The Verification Times
From Bell Labs Design Automation

Formal Verification in a Commercial Setting

By R. P. Kurshan
Bell Laboratories

Introduction

Formal methods long have been touted as a means to produce "provably correct implementations." It is only recently, however, with rather more modest claims, that one formal method-model checking-has been embraced by industry. In stark contrast with its two-decade development, only the last two years have laid witness to its commercial viability. Nonetheless, in this very short time, this technology has blossomed from scattered pilot projects at a very few commercial sites, into implementations in at least five commercially offered Design Automation tools. This acceleration of activity has even caught the attention of the investment community. Happy graduate students of this technology are basking in an unexpected competition for their talents.

We will examine how this rather astonishing rapid acceptance of a new technology came about, where it is now, and where it may lead. First, why? It is with some annoyance that the present-day practitioners of model-checking view the extravagant claims for general formal methods: these claims were (rightfully!) never broadly accepted in the first place, and served mainly to undercut the credibility of the field. Indeed, even the concept of "provably correct hardware" is nonsensical: one cannot prove anything about a physical object. "Proof" can be applied only to a mathematical model of a physical object, and as such necessarily excludes most of the physical details of the physical object. Moreover, on account of this intrinsic abstraction, it is of questionable value to undertake a tedious, detailed proof process, when it is not so certain what it really means in physical terms when the process is successful.

Instead, model-checking today is seen by the hardware design industry not as a means to ``Bless the Fleet'', but merely as a new and uncommonly effective debugging tool. With the debugging potential afforded by model-checking, designs not only can be made much more reliable than ever before, but (and this may be the real reason for all the excitement) model-checking is seen to accelerate the design process, significantly decreasing the time to market. Increased reliability comes from the ability of model-checking to check ``corner-cases'', which are hard or infeasible to test through simulation. These include especially complex

Welcome

Welcome to Bell Labs Design Automation's Verification Times, a new forum for verification issues!

At BLDA, we've devoted more than twenty years to the development of effective design verification solutions. We're excited about what's happening in the verification community today as the challenge of systems-on-silicon is confronted. We can foresee an exciting future in chip design as today's verification puzzles are solved, paving the way for tomorrow's super chips. And we plan to be there when it happens.

But nothing will happen if people aren't talking about verification and making it a priority. So we've developed this forum to help do just that, and we've chosen the hot subject of formal verification as the inaugural topic. Our feature article, which explores the challenges of commercializing formal verification, is written by Bob Kurshan, one of the foremost experts on this topic in the world. Sidebars throughout this article highlight different aspects of formal verification, as well as our new model checking tool, FormalCheck™. By the way, that's why the "belly-band" wrapper of this issue of *Integrated System Design* displayed a penguin inviting you to check out this insert; our penguin friend appears to be ever ready for a formal event and thus has been adopted as the official mascot for FormalCheck.

If you would like to receive future versions of Verification Times or if you have any questions or comments about this one, either send us e-mail (see last page) or return the reader reply card found on the "belly-band" wrapper of this issue of ISD. We look forward to a fruitful dialogue with all of you as together we face (and solve!) the design verification puzzles of today and tomorrow.

Happy reading!

scenarios unanticipated by the designer. Decreased time to market comes from the ability to apply model-checking earlier in the design cycle and thus find bugs sooner than is possible with simulation. Since model-checking is relatively easier to apply than simulation which needs test vectors and a test bed, model-checking may be used when the design is fluid or only partially defined. Finding bugs early in the design cycle is a well-known accelerant of design development.

Between 1980 and 1990, there were several commercial applications of model-checking in AT&T Bell Labs pilot projects. There were a few similar pilot projects in France, Holland and the UK, and undoubtedly some others of which I am unaware. But the total number of commercial applications (by which I mean ones in which verification was actually inserted in a commercial development process—not just practiced on the side) were pitifully few. This was in spite of the technology having ``proved'' itself in a number of these projects. Around 1990, other serious commercial projects began ramping up, notably in Intel but also at IBM, Motorola and somewhat more speculatively at a number of other companies. But the applications remained largely in pilot projects, and although there was heightened interest, the general attitude remained "wait and see." Today, only a few years later, one can purchase verification tools from Abstract Hardware Ltd. (CheckOff—core technology developed at Siemens), Chrysalis (Design Verifyer), Compass (VFormal—core technology developed at BULL), IBM (RuleBase—core technology developed at CMU), and Lucent Technologies (FormalCheck). [All of the above names are trademarks of the respective companies.] Although the main tools of Chrysalis and Compass are equivalence-checkers (to check the logical equivalence of two designs), they each have model-checkers under development. In addition to these, Intel has very substantial in-house model-checking support and Motorola also has in-house support, in both cases based on core technology developed at CMU. IBM has an in-house equivalence-checker called Verity, which was briefly offered as a commercial tool called BoolesEye. There also is a small industrial effort focused on software verification, notably Telelogic's SDT/SDL tool for the protocol specification language SDL. This involves a different execution semantics than is used with hardware (an asynchronous interleaving of local events) [5], which I will not address further.

