Computer-aided verification

Theorem proving and model checking are powerful tools that can
verify the logical correctness of today's ICs or find their hidden bugs

THE BUG IN THE FLOATING-POINT DIVISION circuitry of Intel Corp.'s Pentium chip brought notoriety to the huge cost that may be incurred when a logical bug is committed to silicon. (For the Pentium, some have put it at US $500 million.) But behind every such bug that makes the news, an uncounted number of logical errors go undetected throughout the design, implementation, and marketing phases of many products. Either these errors will finally be detected and corrected in hardware, software, and subsequent releases—or they will simply remain as bugs, known to a few, experienced users.

The debugging of most hardware circuits and many highly complex software systems is done by an elaborate testing process involving extensive runs of computer models of the target implementations. When a circuit or program is synthesized from a simulation model of this nature, any errors in the model may appear in the resulting implementation. Finding and correcting such errors before synthesis can reduce correction costs by an order of magnitude.

Reliance on running simulation models alone becomes a weakness in designing and developing large-scale, highly complex systems. An example is the control system for asynchronous environments, with its many coordinating components, such as communication protocols and telephone switches, as well as circuits that execute many instructions in parallel. In systems of this kind, the set of possible behaviors so greatly outstrips the set that can be simulated that even billions of tests, run continuously for a year or more, can never examine most combinations of behaviors. Among those unexamined behaviors may lurk critical untested combinations or, worse yet, unknown system failure modes.

Thus, random testing, once considered an important tool for uncovering faults in unanticipated behaviors, is now viewed widely as inadequate. On the other hand, tests designed for specific scenarios leave unexplored possible combinations of behavior that fall outside the anticipated patterns.

Such difficulties have spurred research into methods that attempt to prove a system correct, in the same sense that a mathematical theorem is proved correct. The ideal algorithm would automatically analyze a system model and either conclude it was correct or reveal a bug within a reasonable number of steps. From a mathematical perspective, though, only very simple systems are suited to such decision procedures. Although such automated theorem provers had some successes, they were generally unable to pinpoint errors in incorrect designs and were difficult for a nonexpert to operate effectively.

EDMUND M. CLARKE
Carnegie Mellon University
&
ROBERT P. KURSHAN
Bell Laboratories

One common observation of hardware and software designs holds that in the life cycle of a development project, the design mostly is incorrect and approaches correctness only (hopefully!) at the end of the design and testing process, when the "last" errors have been detected and eliminated. Thus, a formal method that sometimes can prove a design correct, but not easily locate flaws when the design is incorrect, is more useful at the very end of the design and test process. Furthermore, exigencies of the marketplace demand that design verification fit unobtrusively within the development process and, at any rate, not delay that process any more than the current practice of simulation testing.

For these reasons researchers sought design verification methodologies that were more limited in scope than the theorem-provers. They also wanted techniques that could be highly automated and could locate bugs in faulty designs as readily as they could prove error-free designs correct. Attention at first focused on finite-state models, which can assume only a finite number of distinct configurations during any arbitrarily long or even endless execution. Although limited in a mathematical sense, finite-state models necessarily encompass every digital circuit and every software system implemented on a digital computer. Their computational feasibility for systems with a finite but staggeringly large number of states, however, is a key concern.

Today, the first computer-aided verification tools are becoming commercially available. They are based on methods that in many cases can reduce the complexity of verification (without sacrificing
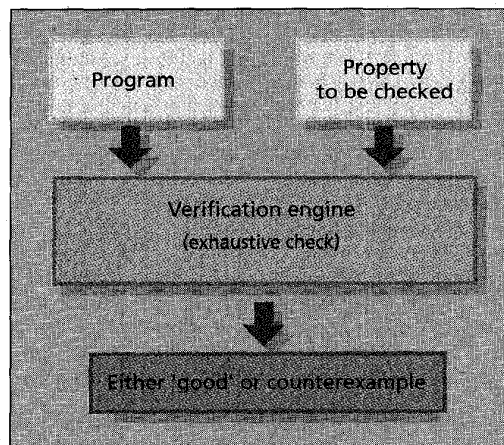
The verification process begins with a program (or a description of a circuit design) and a list of properties to be checked. The verification tool determines if the properties are true, or if not, provides a counterexample.

 61

guaranteed correctness) to such a degree that it becomes computationally feasible. Among the most powerful of these methods are symbolic model-checking and homomorphic reduction, both of which represent a complex system in terms of a compact and computationally more tractable structure. Moreover, the two can be used together with a multiplicative reduction effect, since they work independently of one another. Of special importance is the fact that they each can be implemented automatically, so the task of reduction is programmed into the computer rather than presenting a burden to the design engineer.

