Computer-Aided Verification

E. M. Clarke
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

R. P. Kurshan
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

February 17, 1996

ABSTRACT

Highly complex coordination among components of present-day integrated circuits has led to numerous logical failures committed to silicon. Although some of these failures have become notorious, most are just quietly fixed, often at great unbudgeted expense. This problem has spurred research in how to check such systems for logical correctness.

This will appear as an invited article in the IEEE Spectrum Magazine.

Computer-Aided Verification

E. M. Clarke
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

R. P. Kurshan
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

The Intel Pentium FDIV bug provides a notorious example of the huge cost (some have estimated $500 million) that may be incurred when a logical bug is committed to silicon. However, behind every such bug that makes the news, an uncounted number of logical errors remain undetected throughout the design, implementation, and marketing phases of many products. These errors finally are detected and corrected in hardware, software, subsequent releases — or simply remain as "bugs," known to a few, experienced users. While the cost of these bugs has never been calculated seriously, it is generally held to be quite significant. This expense, as well as the less tangible ones of damaged market reputation associated with unreliable product, potential liability in safety-critical and security-critical applications, and the overriding interest in having circuitry behave correctly, all have spurred research in methods to check the logical correctness of integrated systems.

Most hardware circuits (and many advanced or highly complex software systems) are debugged through an elaborate testing process that involves extensive runs of computer models of the target implementations. When a circuit (or program) is synthesized from such a simulation model, any errors in the model may appear in the resulting implementation. Detecting (and correcting) such errors prior to the synthesis step can save "an order of magnitude" in correction costs.

The problem with relying exclusively upon simulation becomes evident in designing and developing large-scale, highly complex systems, which push simulation to the breaking point. These applications include control systems for asynchronous environments with many coordinating components, such as communication protocols and telephone switches, and circuits that achieve speed by executing many instructions in parallel. In these systems the set of possible behaviors so greatly outstrips the set which can be simulated that even billions of tests, run continuously for a year or more, necessarily fail to examine most combinations of behaviors.

Among the unexamined behaviors may hide important untested combinations and, worst of all, unknown system failure modes. Thus random testing, once considered an important tool for uncovering faults in unanticipated behaviors, now is 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.

These difficulties have led researchers to formulate methods that attempt to *prove* a system correct, in the same sense that one proves a mathematical theorem. Ideally, one would like a *algorithm* (a method that can be implemented as a computer program) that automatically analyzes a system model and either concludes it correct or reveals a bug — all within a finite and, in fact, computationally feasible number of steps.

From a mathematical perspective, only very simple systems can have such decision procedures. Nonetheless, many researchers contended that mathematical proofs held promise for overcoming the problems associated with simulation. By 1960 they were building automated *theorem-provers*, which could conclude that *some* assertions about *certain* models were true. In other cases, these algorithms simply would never terminate. The principal drawbacks of the automated theorem-provers, however, were their general inability to pinpoint errors in incorrect designs, and the difficulty for a non-expert to operate an automated theorem-prover effectively.

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, could be highly automated, and also could locate bugs in faulty designs with the same facility that they could prove error-free designs correct. Attention focused on *finite-state* models: ones that can assume only a finite number of distinct configurations during any arbitrarily long (or even nonterminating) execution. Although surely limited in a mathematical sense, finite-state models necessarily encompass every digital circuit and every software system implemented on a digital computer. The issue of computational feasibility, for systems with a finite but staggeringly large number of states, however, is a fundamental issue to which we will return.

Today, the first computer-aided verification tools are becoming commercially available. These are based on methods which in many cases can reduce the complexity of verification (without sacrificing guaranteed correctness) to a point at which it is computationally feasible. Among the most powerful of these methods are *symbolic model checking* and *homomorphic reduction*. Both of these serve to represent a complex system in terms of a compact and computationally more tractable structure. Moreover, the two methods can be used together with a multiplicative reduction effect, since they work completely independently (othogonally) to one another. Of great importance, 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 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 notions such as "if a process requests a bus, then eventually it gets to access it". In Linear Time Temporal Logic, if req denotes a bus request and acc denotes obtaining access, the logical expression is $req \Rightarrow F acc$ (where $\Rightarrow$ means "implies" and F means "eventually" ("finally"). Although originally proposed as a convenient way to study the evolution in time of the discrete control events in a computer program, it soon became understood that even more significant was the *finite state* character of these logics. The reason this was significant was that the correctness of all assertions expressible within the logic could be decided by a single algorithm or "decision procedure" based upon an exhaustive search of a related finite state model. Hence, it was possible in principle always to determine whether or not a system model, described by one logical formula, satisfied a "correctness" property described by another.