So what happened between 1990 and now to cause a technology so recently held in circumspect reserve, to suddenly be the focus of such intense commercialization? There is no single answer, but a number of clear and compelling ones, which not singly but all together provided the stimulus.

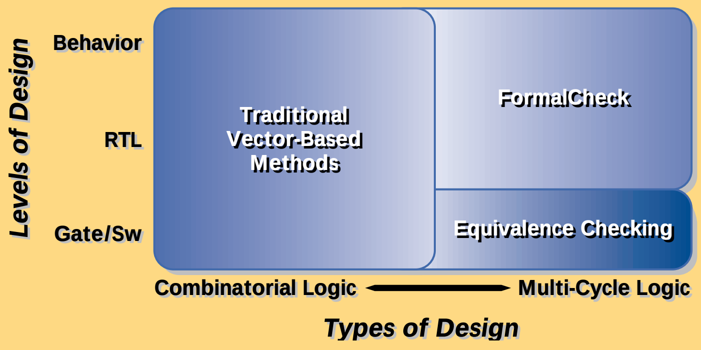
In the beginning of the decade, a number of companies perceived the need for something better than simulation test, understood the promise of model-checking and even accepted the significance of the success of several pilot projects. However, it remained a major step to commit the resources necessary to support (much less commercialize) a mainstream tool. There were all the lurking uncertainties of whether the technology really would generalize, would be viable in the hands of non-experts, and would pay for its own support. But designs were becoming untestable, the cost of bugs was skyrocketing and the need in the hardware design industry for some new testing technology was becoming painfully apparent. Like runners tensed at a starting line, a number of forward-looking companies were waiting for some signal. They wanted neither to chase windmills nor to be also-rans. The signal came not as a seminal event, but a course of events. Equivalence-checkers had paved a path, showing the utility of even this weak form of model-checking. Bugs were becoming news items even before the notorious Pentium bug. Computers ever faster, memory ever larger and BDD-based algorithms [2, 7] made the application of verification technology simpler and simpler: what needed days and advanced techniques only a few years earlier,

Formal Verification— The Next Logical Step

Formal verification is a whole new approach to verifying correct behavior in logic designs. Unlike simulation—where "confidence" comes from running an arbitrary number of test cases through a design—formal verification uses mathematical techniques to examine the entire solution space of a specified design property. There are no vectors. If formal verification says a property is verified, it is—under all conditions. Thus, while simulation is open-ended and uncertain, formal verification removes uncertainty, increasing designer confidence and reducing verification time.

Today, there are two types of commercial formal verification products: equivalence checkers and model checkers. Equivalence checkers can compare two versions of a circuit to guarantee that they are logically identical—a common and formerly tedious post-synthesis task. By contrast, model checkers are most effective when checking a high-level design model (like the Golden RTL) against the original spec. Since model checkers insure that each property is checked under all possible scenarios, the designer can spend more time insuring that all aspects of the spec have been considered.

Functional Verification Tools



Model Checking—An Overview

Model checking represents the most advanced application of formal verification today. Unlike equivalence checking, which is limited to verifying that two versions of a design fulfill the same function, model checking allows the designer to verify the design's behavior against the specification. That makes model checking most useful before synthesis where the cost of repairing design flaws is lowest.

Early model checkers required their users to learn complex languages for property specification. Today, things are different. For example, Bell Labs Design Automation's model checker, FormalCheck, uses a straightforward template approach that guides the user through assembling a complete query. FormalCheck queries specify not only the behavior to be verified but also the pre- and post-conditions which must also be present. If the property is proven true, the designer has 100% confidence that the desired behavior is present under all conditions. That's without writing thousands of vectors.

But what if the query fails? Model checkers provide all the data needed to trace the source of a failing case. FormalCheck, for example, provides a familiar waveform display with a link back to the source design code. Design flaws can be quickly found and repaired.

No vectors and complete confidence—no wonder model checking is taking the design community by storm!

now could be done automatically in a few hours. The race was on.

The what, the how, and the why are the subjects of the following sections.

Disclaimer: to the best of my knowledge, the foregoing and following discussion of various companies' practices is correct. However, all my information has been obtained from second-hand sources, and hence there could be inaccuracies, for which I apologize in advance.