As early as 1976, Amir Pnueli, a researcher at the Weizmann Institute, Rehovot, Israel, and other computer scientists began proposing deductive systems, notably temporal logics, that could verify finite models. These logics support a syntax in which it is simple to express such notions as "If a process requests a bus, then eventually it gets to access it." This approach suffered several practical drawbacks, however. Since temporal logics are most useful for describing requirements placed on simple causal relation-

ships, designers really could not be expected to define an entire system model in temporal logic. (Even logicians sometimes are unsure of the effective meaning of complex temporal formulas.) Moreover, although the decision procedures were guaranteed to give an answer, the computational complexity of the required checks grew exponentially with the size of the formulas. For these reasons, interest in this approach was largely academic.

The first practical breakthrough came independently in 1980 from Edmund Clarke and his student Allen Emerson at

Harvard University and from Joseph Sifakis and his student Jean-Pierre Queille in Grenoble, France. Each team modeled a system as a computer program that could be checked automatically. The check determined whether the program satisfied particular specifications (properties) defined by the programmer and expressed in temporal logic.

Dubbed model-checking by the Harvard pair, the approach had the great advantage of producing counterexamples through which programmers could locate errors. Computationally, model-checking appeared promising, especially since under reasonable restrictions its complexity grows linearly with model and formula size. This promise was at best elusive, though, since model size itself generally grows exponentially with the number of variables that define the system, a phenomenon often termed state-space explosion.

Since the problem of state-space explosion is intractable in the worst case, research on model-checking has focused on heuristics or methods that circumvent the complexity barrier in special cases. Some heuristics have simply formalized techniques already used in simulation: abstracting inessential parts of the model and exploiting the model's hierarchical structure and symmetries.

Verification at AT&T

In 1985 Bell Laboratories began work on a finite-state verification tool called Cospan. Its function was to check the behaviors (input/output sequences) of one program to see if they were contained in the behaviors of another program or, alternatively, if they were consistent with a given property. This paradigm supports a top-down development methodology based upon stepwise refinement.: the user starts with an abstract design, debugs and verifies that design, and then refines the abstract design by adding more detail.

The refined design comprises the second refinement level. Refinement is repeated through a succession of levels, such that all properties true of the previous level of abstraction are inherited by the next level. At each subsequent level of refinement, new properties are checked that are not relevant at higher levels of abstraction. The process continues until details of the ultimate implementation are included. From this lowest level design, hardware is synthesized or C-code is generated (or a combination of these two), automatically implementing the low level design.

Through stepwise refinement, bugs can be detected and corrected earlier in the design cycle than previously possible, thereby accelerating the design process. But stepwise refinement is not automated: the user must design each refinement step (the tool then checks to make sure that it is consistent with the previous design level). Alternatively, the tool can be applied to a single level of the design, as in FormalCheck, a tool developed by Bell Labs Design Automation that uses Cospan as its verification engine. The tool embeds Cospan in a graphical user-interface that facilitates its use and links it to the VHDL and Verilog languages, so that users need not learn Cospan's native language.

Cospan runs homomorphic reduction algorithms, both symbolically (using binary decision diagrams) and explicitly, to cope with the enormous computational complexity in typical designs. Although FormalCheck's core contains the algorithms necessary for stepwise refinement, this tool does not currently support them. Likewise, Cospan contains other more experimental algorithms not supported in FormalCheck, including timing verification and software bridges to theorem-provers, as well as to two other finite-state verification systems: the Symbolic Model Verifier (SMV) and SPIN.

The SPIN software verification system, developed by Gerard J. Holzmann at Bell Labs in 1989, is based upon an interleaving model of concurrency, in which, unlike with hardware, only one component of the system state is allowed to change at a time. This restriction, which is popular for software models, makes the verification run faster than synchronous models (the kind used with Cospan or SMV), where any number of components can change at a time. It runs faster because each state update is a simpler operation, being restricted to one component only.

Moreover, the interleaving semantics supports a reduction algorithm, developed by Doron Peled at Bell Labs, that exploits symmetries in the order of execution (partial order reduction), which is not feasible in tools like Cospan or SMV. SPIN also incorporates the "Supertrace" algorithm that facilitates a very memory-efficient partial search of a state space.

Although FormalCheck is proprietary to Bell Labs, the Cospan and SPIN tools are available at no charge. Cospan may be licensed by universities for noncommercial research and educational use and SPIN is available by anonymous ftp (inquire of k@research. bell-labs.com and gerard@research.bell-labs.com, respectively). SMV is available through anonymous ftp from Carnegie Mellon University (inquire of Edmund.Clarke@cs.cmu.edu).   —R.K.

