

of another signal, or one set of signals was a decoded version of another set of signals. This redundancy had to be taken into account in the verification. Tracking down these redundant signals and determining their relation to other control signals was a time consuming process. This might have been avoided if the development process had begun using formal methods and the interfaces between modules been formally defined (for example, in temporal logic).

6 Conclusions

As we have seen, the use of formal methods has certain implications for the design process, mostly deriving from the need to control complexity. Below are a few suggestions for design practices that fit in with a formal approach to verification. It is interesting to note that these suggestions are good design practice regardless of whether formal methods are used:

- More effort should be concentrated on the early phases of a design. In particular, modeling should start in the conceptual phase of design. Nondeterminism in the models can be used to postpone design decisions. Abstract models should be verified to the extent possible before detailed models are made.
- Decompose systems into modules based on boundaries of least communication. Try to design modules in a way that requires little knowledge of the state of other modules. This will simplify the definitions of interfaces, and thus make it easier to decompose large verification problems into small verification problems. It also makes symbolic model checking techniques more effective, and means that design changes in one module have less impact on other modules.
- Precise definitions of interfaces are required for decomposing a large design into parts. An interface definition is simply an abstract model of the components as seen through the interface. Note that many designs require maintaining compatibility with existing chips, protocols, etc. Having formal models (*i.e.*, abstractions) of interfaces would make it easier determine whether a new product is compatible with an old one.

To summarize: some of the possible benefits of formal methods are more robust designs, early detection of errors, and ease of exploring the design space. We have observed that it is not necessary to have a complete proof of correctness with respect to a complete formal specification in order to obtain some of these benefits. In fact, very subtle errors can often be found automatically by checking general properties, like absence of deadlock.

Automatic property verification techniques, such as model checking, can be used either for very abstract global models of a system (as in the Gigamax verification), or for gate level models of small parts of a system (as in the queue module verification). These techniques are still limited mainly by the state explosion, and though progress in this area continues, we would not expect to see completely automatic verification of large systems at the gate level.

Gate level verification techniques (such as symbolic simulation) may be the first to have a significant impact on

design practice because they require the least expertise on the part of the designer. However, using verification techniques early in the design process can catch errors that are difficult to find at later stages. In addition, gate level verification can be made more difficult than necessary by a lack of formal definitions of the interfaces between modules. In the long run, the most effective approach would be one that ties together verification at every level of the design hierarchy.

References

- [1] D. Beatty. *A Methodology for Formal Hardware Verification, with Application to Microprocessors*. PhD thesis, Carnegie Mellon, 1993. tech. report CMU-CS-93-110.
- [2] W. R. Bevier, Jr. W. A. Hunt, J. S. Moore, and W. D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.
- [3] S. Bose and A. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *IEEE International Conference on Computer Design*, 1989.
- [4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
- [5] R. E. Bryant. Boolean analysis of MOS circuits. *IEE Trans. CAD*, 6(4):634–649, July 1987.
- [6] R. E. Bryant, D. Beatty, K. Brace, and K. Cho. Verifying a static RAM design by logic simulation. In J. Allen and F. T. Leighton, editors, *Proc. 5th MIT Conference on Advanced Research in VLSI*, pages 335–349. MIT Press, 1988.
- [7] Clarke, Emerson, and Sistla. Automatic verification of finite state systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 1986.
- [8] D. L. Dill and P. N. Lowenstein. Verification of a multiprocessor cache protocol using simulation relations. *Formal Methods in System Design*, 1(4):335–383, December 1992.
- [9] A. Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2/3):151–238, October 1992.
- [10] R. P. Kurshan. *Formal Verification of Coordinating Processes*. Princeton Univ. Press, 1994. to appear.
- [11] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [12] K.L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *International Symposium on Shared Memory Multiprocessors*, 1991.
- [13] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Lecture Notes in Computer Science*, volume 224, pages 510–584. Springer-Verlag, 1986.
- [14] J. M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.

plementation were not covered by formal verification in any way. Nonetheless, several significant errors in the protocol were discovered by the model checker and corrected. These errors would have been much more difficult to detect and diagnose either in simulation or in testing the actual hardware. One of these errors involved a complex sequence of messages leading to a deadlocked state, where two “read” messages crossed a link in opposite directions, thus blocking each other, since each held a lock on the bus at one end. Such a counterexample, produced by the model checker, is much more easily understood than the corresponding error occurring in a detailed simulation trace.