Reduction

If we focus on verification as it is practiced today in hardware design industries, then what we see is model-checking. What makes this technology so attractive to industry is its high degree of automation: the tools can be used by mainstream designers, undiverted by a great deal of thought about the verification process. However, this works only so far as the algorithms actually can handle the size designs the designers need to verify. Even with the best model-checking technology available today, compromises are necessary. One cannot even think about entering a whole microprocessor, much less an entire circuit board design into a verification tool. In fact, although the maximum size design that may be verified is growing literally month by month, the upper limit for verification today is toward the lower limit of a moderate-sized RTL level block. We have succeeded in checking designs with 5,000 latches and 100,000 combinational variables (counting busses and enumerated types as single variables), but for some properties even 500 latches and 50,000 variables is more than we can handle. In the latter cases, in keeping with the need to remain highly automated, we simply pass over these properties, focusing instead on the ones which can be handled automatically. This is in contrast to the academic community, which may dwell on such difficult-to-verify designs, apply advanced ad hoc techniques and ultimately succeed.

There is another model for the verification process, in which verification experts dwell on such hard-to-check properties. However, at Lucent Technologies we have not been successful with this model: as the verification experts commonly are not conversant with the details of the design, they find it hard to keep up with the product development pace.

Thus, it is of paramount importance that the tool be able to reduce the model automatically relative to the property under check, to the greatest extent possible. Most commercial model-checkers have built-in utilities for doing this to some extent. However, there is a great variability in the success of these utilities. Since these utilities determine the extent to which a tool will be able to check a range of designs, they could be considered the most critical aspect of a model-checking tool.

Most of the reductions tend to be property-dependent localization reductions [6], in which the parts of the design model which are irrelevant to the property being checked, are (automatically) abstracted away. In COSPAN [4], the verification engine of FormalCheck, localization reduction is applied dynamically as illustrated in Fig. 1. At each step of the algorithm, the model is adjusted by advancing its "free fence" of induced primary inputs, in order to discard spurious counterexamples to the stated query [6].

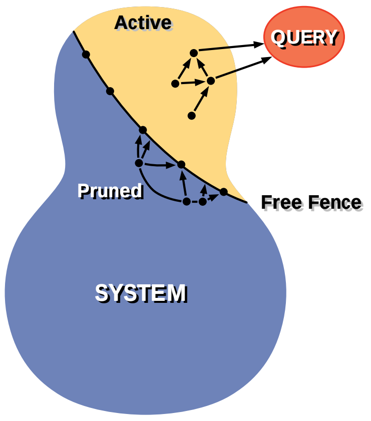


Figure 1. The COSPAN *Localization Reduction* algorithm, through which a design model is reduced dynamically, relative to the query being checked.

Interfaces

A vital part of any commercial verification tool is its user interface. Until recently, the academic community largely ignored user interface issues (it was boring research!), which helped to retard industry acceptance of model-checking. FormalCheck has addressed the user interface issue head-on, providing a commercially-acceptable solution to this tough problem. [Editor's Note: See sidebars elsewhere in this issue.]

Support

Critical to the success of a model-checker, or any commercial tool, is support. This includes documentation, tutorials, an active help-line and of course timely bug fixes in the tool itself. Unlike most other tools, however, an industrial model-checking tool must keep up with a still rapidly evolving technology. This requires a highly competent staff capable of implementing new ideas as the technology develops, as well as originating new algorithms internally. As much of this technology is being patented, commercial players need to be active participants.

Examples of Practice

To be most effective, model-checking should be introduced into the design process at the same time that the first behavioral models are written. The designer is the one who can apply the tool most effectively, as it is the designer who best knows the areas of the design which need the most checking, how to interpret an error track waveform, and what is wrong in an invalid waveform. Today, "behavioral" tends to mean RTL. However, there is a strong movement toward more abstract designs. For now, these too can be represented in VHDL or even Verilog, with the addition of nondeterminism as an abstraction mechanism [6]. There are several ways to introduce nondeterminism, but the most direct may be through an added primary input (which then implements a nondeterministic choice operator). Using this simple stratagem, designs at any level of abstraction may be defined, verified and then refined in a logically consistent manner to a more detailed level of specification. Repeating this process gives rise to a classical "top-down" design strategy, implemented as step-wise refinement. The model-checker can verify the consistency of each level with the previous level, thereby guaranteeing that properties checked at one level of abstraction are inherited by all subsequent levels. When an automata-theoretic framework is used, the consistency of constraints also may be verified from one level to the next.