In 1987, one of us (Kurshan) unified and formalized these techniques into a single paradigm called homomorphic reduction. (It was named for a mathematical structure-reducing homomorphism— "shape-" preserving function—on the Boolean algebra of formulas in program variables.) The paradigm defines associations among system events that need not be distinguished for a particular verification task. For example, to verify some property, it may be unnecessary to distinguish the non-zero values of some variable V. So the homomorphism may replace V with a new variable, the values of which are zero and non-zero.

The verification program, possibly with initial aid from the user, makes a guess about system values that need not be distinguished. The verification algorithm then checks that this guess (or one of its own) is correct. If so, the abstracted version is used for model-checking; if not, this "localization reduction" algorithm

Debugging a cache flush unit

CARL PIXLEY

Model-checking is used at Motorola Inc. to design and debug some commercial products. The company's internal model-checking tool, Verdict (produced by Bernard Plessier and the author), is based upon the Symbolic Model Verifier tool—thus using computation tree logic (CTL)— and the Verilog hardware description language.

In one model-checking success, a byte data link controller (BDLC)—a serial/parallel bus interface module that connects an automotive serial bus to a microprocessor—was designed in Verilog and verified with Verdict [see figure]. Implemented in 6000 or so transistors, including some 150 latches, the BDLC was first simulated on a few test cases to eliminate obvious errors. The preliminary simulation is important because model-checking designs with gross errors tends to be time-consuming.

One unit within the BDLC is the symbol submodule, which monitors the serial bus and, from the duration of signal (at high or low voltage) it receives and from its own state, decides what symbol (such as logic 0, logic 1, or start-of-message) is being sent. Characters from the symbol module are received by the byte submodule, which forms them into one-byte messages.

During the design of the BDLC, Motorola's verification team performed a series of model-checking experiments based on abstractions of the submodules. For example, in model-checking the byte module, the team coupled it with a manually abstracted symbol module. Whereas in the real design, the symbol module changes state according to the activity on the bus, the abstract symbol module has no bus interface, only states. For instance, when the abstract symbol module is in the "bus is idle" state, then, on each clock cycle, it nondeterministically transitions to the "bus is active state" or remains in the "bus is idle" state.

This activity generalizes the behavior of the real symbol module, whose state determines what signals are communicated to the byte module. Properties specific to the byte module were then model-checked with the simplified abstract symbol module. After these individual modules were checked and if necessary redesigned, the team was able to put all of the modules together with no abstractions and to model-check some properties of the whole design. Consequently, the first test chip for this module passed all functional tests.

The most remarkable feature about model-checking is its ability to generate sequences of events that demonstrate exactly how a property can fail. Early in the design of the BDLC, for instance, Verdict found a design error that was indicated by the failure of a property, and produced a specific sequence of events that demonstrated the failure of the property. The sequence lasted 208 clock cycles—the shortest sequence that will cause the prop-

erty to fail. It seems unlikely that a designer would think of exactly the sequence of events that would cause this property to fail.

In another successful use of Verdict, the team subjected a small module, call it CF, of a commercial microprocessor to model-checking after the design was complete. The CF unit consisted of logic gates and 35 flip-flops (that is, state-holding elements) that implement a protocol. The unit was thought by its designers to have no errors because it had been extensively simulated.

The CF unit is a one-hot implementation of a nine-state state machine. The term means that there are nine specific flip-flops, each of which represents one of the nine states. The CF unit is "in" state s when flip-flop s has value 1. Since the design should clearly be in one, and only one, state at any time, an obvious property to check is whether every state reachable from a valid reset state has exactly one of the nine flip-flops asserted.

Serial bus
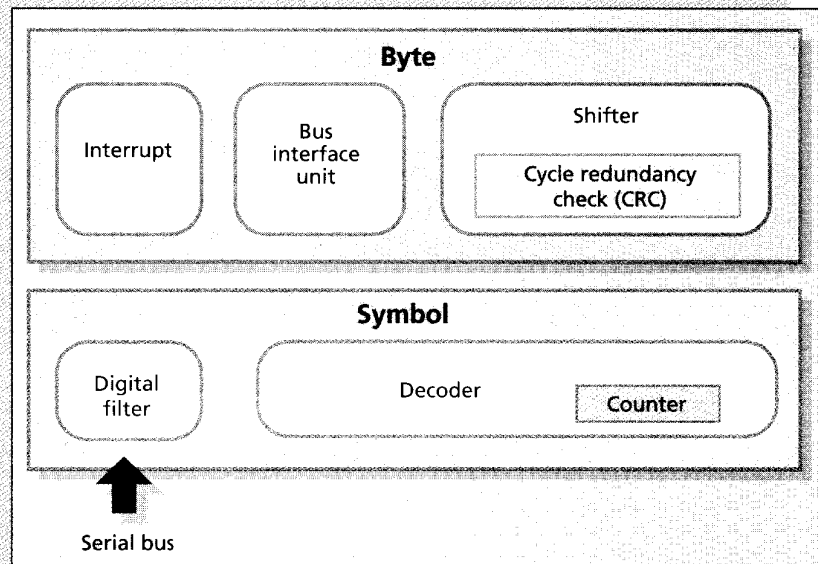
The one-hot property failed immediately for the original CF unit. After the design was repaired, the same properties were run on the revised design and the team discovered that the design could get into another invalid state in a different way.