Because the model checking process required only ten minutes to an hour to run, design changes could be easily incorporated and then tested to see if they introduced new errors. Thus, the initial effort of constructing the abstract model was amortized as the process of finding and fixing errors progressed. One reason for the efficiency of the model checking was the use of a technique called *symbolic model checking*, that uses ordered binary decision diagrams to implicitly represent the state graph of the model. This technique is particularly well suited to systems of processes with a limited communications structure, such as the bus structure of the Gigamax protocol. For protocols with other communications structures, the symbolic model checking technique might be much less efficient, making the time for one model checking cycle unacceptably long. In fact, for the Gigamax, models with two cache blocks instead of one required an unacceptably long time to check (over a week). In general, however, one of the strengths of the model checking technique is the ability to quickly change and reverify a model.

5.2 Gate Level Verification

Now we consider a case of model checking at the gate level – specifically, verification of submodules of a packet router chip. This chip was described in a synthesizable subset of the Verilog hardware description language, which was translated to a gate level netlist by synthesis tools. At this level of detail, it was not possible to apply model checking techniques to the chip as a whole. Instead, the formal verification concentrated on critical submodules of the chip, while simulation work proceeded in parallel, testing configurations of several router chips embedded in system level models.

The most interesting of the modules was one that maintained queues of packet buffers using linked list structures. This module used complex data structures, as well as significant parallelism and pipelining. The module contained register files holding pointers to the head and tail of each queue, as well as a register file holding a forward link for each packet buffer, pointing to the next buffer in the queue. The enqueueing and dequeueing logic allowed packets to be dequeued from two queues simultaneously, while one packet was being enqueued. Both these operations were pipelined in several stages, the pipelines being complicated somewhat by the limited number of ports of the register files, which prevented some accesses from happening simultaneously. Even though the queue manager was only one submodule of the router chip, with roughly 1400 register bits it was signif-

icantly larger than could be handled by a straightforward model checking approach.

The approach taken to the verification was to use “representation functions” for the pipeline, as described above, mapping the state of the pipelines onto an idealized un-pipelined state. The allowed transitions of this idealized state were specified by temporal logic formulas, and these formulas were verified using SMV (although for this purpose, symbolic simulation could also have been used, had the necessary tools been available). In addition to these formulas, several “representation invariants” had to be proved. These were used to rule out certain illegal states of the pipe that do not correspond to any idealized state.

Model checking revealed a number of fairly subtle bugs in the design. These were typically problems that would occur when two or three particular operations were inserted in the pipe in particular temporal relations to each other. In general these problems had not been found in the simulations because enough packets of a particular type had not been generated to cause required coincidence of traffic to occur in the pipeline. In perhaps a greater number of cases, the counterexamples found by the model checker were not in fact bugs because they were ruled out by logic outside the queue management module, or even by software. For example, a sequencer outside the queue manager ensured that packets could not be dequeued from the same queue on consecutive cycles. This fact was used in optimizing the bypass logic of the pipelines. Such conditions had to be incorporated into the specifications of the queue module as assumptions. The chip designer was the final arbiter as to what was a bug and what was not. One advantage of model checker counterexamples over simulation output was that the former tended to point more clearly to the source of the problem. Model checking counterexamples consisted of only a few steps, while the simulation output was lengthy, making it difficult to find the source of errors resulting many clock cycles later in incorrectly delivered packets.

It was necessary to make several optimizations by hand in order to make the model checking run in an acceptable amount of time. First, the specification was broken down by hand into properties that were simple enough for the model checker to verify. Second, state variables were identified by hand that were not relevant to the properties being proved, and these were abstracted away. Third, the ordered binary decision diagrams were optimized by hand (this involved adjusting the order of the state variables of the model as they appear in the binary decision diagrams). Thus, changes to the design could not be as easily accommodated as they were in the Gigamax verification, since they might require modifying the properties to be proved, the abstractions, or the variable ordering. Most changes were small and required little or no change to rerun the verification, however in some cases significant effort was required to accommodate design changes.