However, this approach suffered several practical drawbacks itself. Temporal logics are most useful for describing requirements on simple causal relationships, as above. For complex sequential interactions, constructs such as circuit diagrams or finite state machines are found to be easier to write and understand. It was not realistic to expect a designer 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 major practical breakthrough came independently in 1980 from Edmund Clarke and his student Allen Emerson at Harvard, and from Joseph Sifakis and his student Jean-Pierre Queille in Grenoble. The two teams proposed similar approaches. Each modelled a system as a computer program which 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 Clarke and Emerson (a natural coinage that has achieved universal recognition), this approach offered the all-important 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. However, this promise at best was elusive, since model size itself generally grows exponentially with the number of variables that define the system, a phenomenon often termed "state-space explosion."

Subsequent research on model-checking has emphasized methods for handling state-space explosion. Since the problem is intractable in the worst case, work has focused on heuristics or methods that circumvent the complexity barrier in special cases. Some heuristics simply have formalized techniques already used in simulation: abstracting unessential parts of the model and exploiting the model's hierarchical structure and symmetries. In 1987 Robert Kurshan unified and formalized these techniques into a single paradigm called *homomorphic reduction*. Named for a mathematical structure-reducing homomorphism ("shape-" preserving function) on the Boolean algebra of formulas in program variables, this paradigm is capable of working both in a *static* and a *dynamic* mode. In the statc mode, parts of the system irrelevant to a property under check are removed prior to model-checking. For example, if a part of the system model has no computational effect on the parts being checked, it surely can be eliminated. In the dynamic mode (which may be preceded by a static reduction), the computer program (possibly with aid from the user) makes a static guess about system values which need not be distinguished. The verification algorithm then checks (dynamically) that this this guess is correct. If it is correct, the abstracted version formed by associating values which need not be distinguished is used for model-checking; if it is incorrect, the reduced model is adjusted and tried again. An instance of this is the notion of *data abstraction* introduced by Pierre Wolper, where data values may be associated when their distinct values are irrelevant to the the property being checked. For example, in order to verify that a message-passing communications protocol delivers every message, it may be enough to test the protocol for only *two* messages: one representing a generic message being tracked and the other abstracting all the other messages in the system. (For this reduction to be valid, the system must treat all messages symmetrically— which most often is the case.)

Replacing the given verification problem with a computationally simpler verification problem in such a correctness-preserving fashion was shown to be formally equivalent to finding a Boolean algebra homomorphism which preserves some — but not all, or there would be no reduction — of the structure of the corresponding program. Examples of such homomorphisms include mapping complex data structures and control sequences into simpler ones, which retain sufficient information for the verification task at hand. This subsumes *data abstraction*, through which data can be reduced to just a few distinct values, *symmetry reduction*, through which a model can be replaced by a "quotient" model that factors out symmetric distinctions, and *localization* which focuses model-checking upon just that portion of the model needed to verify a given property. Homomorphic reduction gives designers the possibility (in many cases) to verify arbitrarily large models.

Around the same time 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 apposed to individual states. For example, if x is a variable of the system, then the formula "$x = 0$" can be understood to define the set of all vectors of values of all the system variables in which $x = 0$, a very large set of vectors. This potential for succinctly expressing a large set of system values can be exploited computationally during model-checking. These formulas are stored in a compressed form of binary decision tree called a *binary decision diagram* (BDD).

To understand how BDD's impact model-checking, we first must understand better how model-checking itself works. Suppose we have a program (or a hardware description), and we have expressed a property required of the program, 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 "model-checking" in this case 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, beginning with the initial state of the program. From the initial state, every possible succession of state transitions of the program is generated, starting with all possible single transitions. There may be many transitions 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, many different "immediately following" events may be scheduled from a given state, and each of these must be explored. Each possible immediately following state which has not been previously generated, is placed in a pool of states to be expanded in the same fashion. This step is repeated until no new states are found, defining a *breadth-first search* of the model state space. 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, to find a counterexample sequence. This too is accomplished using techniques of search. Consider the example of resetting the variable V to 0. To check this, we first mark all the states where V is 0. Then looking backward, we mark all states that must reach a marked state in one step. Marking these, we then repeat this procedure until it reaches a "fixed point", where no new states can be marked. We now have all the states from which V must eventually set to 0. If any state where V is one is not marked, then the formula is false.