So two real "bugs" were discovered by a single set of properties. This is an example in which model-checking a simple, intuitive property exposed two errors in a relatively small design that was thought to have no errors. Of course, many other properties were checked as well. Since statements of individual specifications expressed in computational tree logic tend to be only a few lines long, the set of those properties checked becomes a concise document of the design, and the specification can be used on similar designs in the future.

Carl Pixley is senior scientist for Motorola Inc., Austin, Texas. His interests are model-checking, sequential equivalence, and retiming and resettability.

automatically adjusts the reduced model and verification is tried again.

Replacing the given verification problem with a simpler one in such a correctness-preserving fashion was shown to be formally equivalent to finding a Boolean algebra homomorphism that preserves some—but not all, or there would be no reduction—of the structure of the corresponding program. Such homomorphisms include mapping complex data structures and control sequences into sim-

Debugging a communications chip

MASAHIRO FUJITA

During field tests, engineers at Fujitsu Ltd. in Japan observed that a complex communications chip slated for commercial applications behaved abnormally several seconds after power-up: some data was duplicated while other data disappeared entirely. The IC was designed for high-speed switching operations at 156 MHz and had 111 000 or so gates (32 000 for random logic and 79 000 for RAM). Initially, extensive simulation had been used to validate the circuit.

Because the abnormal behavior in the tests occurred only after several seconds of operation, hundreds of millions of simulation cycles would be necessary to reproduce it. Consequently, simulation was impractical as a debugging technique. Even if it were possible to generate so many test cycles (by emulators, for example), it would almost certainly be impossible to analyze the enormous amount of information produced in enough detail to find the error.

Since obtaining an error-free version of the IC had high priority, the author, along with Ben Chen, a Fujitsu computer-aided design engineer familiar with formal verification, and their colleagues turned to the Symbolic Model Verifier (SMV) model-checker to debug the design.

Information about the chip design was available only in gate-level circuit descriptions. Since the circuit had more than 100 000 gates, SMV could not be directly applied to the gate-level circuit description. Chen decided to exploit the modular structure of the circuit to reduce the state explosion problem. He started with a first-in, first-out buffer (FIFO) that was believed to be the most likely source of the error. Designers were not confident of the behavior of specially designed cache registers used in the FIFO to speed up the operation of the chip. After checking many properties with SMV, Chen was unable to find any errors in the FIFO. He concluded that the error must be in some other part of the circuit.

Eventually, Chen determined that all the major components of the circuit were correct, meaning that the error must be caused by the way in which they were connected. He built a concise model for the chip in the SMV language. The model did not describe the hardware in the chip that was clearly unrelated to the cause of the error; and the width of the data path was reduced as much as possible to cut the number of states.

Chen's specifications were written in temporal logic that described the abnormal behavior. Running on a workstation, the SMV model-checker took less than half an hour to determine that the error arose from a condition in which the same address appeared twice in the FIFO. This condition—the result of incorrect resetting of address recycling circuits—caused the data to be sent twice, the second time overwriting the first, leading to duplication or disappearance of the data, depending upon the instant that the data was read.

The abnormal execution trace found by the model-checker was more than 50 clock cycles long. This meant that, starting in the initial state, it took at least 50 clock cycles for the error to occur. Since the width of the data path in the actual chip was larger and the input data tended to be more random, the error occurred only after a considerable period of time, which explained why it was so hard to find.

