy into other formal models. The design process has to be characterized by the successive refinement concept, so that anything that has been proven or verified by any available tool is still true at all successive steps unless a major re-design is required in face of errors. Tools should be developed to help the designer to maintain consistency and to verify that refinement holds.
- The specification of a design must have clear semantics. When the underlying model is hardware, the HDL must have hardware semantics. Both Verilog and VHDL violate this with underlying event-queue semantics. It is imperative that the formalization process be *complete*, which implies that the specification should also explicitly include all constraints and properties required for the design, thus avoiding the lack of clarity and documentation that occurs in their absence.
- There must be support for abstraction in a specification language. Two critical abstractions that necessarily must be supported are the separation of function and time, and nondeterminism. Abstraction should be supported by tools that verify the consistency of the abstraction with respect to the original formalization. An important side-effect of the use of abstraction is the elimination of the *over-specification problem*, *i.e.* to avoid giving implementation details that may limit the quality of the design without reflecting accurately the initial goals and functionality requested of the design. Little or no support is provided by both Verilog and VHDL on this front.
- There must be support for decomposition beyond just what is allowed by existing HDL's such as breaking code into modules and support for ports lists. It is our contention that in order to fulfill the promise of synthesis and verification technologies, HDL's must incorporate structured forms of communication between modules that are easily and efficiently implemented in hardware. Only this step will lead to the ability to decompose design problems. Formal descriptions of the design should be modular, *i.e.* the behavior of each component of the design should not be affected by the behavior of any other module except through its interface.

If these paradigms (or others similar in concept) are followed, then all verification techniques can be used with maximum efficiency. In fact, formal verification is unthinkable without complete formalization and the use of abstraction to make it feasible on large designs. In addition, the problem of having multiple not formally correlated descriptions of the same design to be able to run different verification tools severely hampers the quality of verification.

It is clear that designers are searching for some breakthrough to cope with the problems they encounter: it is by no chance that panels and tutorials on formal verification are now common and that major companies are setting up groups to investigate new verification techniques. However, we strongly recommend not to focus on *techniques* only but to focus instead on a rigorous *design process*.

The future is dense with opportunities and dangers. A verification aware design methodology can be the light to avoid the pitfalls and to choose the right path to take.

References

[1] L. Augustin. An algebra of waveforms. *Technical Report, Computer Systems Laboratory, Stanford Unive rsity*, 1989.

[2] F. Balarin, L. Lavagno, H. Hsieh, A. Sangiovanni-Vincentelli, and A. Jurecska. Formal verification of embedded systems based on cfsm networks. In *Proceedings of the Design Automation Conference*, 1996.

[3] G. Berry, P. Couronné, and G. Gonthier. The synchronous approach to reactive and real-time systems. *IEEE Proceedings*, 79, September 1991.

[4] D. Brand. Verification of large synthesized designs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 534–537, 1993.

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

[6] R. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler. COSMOS: A compiled simulator for MOS circuits. In *Proceedings of the Design Automation Conference*, 1987.

[7] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogen eous systems. *Interntional Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[8] J. Burch. Techniques for verifying superscalar microprocessors. In *Proceedings of the Design Automation Conference*, 1996.

[9] A. Chandrakasan and R. Brodersen. *Low power digital cmos design*. Kluwer Academic Publishers, 1995.

[10] M. Chiodo, T. Shiple, A. Sangiovanni-Vincentelli, and R. Brayton. Automatic compositional minimization in ctl model checking. In *Proceedings of the International Conference on Computer-Aided Design*, 1992.

[11] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Proc. of Principles of Programming Languages*, 1992.

[12] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the International Symposium on Logic in Computer Science*, 1989.

[13] R. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1994.

[14] L. Lavagno, A. Sangiovanni-Vincentelli, and H. Hsieh. Models and algorithms for embedded system synthesis and validation. In G. De Micheli, editor, *Nato Advanced Study Institute*. Kluwer Academic Publisher, 1996.

[15] P. McGeer, A. Saldanha, R. Brayton, and A. Sangiovanni-Vincentelli. Delay models and exact timing analysis. In T. Sasao, editor, *Logic synthesis and optimization*. Kluwer Academic Publishers, 1993.

[16] K. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, 1993.

[17] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *Proceedings of the International Conference on Computer-Aided Design*, 1990.