One fact that significantly hampered the verification effort was the lack of modularity in the design. The control logic was distributed in a complex way among a number of modules in the chip, resulting in a very large number of control signals passing between modules. Many of these signals were redundant, in the sense that one signal was a delayed version

ton. This translation might be done automatically, or the semantics itself might be embedded in the logic as a function, and used in the proofs. In any event, proving that the implementation model is equivalent to (or implies) the specification model requires constructing a detailed proof. Some steps in the proof may be filled in automatically by the theorem prover, but in general, construction of the proof is an interactive process that requires a great deal of expertise on the part of the user, about the particular design, proof techniques, and also the particular capabilities of the interactive prover.

One advantage of theorem provers is that they allow you to reason about a system design before certain parameters have been fixed (for example, the number of bits in a word, or words in a packets, or processors on a common bus). These parameters can simply be replaced by symbolic variables in the model, something that isn't possible with model checkers, whose variables must have fixed finite ranges. Theorem provers also don't suffer from the state explosion problem, since they do not need to examine every possible state of the model. However, it is still necessary in practice to use some form of abstraction to control complexity, especially in systems with significant concurrency.

Another advantage of theorem provers is that they allow you to reason at much more abstract levels [2]. For example, using the Boyer-Moore system, it was proved that a simple microprocessor design implements its instruction set architecture. Then a simple programming language was defined, with an interpreter function and a compiler function, and it was proved that compiling a program into the machine language and executing it is equivalent to interpreting it using the interpreter function. Using the interpreter function, one can then prove properties of programs written in the programming language, *etc.* This kind of top-to-bottom verification is possible only with a general purpose theorem prover.

Theorem provers have been used to verify quite a number of simple microprocessors. Generally, this is done by establishing a *simulation relation* between the implementation and the architecture, in much the same way as one would with symbolic simulation or model checking. The difference is that the simulation relation is established by logical proof, rather than by exhaustive analysis. The designs verified by theorem provers tend to have little concurrency or asynchrony (*i.e.*, there is little use of pipelining, asynchronous interrupts, caches, coprocessors, *etc.*). One notable exception to this is a cache consistency protocol derived using the HOL theorem prover [8]. This was done by establishing simulation relations between models at several levels of abstraction. Since model checking deals well with control and concurrency without the need for detailed proofs, it may be that a combination of methods will be most effective for high level verification. For low level verification, other methods are probably more suitable, since they require less user interaction.

5 Verification in practice

We now consider two case studies using formal verification in different stages of the design process. In both cases formal

verification was done in parallel with standard verification methods (*i.e.*, HDL modeling and simulation), in a commercial development environment. The first is the verification of a cache consistency protocol for a distributed multiprocessor. This protocol was modeled at an abstract behavioral level and verified using model checking. In the second case, model checking was used at the gate level to verify submodules of a packet router chip. We will be concerned mainly with whether or not the verification process was able to generate useful and timely information for the designers, and how the design and verification processes interacted, rather than with details of the systems and the properties verified.

5.1 Behavioral level verification

Our first case study is the verification of the cache consistency protocols of the Gigamax, a distributed multiprocessor developed at Encore Computer Corporation [12]. Formal verification was motivated by the complexity of the protocols – it was considered that using simulation alone could not provide the desired level of confidence in the design. The Gigamax system is structured as a hierarchy, with several “local” buses, each using a bus snooping coherence protocol and connected to a “global” bus via a bus bridge. The bus bridge snoops transactions on the bus at one end, and when appropriate, sends messages to the other end, which are queued and eventually issued as bus transactions. The bus bridge contains cache tags for all cache blocks from local memory that are present in remote caches, allowing it to send invalidate and call-back messages when appropriate.

The protocol was modeled and verified using the SMV (*symbolic model verifier*) system, a model checker that uses ordered binary decision diagrams to reduce the state explosion problem [11]. Nonetheless, it was necessary to use a very abstract model of the protocol in order to make the model checking process tractable. This model had only one cache block, with state information about other addresses eliminated. Operations on the cache block, such as loads, stores and replacements were completely nondeterministic. This eliminated any model of the processor or address trace generation. Similarly, the latency of message delivery was made completely nondeterministic in order to avoid modeling details of message transmission and queueing. Pipelining of the bus protocols was ignored.