RAF, WAF = read/write first-in, first-out buffer
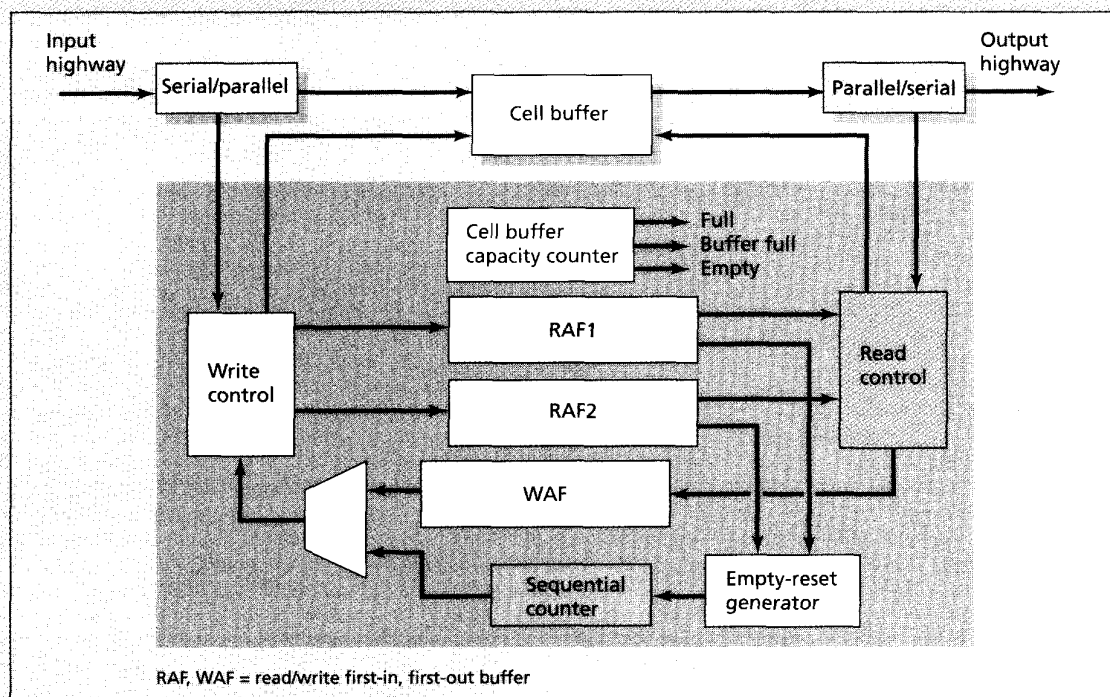
pler ones, which retain enough information for the verification task at hand.

This mapping subsumes data abstraction and symmetry reduction. Through data abstraction, a notion introduced by Pierre Wolper at the University of Liege, Belgium, data can be reduced to just a few distinct values. Through symmetry reduction, a model can be replaced by a "quotient" model that factors out symmetric distinctions. In many cases, homomorphic reduction gives designers a chance to verify arbitrarily large models.

Around the same time, in the late '80s, several other groups independently discovered an alternative—and in fact complementary—form of reduction. Called symbolic model-checking, this approach analyzes sets of states, represented by Boolean formulas, as opposed to individual states. For example, if x is a variable of the system, then the expression "x equals 0" can be understood as the Boolean formula that defines the set of all vectors of values of all the system variables in which x equals 0— a very large set of vectors. This potential for succinct expression of a large set of system values can be exploited computationally during model-checking.

The formulas are stored in a compressed form of binary decision tree called a binary decision diagram (BDD). To understand how those diagrams impact model-checking, it is necessary to see how model-checking itself works. Suppose in using a program or a hardware description, a property required of the program is expressed as a temporal logic formula. For example, the formula may express the property: "If the program ever sets a variable V to 1, then eventually it sets V back to 0." The role of verification (or in this case model-checking) is to determine whether the formula is true for the given program or, in logical terms, whether the executions of the program form a model of the formula. When the program uses only a finite amount of memory, it may be viewed as a finite state machine. In this case, the logical model is the set of all its input/state/output sequences.

The model is constructed by a search that begins with the initial state of the program. From there, every possible succession of state transitions of the program is generated, starting with all possible single transitions. Many transitions are possible from a given state, since each immediately following state depends upon external inputs to the program. Moreover, if the program incorporates parallel processing or asynchrony, several "immediately following" events may be scheduled from a given state, and each of these must be explored.

Every immediately following state that is possible but which has not been previously generated, is placed in a pool of states to be expanded in the same fashion.

The step is repeated until no new states are found, defining a breadth-first search of the model state space. Eventually, the search must terminate since the state space is assumed to be finite, and when the pool of states becomes empty, the model is complete.

Every possible execution of the program thus is represented in the model by a sequence of consecutive states. Model-checking then consists of determining whether every such sequence satisfies the given property, and if not, of finding a counterexample sequence [see figure]. This, sequence, too, is accomplished using techniques of search.

Consider the example of resetting the variable V to 0 in the model just described. To check this, all the states where V is 0 must first be marked. Then, looking backward, all states that must reach a marked state in one step must also be marked. This procedure is repeated until it reaches a "fixed point," from which no new states can be marked. Now all the states from which V must eventually set to 0 are known. If any state where V is 1 is not marked, then the formula is false.

The symbolic solution

The catch is that even very small programs can have a huge number of states. For example, a program that can store a mere 250 bits of data has at least 2^{250} states— more states than there are particles in the universe! When the expanded model becomes too large to store in available memory, the model-checking technique can no longer be applied directly. This is where symbolic techniques enter the picture.

A symbolic model-checker represents the model indirectly, using a Boolean function as above, to determine when a transition is possible from one state to another. A Boolean function takes on the values 0 and 1 and thus may be used to encode the values of all the system variables in terms of the system inputs. This function is put into a unique form (usually the binary decision diagram form) to make it easier to manipulate during analysis.