This is how a model-checking algorithm can check a program for a given property and either conclude that property holds for in the program, or else produce a counterexample demonstrating a case where the property fails (*cf.* Figure 1).

Unfortunately, this is only the beginning of the story. The catch is that even very small programs can have a very large number of states. For example, a program which can store merely 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, we no longer can apply the model checking technique 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

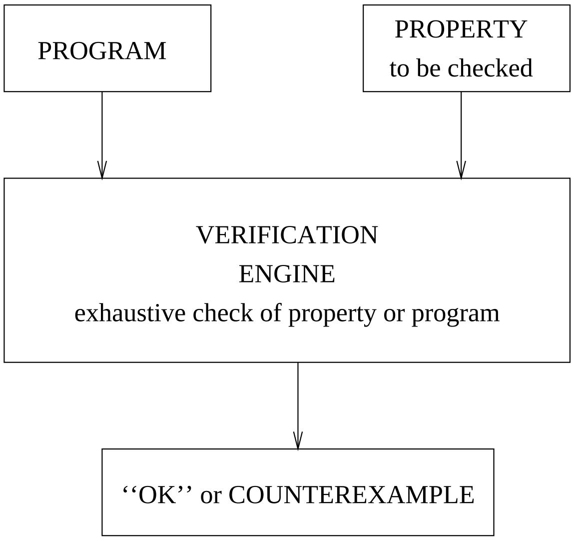


Figure 1: The control flow of finite-state model-checking.

all the system variables in terms of the system inputs. This function is put into a unique form (usually the BDD form) to make it easier to manipulate during analysis. When carrying out a breadth-first search, the set of marked states also is represented symbolically, through a Boolean function. The ability to perform efficiently operations of Boolean algebra on these expressions allows the search to be carried out entirely using the symbolic forms. As a result, the ability to check 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. However, if the representation is well chosen to exploit the structure inherent in the state space of the program, we can verify a model that is many orders of magnitude larger than we would be able to construct explicitly. Just how to do this is somewhat technical and has been the subject of much recent research.

Currently, the most powerful finite-state verification techniques integrate symbolic model-checking and homomorphic reduction, as in AT&T's *FormalCheck* verification system based on the research tool *COSPAN*. (FormalCheck and COSPAN also incorporate explicit state enumeration for those cases which involve arithmetically complex calculations for which explicit analysis is computationally simpler than symbolic analysis.) However, even these advances are insufficient to analyze some models, and computational complexity remains a barrier for these cases. One possible tactic would focus verification efforts upon those models susceptible to the known heuristics. The challenge then becomes how to determine in advance which models have this property. There has been some success in this direction. Modular programming techniques which limit the amount of information which must cross module boundaries, have played an important role in advancing the success both of symbolic model-checking and homomorphic reduction. Since modular techniques, independent of verification, already have been identified as useful to programmers for managing large programs, there may be 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 more immediately practical in an industrial setting, lets the model and available resources guide the verification effort. This technique rests upon an increasingly commonly held 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 aegis of verification. For example, 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 the synthesis procedure is not itself verified, then a correct design may yield a faulty implementation. For example, in Intel's case, designers apparently 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, then even if the table and all associated parts of the FDIV algorithm were correct, the result still would have been faulty silicon.

If there always are ways that a system can fail that are outside the aegis of verification, then perhaps verification should be applied in a way that extends the most benefit to the project for the given resources of time and staff-power. This especially is true in a commercial setting. One may apply, in an automated fashion, all known reduction algorithms to a given model, until either a bug is found, the model is verified, or space or allocated time run out. AT&T MicroElectronics uses this approach with their verification tool FormalCheck. Run innocuously in the background, FormalCheck will find bugs in incorrect designs, and verify the correctness of designs which are bug-free. In some cases, of course, the analysis may run out of memory or allocated time, before it is complete. Requiring little from users, either in the way of knowledge of the verification process or setup time, FormalCheck's low overhead makes designers more forgiving if it generates no useful data. If every few runs reveal previously unknown bugs, the tool pays for itself. When it eliminates some errors before simulation could detect them, or with less effort, the payoff can become significant. If verification finds an occasional error that would altogether elude simulation (and researchers and practitioner have many such examples of this), then the technology represents a major win.

As interest and confidence in and reliance upon finite-state verification grow, designers gradually will learn to use it in more focused and advantageous ways. The more thought the designer puts into applying verification methods, the more benefit is realized.