The specifications were formulated in temporal logic. The main specifications given were that the protocol does not deadlock (*i.e.*, reach a state where some cache is locked out from read or write access), that accesses to the cache line are sequentially consistent, and that the diagnostic system not flag an error while the protocol is running correctly. For the sequential consistency specification, another abstraction was used – the data in the cache line were modeled by a single bit representing a time stamp value. This bit was checked to make sure no processor observed time stamps out of order.

The modeling approach had a number of disadvantages. First, the model was extracted by hand from state transition tables and block diagrams, with some interaction from the designers. This left open a possibility of error. Second, the verified behavioral model was not carried forward to the gate level design stage. Thus, errors introduced in the im-

MOS circuits, taking into account pass transistor effects, like precharge, and charge sharing [5]. The boolean functions obtained at the outputs of a circuit can be compared against the desired functions, and if they are not equivalent, a specific input vector can be produced for which the circuit output differs from the specification. Thus, symbolic simulation differs from ordinary logic simulation in that the use of symbolic variables at the inputs allows one symbolic input vector to cover many logical input vectors. COSMOS uses very efficient algorithms based on a boolean representation called *ordered binary decision diagrams* to compare boolean functions [4]. Nonetheless, comparing boolean functions is a hard problem in general, and so some means is often necessary to reduce the complexity of the problem. One way to do this is to introduce the unknown value X for initial states and inputs that are not relevant to the property being proved. This kind of abstraction was used, for example, to verify a static RAM chip [6]. In this case, informal arguments were used to show that the set of symbolic simulation vectors completely covered all possible required behaviors of the chip.

Symbolic simulation falls in the category of property verification, since each symbolic vector is essentially one property that partially specifies the desired behavior. It applies mainly to gate level and switch level design, the last stages in our design process. Symbolic simulation tools can be applied, however, to proving specification/implementation equivalence (albeit at a fairly low level). For example, suppose we are given a description of a microarchitecture, and would like to show that a pipelined implementation is equivalent to it. This can be done by providing a *representation function*. This function takes the state of the pipeline and translates it into an equivalent microarchitecture state. This is usually the state that would result if the pipe were emptied by inserting no-ops. By symbolic simulation, we can check that one step of the pipelined version is equivalent to one step of the microarchitecture, in terms of the representation function. This has been done, for example, to show the equivalence of a pipelined stack circuit to an idealized stack circuit [3]. A more elaborate technique has been used to compare a transistor level model of a simple microcoded microprocessor to its instruction set architecture [1]. Whether such comparisons can be made between more complex processors (having, for example, internal caches, TLB's, deep pipelining, *etc.*) is unclear, since the symbolic simulation method relies heavily on intracable boolean comparison methods, and the representation functions for such systems may be very complex.

4.2 Model checking

Model checking is another method of property verification that is somewhat more abstract than symbolic simulation. In this case, the properties to be verified are written in *temporal logic* [13]. This is a logical notation that can concisely represent temporal relationships between events. For example, one can easily state in temporal logic properties like

- a and b never happen at the same time
- whenever a happens, b eventually happens

- a is always possible in the future

The first property is a *safety* property, stating that something bad never happens. The second is a *liveness* property, stating that something good eventually happens, and the third is a *modal* property, stating that something good *may* eventually happen. The latter kind of property is useful, for example, to prove that a protocol does not reach a deadlocked state. Specifications of temporal properties may also be described using finite automata instead of temporal logic. In either case, a *model checker* can translate the implementation model into a finite state graph, and then check automatically that the specification is satisfied [7]. In addition, if the specification is not satisfied, the model checker can produce a *counterexample* – a sequence of events that violates the specification. Counterexamples are perhaps the most valuable product of model checking, since they help locate the source of errors. The design can then be corrected and the model checking process repeated. In general, model checking is most useful for verifying control aspects of a design, especially for complex protocols between many concurrent finite state machines.

The main limitation of model checking is that it requires building a complete state graph for the model. In general, the number of states in this graph increases exponentially with the size of the model. This *state explosion* means that it is very important to verify desired properties at the appropriate level of the abstraction hierarchy. It is often necessary to make certain abstractions that are specific to a given property being proved. One verification tool that provides substantial support for abstraction is the COSPAN system [10]. In this system, the models are finite automata, and the abstractions are *homomorphisms*, or functions that map the behaviors of detailed models into the behaviors of more abstract models in the hierarchy. COSPAN can check the validity of these abstractions, as well as checking temporal properties that are specified as finite automata.