When carrying out a breadth-first search, the set of marked states is also represented symbolically through a Boolean function. The ability to perform the operations of Boolean algebra on these expressions allows the search to be carried out entirely using the symbolic forms. As a result, checking formulas depends not on the number of states of the model, but on the compactness of the symbolic forms.

Symbolic techniques are not the ultimate solution to the state explosion problem, since there is no guarantee that the symbolic representation will be any smaller than the explicitly constructed model. Nonetheless, a representation can be cho-

sen to exploit the structure inherent in the state space of the program. As a result, we can verify a model many orders of magnitude larger than any it is possible to construct explicitly. Just how to do this has been the subject of much recent research.

Hurdling the complexity barrier

Currently, the most powerful finite-state verification techniques integrate symbolic model-checking and homomorphic reduction. But computational complexity remains a barrier for some cases. One tactic would focus verification efforts upon any models that are susceptible to the known heuristics. The challenge then becomes how to determine in advance which models have this property.

There have been some successes in this direction. Modular programming techniques, which limit the amount of information allowed past module boundaries, have played an important role in advancing symbolic model-checking and homomorphic reduction. Even apart from verification, modular techniques have already been identified as useful in managing large programs. So there is reason to hope that the best current programming practice and verifiability of programs may converge to a common ground.

Another tactic to promote verifiability, perhaps of more immediate use in an industrial setting, lets the model and available resources guide the verification effort. This technique rests upon an increasingly common view that verification is more valuable in proving a model incorrect (and providing a counterexample to assist in debugging) than in proving it correct.

After all, systems can fail in many ways, some entirely beyond the reach of verification. All verification efforts proceed from assumptions about the environment of the model to be verified. If these assumptions are incorrect, then a faulty system model may be "verified." Moreover, if the synthesis procedure is not itself verified, then a correct design may yield a faulty implementation. Intel explained that the floating-point division bug occurred because designers used a faulty script to implement the Pentium's division table. While the table's design was probably correct, the script produced a hardware version that omitted a few essential values. If a verification procedure failed to examine this script or its result, then even if the table and all associated parts of the algorithm for floating-point division were correct, the result still would have been faulty silicon.

Assuming that verification will never embrace all the ways in which a system can fail, perhaps verification should be so applied to a project as to extend the most benefit possible for the given resources of time and staff. This is especially true in a

Debugging VHDL code: the HDTV example

ARTHUR B. GLASER

The ISO standard MPEG2 main profile decoder chip is a microprocessor-controlled decoder for the program bit stream in a digital television receiver. A component within the chip is the compressed data interface controller (CDIC), which is programmable by the microprocessor to accept either bit-serial or byte parallel MPEG2 compliant input data. It detects the start and end of frames, performs start code alignment and synchronization between the incoming data rate and the internal processor rate, and buffers data, which it ultimately stores in external dynamic RAM. The control logic for the CDIC contains about 2500 gates; and contributing a lot to the computational complexity of verification is an on-chip register file. It is described by about 2000 lines of VHDL code, and contains 1500 latches, typical of the class of circuits that can be expected to be verified fully and automatically.

At Bell Laboratories, engineers used the verification system FormalCheck to determine if the internal first-in, first-out buffers (FIFOs) of the CDIC could overflow under the control protocol. In fact, the analysis showed that there was a condition under which the request to write data to the external DRAM was inhibited, permitting the internal FIFO to overflow.

Because of the complexity of the CDIC model, this analysis would not have been feasible except for a built-in reduction algorithm that automatically produced a reduced model of the CDIC with this overflow error. The starting point for the reduc-

tion algorithm was a simple user-provided "seed": a specification of some major components of the CDIC that probably would be necessary for the analysis, and other components, mainly some of the memory in the internal FIFOs, that could probably be excluded. The reduction algorithm iterates from this seed, attempting to perform its analysis in a logically conservative (correctness-preserving) fashion on a much smaller model than the given one. (The algorithm automatically adjusts the model used for analysis as necessary, and ensures that any reported error is an error in the original unreduced model.)

After only about 90 seconds on a Sparc LX workstation from Sun Microsystems, Mountain View, Calif., the verification tool detected the error in the CDIC algorithm. After the error was corrected, the Bell Labs team repeated the analysis and the verification tool showed that the FIFO now never could overflow. This proof, fully automatic but for the user-provided seed, required the tool to search only 2.5 million states and analyze about 9.8 million transition conditions in the reduced model (a significant reduction over the unreduced model, which has an estimated 10^{30} states). Once the tool found the bug, it produced an error track, which had to be interpreted. In this case, the error track spanned 2000 clock cycles because the buffer had to be filled before the error could be detected.