Sometimes one hears concerns 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 and thus offers a possibility actually to accelerate the overall project. In many modes of use, formal verification has no effect upon the form and efficiency of the ultimate design. On the other hand, increasing interest is becoming focused upon structured and behavioral techniques that may trade some implementation performance for accelerated time to market and increased reliability. Not so long ago, there were major objections to implementation programs written in C, since better performance was obtainable by writing in assembler. Priorities do change. Even when formal verification leads to some design compromises, with ever-faster process technology, bringing a more reliable design more quickly to market — even at the price of some performance degradation — may be preferable.

Some data-intensive algorithms such as arithmetic logic units may remain beyond the scope of purely automatic finite-state methods. While automated theorem-provers may not face the same limitations, their enormous user-overhead likely would make them impractical on a Pentium-size project. One ray of hope is emerging on the research front: recent hybrid verification methodologies integrate finite-state model-checking with automated theorem-proving. Using the two approaches in concert, engineers currently are working on techniques which they hope one day will 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 which can make the symbolic manipulation of arithmetic expressions computationally feasible.

The adjoining sidebars discuss a variety of applications of finite-state computer-aided verification. AT&T, Cadence, IBM, Intel and Motorola all have burgeoning internal verification programs. AT&T apparently has the longest ongoing commercial applications of computer-aided verification (distinguished from earlier defense-related applications of automated theorem-proving). At AT&T, the first commercial product developed using verification was some software for a specialized network data protocol, verified by G. Holzmann in 1984. Since then, finite-state verification has spread, especially to the verification of hardware. Currently, at least four commercial tools for finite-state verification of hardware have been released or announced: AT&T's FormalCheck, Chrysalis' Design VERIFYer, Compass Technology's VFormal and from Viewlogic (under license from Abstract Hardware Ltd.), CheckOff.

The sidebar by Carl Pixley describes a verification project at Motorola which used a proprietary tool named VERDICT. It is based upon the SMV (State Machine Verifier) tool developed recently by Kenneth McMillan as part of his Ph.d. thesis project under E. Clarke at Carnegie Mellon University. SMV in turn is based upon model-checking algorithms developed for CTL (Computation Tree Logic), a logic developed by E. Clarke and A. Emerson in 1980. CTL can test branching properties of the multifarious executions of a program.

Masahiro Fujita describes a verification project at Fujitsu, which used SMV directly. An important difference between these two projects is that the Motorola project ran the verifier directly on the Verilog RTL (Register Transfer Level) language code, whereas the Fujitsu project ran the verifier on an abstraction developed by an engineer experienced with verification.

Verification at the "algorithmic level" — a still higher level of abstraction which embodies the level at which a program is understood as an algorithm, is described by Ken McMillan, the developer of SMV, who used SMV to debug a notoriously complex cache coherency protocol. (Ken also helped with the general preparation of this article.)

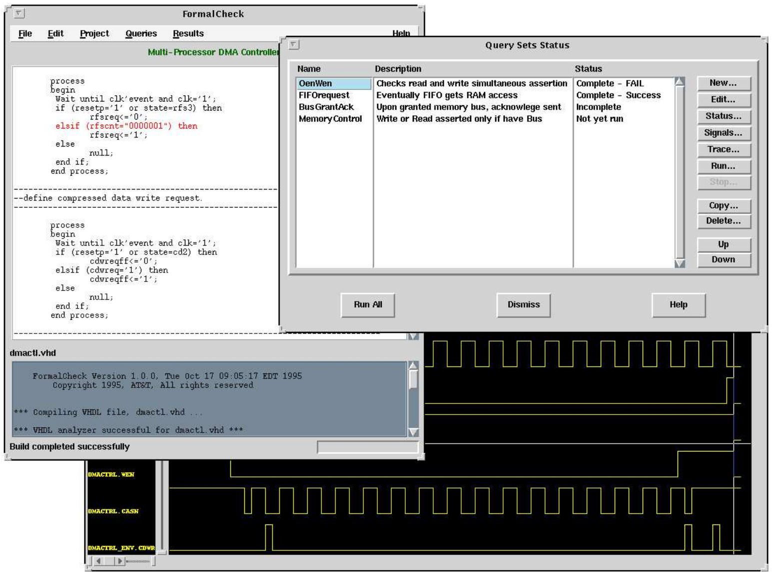
Arthur Glaser has been using finite-state verification for hardware development at AT&T since 1987. His verification-based projects have ranged from a self-contained multi-staff, multi-year project which developed an entire integrated circuit (a packet layer controller), to very small projects completed in a day. The High Definition TV (HDTV) project described here is of the latter type. A distinguishing feature of the commercial FormalCheck tool he uses is its built-in homomorphic reduction algorithm. This complexity reduction algorithm supports the automatic analysis of very large circuits, reducing the necessity to come up with abstractions by hand.