As we will see, model checking can be applied early in the design process, at the behavioral level, or later at the gate level.

4.3 Theorem proving

The most general and powerful methods of verification are based on general purpose theorem provers. A theorem prover (or more properly a theorem checker) is based on a logic – a formal language for stating mathematical propositions. A logic is equipped with a proof system – a set of axioms and inference rules that make it possible to reason in a step-by-step manner from premises to conclusions. Depending on how powerful the logic is, the proof system may or may not be complete (meaning that all valid propositions in the logic can be proved). Most theorem provers are interactive, requiring guidance from the user in order to generate proofs.

A design description can be imported into the theorem prover in several ways. It might be embedded directly in the logic – for example, each gate might be represented by some proposition in the logic. Alternatively, a hardware description language might be given a semantics that translates each program into a mathematical object, like an automa-

2 Verification paradigms

There are two basic paradigms for formal verification. The first uses two models of a system: The specification represents the desired behavior. The implementation is more operational in nature, and is closer in structure to the actual device. For example, a specification might be the “programmer’s manual” of a processor, which models the effect of executing a single instruction on memory and registers that are visible to the programmer. The implementation would include the complete register state of the processor, including any temporary registers, microcode sequencing, pipeline registers, *etc.* The verification task would be to prove that the specification and implementation are equivalent in a suitable sense. This notion of equivalence would depend on what state information is considered visible to the programmer, and might entail a change of time scale.

The second verification paradigm is sometimes called “property verification”. The specification in this case consists of a set of properties that are to be proved about the implementation model. These properties need not be a complete specification of the desired behavior of the system. Instead, they tend to be rather general properties, for example that a given protocol does not deadlock, or that it guarantees mutual exclusion or fair access to some resource. In a sense, the property verification approach assumes that the implementation is functionally correct for the typical cases, and that the goal is to discover the infrequently occurring “corner cases” that result in deadlocks, access conflicts, *etc.* Often, the exact property being checked turns out to be unimportant – once an error occurs, the system becomes either locked up or highly chaotic, hence the error can be found by checking almost any nontrivial specification. This approach has been very effective in discovering control errors in highly concurrent systems.

The two styles of verification are not mutually exclusive – in fact they are somewhat complementary. The property verification approach has the disadvantage that it is not possible to know when enough properties have been specified. Thus, there is the danger of underspecification. On the other hand, specifying equivalence to an abstract model is also somewhat dangerous in that a high level model often specifies unintended behavior for cases that were not considered by the designer. Complete specifications tend to be prone to error themselves (errors which are easily propagated to implementations), while simple abstract properties are more easily understood. Thus, it can be useful to verify important properties of both the “specification” and the “implementation”, as well as verifying the equivalence of the two.

3 Abstraction

The most important issue in formal verification (and perhaps design in general) is the control of complexity. For this purpose, the essential tool is abstraction. Generally, abstraction means eliminating detailed information in the model so as to make it tractable to analyze. There are various kinds of information that may be eliminated by abstraction. For example, details of timing or sequencing can be eliminated by compressing several time steps of a detailed model into

one time step of an abstract model. Abstraction can also involve hiding internal details of a module or a data structure. In property verification, abstraction is typically used to suppress aspects of the system state that are not relevant to the particular property being verified. These abstractions have been called “unbalanced” in the sense that they concentrate on the state of some components of a system, while ignoring others.

Abstraction is most useful when done in stages. One begins with a very abstract model, and proceeds through a succession of increasingly detailed models. As the level of detail is increased, components of the system are decomposed into subcomponents. This produces a hierarchy of models, in which large system components are modeled at a very abstract level, while smaller subcomponents are modeled in greater detail. Each model serves as the specification for the set of components immediately below it in the hierarchy. The verifier’s job is to prove an appropriate relationship between each abstraction level and the level below. This job is made easier by the fact that each individual model in the hierarchy is simple, even though the system as a whole may be quite complex.

Most verification methods rely to some extent on abstraction to reduce complexity. The differences are essentially in the kinds of relationships that are specified between levels of abstraction. The stages of the design process mentioned above, from concept to gates, represent particular levels of abstraction. These abstractions, however, are not necessarily sufficient for formal verification purposes. Generally, formal verification requires much more careful thought about how to structure the abstraction hierarchy and what information to hide at each level so as to make the verification process tractable. This may require considerable expertise, and thus is one area that may be an obstacle to integrating formal methods into the design process.