Arthur B. Glaser is a distinguished member of the technical staff in the Bell Labs Design Automation Center of Lucent Technologies Inc., Murray Hill, N.J. His current interest is the development and application of model-checking tools for formal verification.
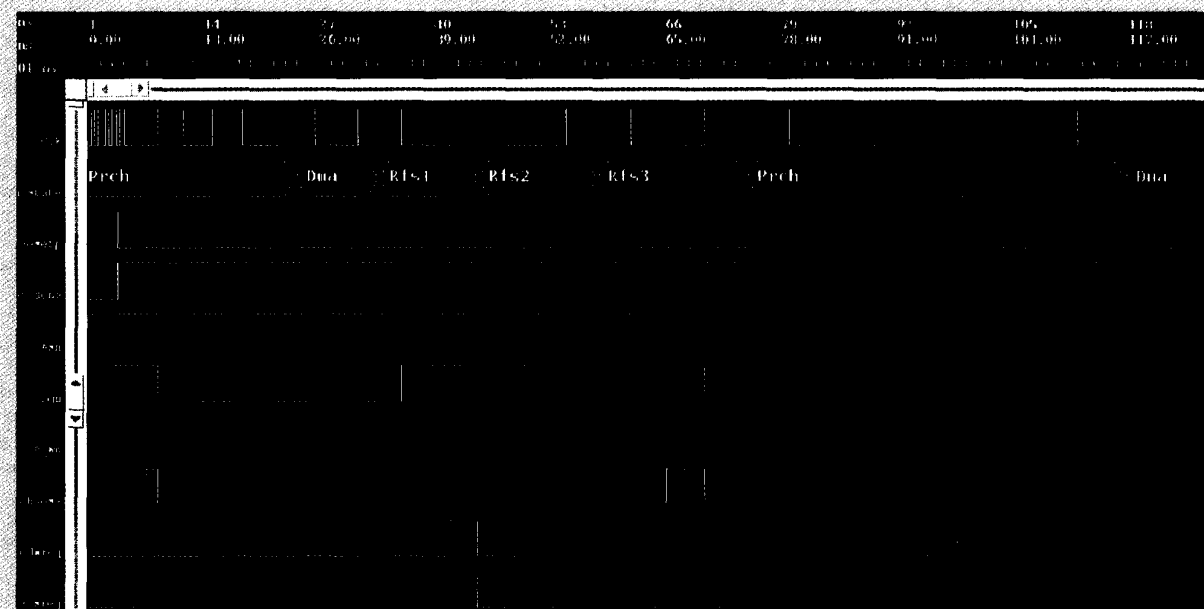
commercial setting. All known reduction algorithms may be applied to a given model in an automated fashion until either a bug is found, or the model is verified, or the space or time allocated runs out. Bell Laboratories uses this approach with its verification tool FormalCheck.

As confidence in and reliance upon finite-state verification grow, designers will slowly learn to use the process in more focused and advantageous ways.

Sometimes the fear is that merely learning how to integrate formal verification techniques into the design process may slow development unacceptably, or that the process itself may result in less efficient circuits. In fact, verification may detect errors earlier in the design cycle, thereby actually speeding up the overall project. In many modes of use, formal verification has no effect upon the form and efficiency of the ultimate design. But even when formal

verification leads to design compromises, some performance degradation may be worth the price, especially in view of ever faster circuitry, since a more reliable design is being brought faster to market.

Still, some data-intensive algorithms such as arithmetic and logic units may remain beyond the scope of purely automatic finite-state methods. Although automated theorem-provers may not face the same limitations, their enormous user-

Verifying cache coherence protocols

KEN MCMILLAN

The shared-memory multiprocessor architecture is becoming prevalent in high-end servers designed to handle many users or large parallel computations. In such a machine, several parallel processors share a common store. To avoid communications bottlenecks, each processor has its own local cache memory that stores recently accessed data from the shared memory. One of the machine's most complex parts is its cache coherence protocol, a system of messages implemented in hardware, by which the processors ensure that their local caches are consistent. Simulation of such systems is particularly unreliable because of their highly asynchronous nature: the exact time at which a given processor will access a given memory location and the exact time delay of messages are unpredictable. As a result, many "race conditions" must be considered in the design and test of the protocol.

For example, suppose processors A and B are both trying to modify the contents of memory location V. Typically, both would send a message to the "home" location of V, asking for an "exclusive" copy of V. If the message from A arrives first, the "home" will dispatch an exclusive copy of V to processor A. When the request from B arrives, it might simply be forwarded to A. The exclusive copy of V and the request from B are now in a "race" toward processor A, and the designer must consider what would happen if their order of arrival was reversed, so that A receives a request for V before V actually arrives.