Verification at AT&T

In 1985 work began at AT&T Bell Labs upon a finite-state verification tool called COSPAN. Although not the first such tool, it may be the earliest to be used on commercial projects which is still supported today. Its paradigm of verification is to check if the behaviors (input/output sequences) of one program are contained in the behaviors of another program or alternatively, if they are consistent with a given property. (Technically, this is *automata language containment*). This paradigm supports a top-down development methodology based upon *stepwise refinement.* In this methodology, one 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. The refinement step is such that all properties true of the previous level of abstraction are inherited by the next level. At each subsequent refinement level new properties are checked that are not relevant at higher levels of abstraction. This stepwise refinement process continues until reaching a level at which all details of the ultimate implementation design 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 the stepwise refinement process, one can detect and correct bugs earlier in the design cycle than hitherto possible, thereby accelerating the design process. However, stepwise refinement is not automated: the user must design each refinement step (the tool then checks 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.

The FormalCheck tool, developed by AT&T Microelectronics, uses COSPAN as its verification engine. It embeds COSPAN in a graphical user interface (Figure 2) that facilitates its use, and links it to the VHDL and Verilog languages (COSPAN works only on its native S/R language.) Through the FormalCheck graphical interface, the user defines the properties that are to be verified, input constraints (if any) and runs the verification. Moreover, the interface aids in the analysis and interpretation of any errors that are found in the design under verification. Figure 5 illustrates an example of a graphical visualization of an error track produced by a FormalCheck analysis. It is presented as waveforms for each program signal, just as is done commonly for simulation. Interpretation of design errors is simplified by a utility which relates back to the source code, errors identified in the waveforms. This "back-referencing" is accomplished through a "point and click" operation. FormalCheck may be applied largely automatically to any synthesizable VHDL or Verilog program.



```
        process
        begin
         Wait until clk'event and clk='1';
         if (resetp='1' or state=rfs3) then
                rfsreq<='0';
         elsif (rfscnt="0000001") then
                rfsreq<='1';
         else
                null;
         end if;
        end process;

---------------------------------------------------------
--define compressed data write request.
---------------------------------------------------------

        process
        begin
         Wait until clk'event and clk='1';
         if (resetp='1' or state=cd2) then
                cdwreqff<='0';
         elsif (cdwreq='1') then
                cdwreqff<='1';
         else
                null;
         end if;
        end process;
```

```
    FormalCheck Version 1.0.0, Tue Oct 17 09:05:17 EDT 1995
        Copyright 1995, AT&T, All rights reserved

*** Compiling VHDL file, dmactl.vhd ...

*** VHDL analyzer successful for dmactl.vhd ***
```

Build completed successfully

Name	Description	Status
OenWen	Checks read and write simultaneous assertion	Complete - FAIL
FIFOrequest	Eventually FIFO gets RAM access	Complete - Success
BusGrantAck	Upon granted memory bus, acknowlege sent	Incomplete
MemoryControl	Write or Read asserted only if have Bus	Not yet run

Figure 2: FormalCheck Graphical User Interface. The bottom window is a wave-form rendition of an error (counter-example to a property under verification), shown more fully in Figure 5. On top of that is a pop-up editor window, with the cursor at a line of the source in which the error may have occurred (refer to the caption of Figure 5). At the front is a window which describes the current status of four properties under check.

COSPAN (and FormalCheck) run homomorphic reduction algorithms, both symbolically (using BDD's) and explicitly, to cope with the generally enormous computational complexity in typical designs. Although FormalCheck contains in its core the algorithms necessary for stepwise refinement, these are not currently supported in the FormalCheck tool. Likewise, COSPAN contains other more experimental algorithms not supported in FormalCheck, including timing verification and software bridges to theorem-provers, as well as to the SMV and SPIN finite-state verification systems.

The SPIN software verification system, developed by G. Holzmann at AT&T 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, very popular for software models, makes the verification run faster than synchronous models (such as 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 based upon exploiting symmetries in the order of execution (*partial order* reduction), 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, and has led to two internal tools used for commercial software development: SDLValid, using the SDL language as front-end, and VFSMValid, supporting the VFSM paradigm, both based on SPIN.