4 Methods and tools

In this section, we consider some of the available tools and formalisms for verification, starting with those methods that are closest to current practice in terms of the skills required, and moving up to more powerful methods requiring greater expertise. It also happens that we move from methods that are mainly applicable in later, more detailed stages of the design process, to more abstract methods applicable early in the design process. This brief survey does not even begin to cover the variety of formal methods that have been studied for hardware verification. For more complete surveys, see [9, 14].

4.1 Symbolic simulation

Of the various formal verification methods, *symbolic simulation* is closest in spirit to current engineering practice. In a symbolic simulation of a circuit, initial values and inputs are given not as boolean values (0 or 1), but instead as symbolic variables. At each simulation step, the simulator computes the values of signals as boolean functions of these variables, rather than as definite boolean values. The COSMOS symbolic simulator can do this for switch level

Fitting Formal Methods into the Design Cycle

K. L. McMillan
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

This tutorial introduces several methods of formal hardware verification that could potentially have a practical impact on the design process. The measure of success in integrating these methods into a design methodology is arguably not the ability to provide formal guarantees of correctness, but rather to detect design errors in a timely manner, as the design evolves. Based on this criterion, and some limited practical experience, we consider where the various methods might fit into the life cycle of a design, what their capabilities and shortcomings are, and how the design process might change in order to accommodate formal methods.

1 Introduction

The term *formal verification* in system design generally means using methods of mathematical proof rather than simulation and testing to insure the quality of a design. There are several possible benefits of taking a formal approach over simulation. The primary benefit is that formal verification covers all possible behaviors of a model, whereas simulation can miss important behaviors. Thus, formal verification can improve the robustness of a design. Development time can also be reduced using formal methods since bugs are caught in earlier stages of the design process. In addition, formal mathematical specifications are useful in themselves as tools for communication between designers, especially for defining interfaces and protocols. There are several barriers, however, to obtaining these benefits in practical design. For less automated verification methods, the chief obstacle is the considerable expertise in methods of mathematical logic and theorem proving that is required to develop the necessary proofs. More automated methods exist that do not require the generation of detailed proofs, but these methods tend to be limited to small systems. For these reasons and others, including a lack of robust tools, it has been difficult to integrate formal methods into commercial design practice in a significant way.

It appears in practice, however, that considerable benefit can be obtained from formal methods without necessarily

applying them to all aspects of a design, or generating a complete formal proof of correctness relative to a complete specification. In fact, formal verification can be applied at various levels of detail and on various scales in a system design. This makes it possible to find and correct design errors before detailed design is completed. In addition, some automated verification tools can provide feedback about the cause of errors, making it easier to find the flaws in a design. The practical benefits of using formal methods will more likely be in the ability to find and correct errors quickly, while a design is evolving, rather than to provide formal guarantees of correctness once the design is complete.

In this article, we will consider the various stages of the design process, and how the tools and methods of formal verification fit into these design stages. The emphasis will be on methods that appear to have the most promise for practical application in the near future. Our model of the design process will assume the use of current methods based on hardware description languages, simulation and gate level synthesis. The design process will be assumed to proceed in several roughly defined phases, from conceptual design, through abstract behavioral modeling, register transfer modeling, and synthesis of a gate level net list. We will survey some of the available methods and tools for formal verification, and see how they can be used to advantage in various stages of this process. Two anecdotal examples of the application of formal verification tools in commercial design projects will be used as illustrations. Some observations will be drawn from these experiences about the strengths and weaknesses of the methods, and about design practices that might facilitate a formal approach to verification.

One of the first observations to be made about the design process is that designs evolve very rapidly, especially in the early stages, because of requirements changes, performance tradeoffs, added features, *etc.* Thus, one important test for formal methods in practice is how quickly they can adapt to design changes, and provide feedback to designers about the impact of these changes. If the verification process can keep up with the design process, then verification can allow more rapid and extensive exploration of the design space than would otherwise be possible. The ability to absorb design changes quickly is one argument for more automated methods over less automated (but more powerful) methods.

Another observation about formal methods and the design process is that using formal methods, much more time will be spent on the earlier stages of the process than using simulation alone. The tradeoff for this should be less time spent in the later stages simulating and debugging.