Such "races" can be extremely complex in a real system, sometimes involving 10 or 20 messages. So it is difficult for the protocol designer to anticipate and correctly handle them. Even in the best-designed simulations, the more complex races occur rarely, so a simulation methodology cannot guarantee that all have been tested. But since the protocols are finite-state, they are natural candidates for computer-aided verification.

The first company to use computer-aided verification on a cache coherence protocol was Encore Computer Corp., Fort Lauderdale, Fla. In collaboration with researchers at Carnegie Mellon University, Encore used this tool for its Gigamax system—a shared-memory machine with a hierarchically structured cache protocol. To create an abstract model of the protocol, the Symbolic Model Verifier (SMV) system was used. Although many details of communication in the machine were left out, the protocol itself (the system of rules for sending messages) was modeled in its entirety.

After an appropriate set of specifications in computation tree logic was formulated, the use of binary decision diagrams in SMV as an indirect representation of the state space allowed these properties to be checked in a few minutes, despite the large number of states the model could reach. One particularly important error found by model-checking was a deadlocked state resulting from an unanticipated race condition. In all, 13 messages participated in the shortest scenario producing this deadlock, making it extremely unlikely that the condition would occur in a simulation run. The short turnaround time of the model-checking process allowed the fix for this error to be quickly checked for its impact on correctness.

More recently, model-checking combined with abstraction found an error in the cache coherence protocol for the proposed Futurebus+ standard. These achievements demonstrate the value of using exhaustive verification of abstract models to prevent high-level errors from propagating into detailed designs where they are much more costly to find and correct.

Kenneth McMillan is the creator of the Symbolic Model Verifier. He is a research scientist at Cadence Berkeley Laboratories, Berkeley, Calif., and works on efficient computational methods for formal verification and computer system design for verifiability.

---

overhead probably would make them impractical on a Pentium-sized project.

But a ray of hope is emerging on the research front: recent hybrid methods of verification integrate finite-state model-checking with automated theorem-proving. Using the two approaches in concert, engineers currently are working on techniques that they hope will one day be used to develop an entire complex microprocessor in considerably less time and more reliably than currently possible. At the same time, new forms of binary decision diagrams have evolved that can make the symbolic manipulation of arithmetic expressions computationally feasible.

Applications of finite-state computer-aided verification are many and varied. AT&T, Bell Laboratories, Cadence, IBM, Intel, and Motorola all have burgeoning internal verification programs. [For some specific examples, see p. 62, p. 63, p. 64, p. 66, and p. 67]. At Bell Labs, the first commercial product developed using verification was some software for a specialized network data protocol, verified by Gerard J. Holzmann in 1984. Currently, at least four commercial tools for finite-state verification of hardware have been released or announced: FormalCheck from Bell Laboratories (of Lucent Technologies) and Viewlogic's CheckOff, under license from Abstract Hardware Ltd., are general-purpose model-checkers. Chrysalis' Design VERIFYer and Compass Technology's VFormal are equivalence checkers, which verify that a perturbed model is logically equivalent to an original model that is believed to be error-free. ●

To probe further

Edmund M. Clarke, Allan Emerson, and Prasad Sistla, described their first working model-checker in: "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications" (TOPLAS, Vol. 8, no. 2, pp. 244–263). It contains basic definitions and a simple example. Robert P. Kurshan's *Computer-Aided Verification of Coordinating Processes* (Princeton University Press, 1994) explains the semantic model and the reduction methodology of Cospan.

Gerard J. Holzmann's *Design and Validation of Computer Protocols* (Prentice Hall, 1991) describes the software verification system, SPIN. Kenneth L. McMillan's *Symbolic Model Checking* (Kluwer, 1993) contains a good description of the Symbolic Model Verifier model-checking system. The standard reference on symbolic model-checking is the 1994 paper by Jerry Burch, Edmund M. Clarke, David Long, Kenneth L. McMillan, and David L. Dill titled "Symbolic Model Checking for Sequential Circuit Verification" (*IEEE Transactions on Computer Aided Design*, Vol. 13, no. 4, pp. 401–24).

For a description of current technical activity in formal verification, see the Web site of the DIMACS Special Year on Logic and Algorithms: http://dimacs.rutgers.edu/SpecialYears/1995 _1996/index.html.

About the authors

Edmund M. Clarke is professor of computer science at Carnegie Mellon University in Pittsburgh. In 1995 he became the first recipient of the FORE Systems Professorship, an endowed chair in the School of Computer Science. He is editor-in-chief of *Formal Methods in Systems Design* and is on the steering committees of two international conferences.

Robert P. Kurshan is a distinguished member of the technical staff at Bell Laboratories, Murray Hill, N.J. He and his colleagues designed and built the Cospan verification system, which is used in the commercial verification tool, FormalCheck.

Spectrum editor: Linda Geppert