In addition to algorithmic verification, there are a number of efforts in automated theorem-proving which have been applied to verification problems at AT&T, as well. Currently, there are over 25 researchers at AT&T Bell Labs whose research interests are partially or wholely formal verification. Even as other areas of research are shrinking, this number is growing, attesting to the importance that AT&T assigns to formal verification. Although FormalCheck, SDLValid and VFSMValid are proprietary to AT&T, the COSPAN and SPIN tools on which they based 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.att.com and gerard@research.att.com, respectively). SMV is available through anonymous ftp from Carnegie Mellon University (inquire of Edmund.Clarke@cs.cmu.edu).

Debugging a Cache Flush Unit with VERDICT

by Carl Pixley

Model checking at the gate and RTL levels is used at Motorola to design and debug some commercial products. Our internal model checking tool, Verdict (by Bernard Plessier and the author), is based upon the SMV symbolic model-checking tool (thus using CTL), and the Verilog hardware description language.

In one model checking success, a Byte Data Link Controller (BDLC) Figure 3 — which is a serial / parallel bus interface module that connects an automotive serial bus to a microprocessor — was designed in Verilog and verified with Verdict. This unit is implemented in approximately six thousand transistors including approximately 150 latches. We use the standard hardware design language Verilog so that when a design is formally verified with Verdict, exactly the same design can be simulated and also synthesized with conventional computer-aided design (CAD) tools. We view formal verification is a complement, rather than an alternative, to simulation. The BDLC was first simulated on a small number of test cases to eliminate obvious errors. The preliminary simulation is important because experience shows that model checking designs with gross errors tends to be time-consuming because the model checker explores a massive set of unanticipated states.

The BDLC is composed of a number of different submodules. For example, the symbol submodule monitors the serial bus and decides what symbol (such as logic 0, logic 1, or start-of-message) is being sent based upon the duration of signal (high or low voltage) it receives and upon the submodule's state. The byte submodule receives characters from the symbol module and forms them into one-byte messages. Both modules implement protocols to send as well as receive data.

During the design of the BDLC, our verification team performed a series of model checking experiments based upon abstractions of submodules of the design. For example, in model checking the BYTE module, we coupled it with an manually abstracted SYMBOL module. In the real design, the SYMBOL module changes state according to the activity on the bus. Unlike the real SYMBOL module, the abstract SYMBOL module has no bus interface but it does have states. For example, 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

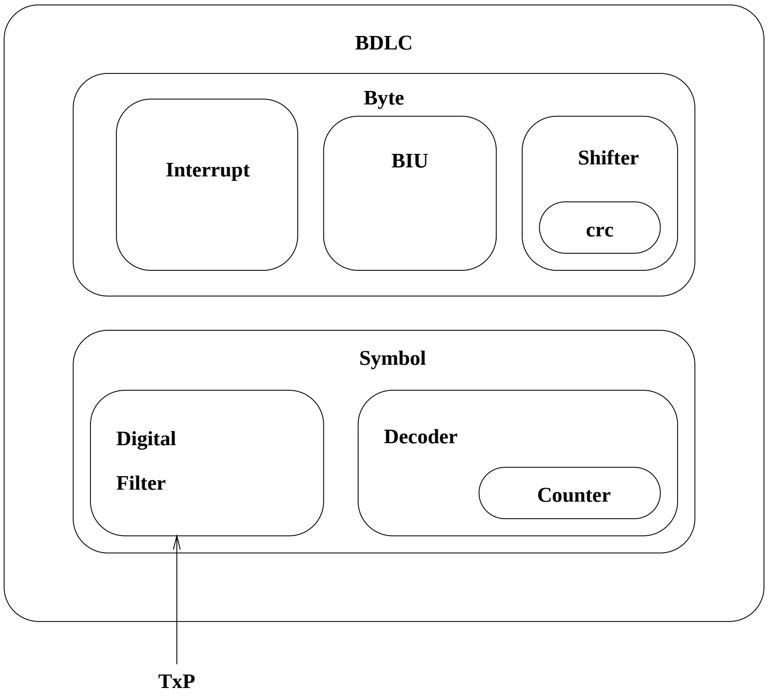


Figure 3: Byte Data Link Controller.

remains in the "bus is idle" state. This behavior generalizes the behavior of the real SYMBOL module. The state of the SYMBOL module determines what signals are communicated to the BYTE module. Properties specific to the BYTE module are then model checked with the simplified abstract SYMBOL module instead of the more complex real SYMBOL module. After checking individual modules in this way and redesigning them if necessary, we were able to put all of the modules together with no abstractions and 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 counterexample traces, that is, sequences of events that demonstrate exactly how a property can fail. For example, early in the design of the BDLC, Verdict found a design error that was indicated by the failure of a CTL property. Verdict produced a specific sequence of events lasting 208 clock cycles that demonstrated the failure of the property. Furthermore, this trace is the *shortest* sequence of events that will cause the property to fail. It seems unlikely that a designer would think of exactly the sequence of at least 208 events that would cause this property to fail. Though Verdict is not based upon simulation, our guarantee with Verdict is that any counterexample trace generated by Verdict can be simulated with the same results by a Verilog simulator. Conversely, if a property is certified by Verdict to be true, no Verilog simulation can be found to violate the property.

Designers new to formal verification often wonder how to discover properties to check. In another successful use of Verdict, we 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. This means that there are nine specific flip-flops each one of which represents one of the nine states. The CF unit is "in" state s when the s-th flip-flop is asserted (that is, has value 1). It is clear that the design should be in one and only one state at any time. So an obvious property to check is whether every state reachable from a valid reset state has the property that exactly one of the nine flip-flops is asserted. This property is very easy to state in CTL and checks in minutes with Verdict.

The one-hot property failed immediately for the original CF unit. Once again, Verdict gave a trace that showed exactly how to get from a valid initial state to a state that violated the one-hot property. After the design was repaired by the designer, *the same properties* were run on the revised design and we 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 also. Since statements of individual CTL specifications tend to be only a few lines long, the set of CTL properties checked becomes a concise document of the design and the specification can be used on similar designs in the future.

Debugging a Communications Chip

by Masahiro Fujita

Masahiro Fujita at Fujitsu Labs, Ben Chen at Fujitsu, and their colleagues successfully used the SMV model checker to debug a complex communications chip (Figure 4) for commercial applications, after field tests had revealed the existence of a bug. The integrated circuit was designed for high-speed switching operations at 156 MHz and had approximately 111K gates (32k for random logic and 79k for RAM). Initially, extensive simulation was used to validate the circuit. After the designers obtained sufficient confidence in the correctness of the circuit, it was fabricated. During field tests, however, Fujitsu engineers observed that the circuit behaved abnormally several seconds after power-up. Some data was duplicated while other data disappeared entirely.

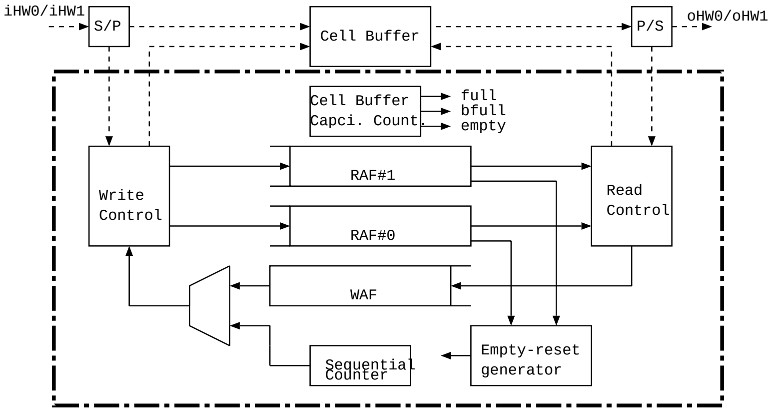


Figure 4: Fujitsu Communications Chip.

Because the abnormal behavior only occurred after several seconds of operation, hundreds of millions of simulation cycles were necessary to reproduce it. Consequently, simulation was impractical as a debugging technique. Even if it were possible to generate so many test cycles (by FPGA-based emulators for example) it would almost certainly be impossible to analyze the enormous amount of information produced in enough detail to find the source of the error. Since obtaining an error-free version of the integrated circuit was a high priority task, Fujita and his colleagues decided to use formal verification techniques.

A CAD engineer, familiar with formal verification, was assigned to debug the chip. The only available information about the chip design was in terms of gate-level circuit descriptions.

Since the circuit had more than 10^5 gates, SMV could not be directly applied to the gate-level circuit description. The engineer decided to exploit the modular structure of the circuit to reduce the state explosion problem. He started with a FIFO that was felt to be the most likely source of the error. The FIFO used specially designed cache registers to speed up the operation of the chip, and the designers were not confident of the behavior of these registers. Surprisingly, the engineer could not find any errors in the FIFO, although SMV was used to check a large number of properties. He concluded that the FIFO was correct and that the error must be in some other part of the circuit.

Eventually he determined that all of the major components of the circuit were correct. This meant that that the error must be caused by the way in which the components were connected together. The engineer decided to construct an abstract model for the entire circuit in the SMV language. By carefully studying the gate-level circuit descriptions of the individual circuit components and conferring with the designers he was able to build a concise model for the chip in SMV. The model did not describe the hardware in the chip that was clearly unrelated to the cause of the error. Also, the width of the data path was reduced as much as possible to reduce the number of states.

Finally, model checking techniques were used to verify this abstraction of the original circuit. Specifications were written in temporal logic that described the abnormal behavior. The SMV model checker (running on a workstation) was able to determine the cause of the error in less than half an hour. The error arose from a condition in which the same address appeared twice in the FIFO. This 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 condition which caused the error derived from incorrect resetting of address recycling circuits. 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. In the actual chip, since the width of the data path was larger and the input data tended to be more random, it took considerably longer for the error to occur. This explains why the error was so hard to find and why simulation and emulation were impractical. It also illustrates the advantage of verification techniques like model checking that methodically explore the entire state-space of a circuit or protocol. From the time the problem first was discovered to the time the cause was understood through model-checking, spanned two months.

Debugging VHDL Code: The HDTV Example

by 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. This chip contains in excess of 100,000 gates. A component of this decoder 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 DRAM. The CDIC also can be programmed by the microprocessor to drop a frame, or sequence of frames, when necessary for buffer control. The control logic for the CDIC contains about 2500 gates. In addition there is an on-chip register file (which contributes significantly to the computational complexity of verification). It is described by about 2000 lines of VHDL code, and contains 1500 latches. In terms of verification capability vs. size, these parameters suggest a small to medium-size example, but very typical in the class of examples that we can expect to verify fully, entirely by automatic means. (Much larger examples may be analyzed and debugged automatically, and verified automatically for special cases, but may require user intervention to verify completely. Our group in AT&T prefers to limit verification to what can be done fully automatically.)

We used our verification system FormalCheck to analyze the CDIC, in particular, to determine if it were possible for the internal FIFO's to overflow under the control protocol, a critical issue for the performance of the circuit. In fact, the analysis showed that there was a condition under which the request to write data to the external DRAM was inhibited, thereby permitting the internal FIFO to overflow: the circuit indeed had a bug!

On account of the complexity of the CDIC model, this analysis would not have been feasible but for a built-in reduction algorithm which automatically produced a reduced model of the CDIC that exhibited this overflow error. The starting point for the reduction 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 some components, mainly some of the memory in the internal FIFO's, that probably could be excluded. The reduction algorithm iterates from this seed, attempting to perform its analysis in a logically conservative (*i.e.*, correctness-preserving) fashion on a much smaller model than the given one. (The algorithm automatically adjusts the model used for analysis, as necessary.) After only about 90 seconds on a Sparc LX work station, the verification tool detected the error in the CDIC algorithm. (The algorithm ensures that in spite of the reduction, any reported error in fact is an error in the original unreduced model.) After correcting this error, we repeated the analysis (with the same seed) 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 a reduced model produced by the algorithm (a very significant state space reduction over the unreduced model, which has an estimated 10^{30} states). The small size of the reduced state space in this case was important, because the complexity of this circuit precluded efficient use of the BDD-based algorithm in FormalCheck. This required us to use instead the FormalCheck algorithm based on explicit state space enumeration. The reduced model is guaranteed conservative in the sense that verifying the reduced model is as good as verifying the original model, in terms of correctness of the result. Verifying the reduced model required a little more than 2.75 hours.

From the time it was decided to check the CDIC to the time the bug was found, took only a few minutes, the time required to set up the verification run by invoking FormalCheck on the appropriate VHDL files, define the property to be checked, and guess a seed for the reduction algorithm. (In this case, choosing a seed was not important, and a default seed would have sufficed.) 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 (the buffer had to be filled before the error could be detected). Interpreting an error track this long can be very tedious, requiring the engineer to examine a succession of actions which ultimately end in a failure, and to determine which action or specific combination of actions caused the bug. FormalCheck presents a waveform visualization of the error track (Figure 5) which may be browsed backward and forward, rearranged and simplified by hiding components, all of which greatly help to interpret the error track. A click with the mouse on a variable in the waveform causes an edit window to pop up with the source code, with the cursor at the place in the source where the designated variable was assigned the value shown in the error track at the point in the waveform where the mouse was clicked. This permits rapid experimentation with proposed bug fixes. Because of this feature, bug fixes tend to be considerably faster than with a simulator. Locating the bug, fixing it correctly (after a few incorrect “fixes”) and the final successful verification run all were completed within the day.