In spite of the availability of this technology today, few designers are using it, preferring instead to produce flat designs specified and verified at the synthesizable RTL level (meaning, without using nondeterminism as abstraction). However, this is sure to change quickly, as soon as the current set of designer-verifiers become more comfortable with their verification tools. In fact, the tools themselves are frequently automatically performing such abstractions internally (cf. localization reduction, discussed above).

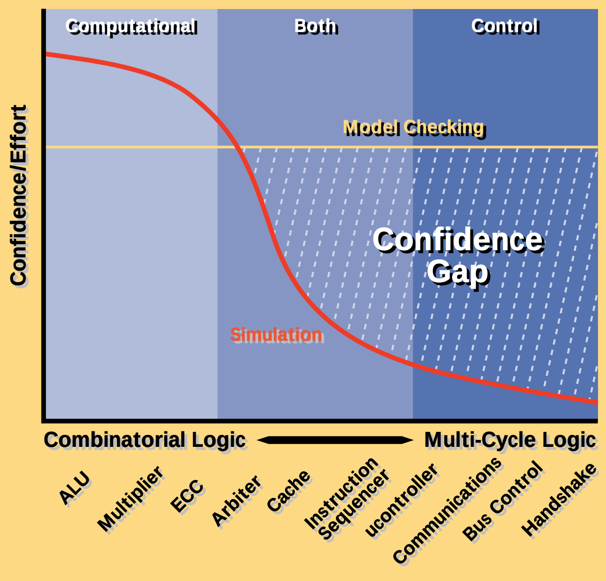
A Case Study: Improving Design Confidence

FormalCheck was recently used to verify a block within an MPEG decoder called the Compressed Data Interface Controller (CDIC). The results present a good example of the tool's speed and efficacy.

The CDIC was responsible for data framing, start code alignment, and synchronization between the incoming data stream, processor, and data buffer. Although the block as a whole was a complex verification problem, much of the complexity was centered around its 2,500 gates of control logic, which controlled the onboard data FIFO containing 2,500 latches (described in 2,000 lines of VHDL code). Bell Labs engineers used FormalCheck to determine if the FIFO within the CDIC would ever overflow under the control protocol.

FormalCheck's automatic reduction algorithms produced a reduced model of the CDIC that accurately and exhaustively analyzed all potential overflow conditions. The new model required searching only 2.5 million states and analyzing 9.8 million transitions, compared with the 10^{30} states on the original design. This reduction enabled FormalCheck to find an error in short order that had been missed in months of simulation.

The Confidence Gap



90 Seconds to Detect an Error

After only 90 seconds of checking the design on a SPARC workstation, FormalCheck detected an error. The trace vectors showed a condition under which the request to write data to external DRAM was inhibited, causing the internal FIFO to overflow. The error trace spanned 2,000 clock cycles, a subtle sequential problem that would have been difficult to uncover with vector-based techniques. After the error was corrected, the analysis proved exhaustively that overflow could never occur.

Bugs of this nature are becoming increasingly common and hard to catch, as demonstrated by today's laborious verification efforts. Model checking is a dramatically faster and more effective way to find them and to improve the confidence of the design.

Although verification can be advantageously applied to global systems such as cache coherence protocols, this often requires some expertise concerning which parts of an otherwise too-large system to include in the verification process. More commonly, industrial practice today is limited to more local "boring" (but nonetheless problematic) controllers such as DMA controllers, bus controllers, MPEG, and arbiters. These alone provide a significant assortment of important applications, more than enough to justify the practice of model-checking, and yet sufficiently limited that the current generation of tools can handle them fairly automatically.

Future

The practice of verification already is evolving in two directions: upward into more abstract behavioral models, and outward into a larger panorama of designs which may be verified automatically. For an overview of current verification practices, see the lecture notes posted from last year's week-long DIMACS tutorial on verification [8].

The upward direction embraces not only abstraction and top-down ("object-oriented" of course!) design development as described in the previous section, but also a new notion of code reuse: at the design level [6]. An abstract verified design may be implemented into several different instantiations, saving not the coding time, but the verification time to check the design.

In the outward direction, strides already have been made at CMU in word-level model-checking [3], permitting the verification of arithmetic units long thought to be beyond the reach of model-checking. Intel (naturally!) has embraced this new technology and reportedly is using it in its current suite of verification tools.

Timing verification [1] is an area in which the technology has advanced well beyond current practice. However, with a renewed interest in asynchronous design, applications may soon be found. Moreover, as designers gain confidence in verification, they may dare to implement prospective design efficiencies that depend upon timing, armed with the confidence that the soundness of these dependencies may be verified.

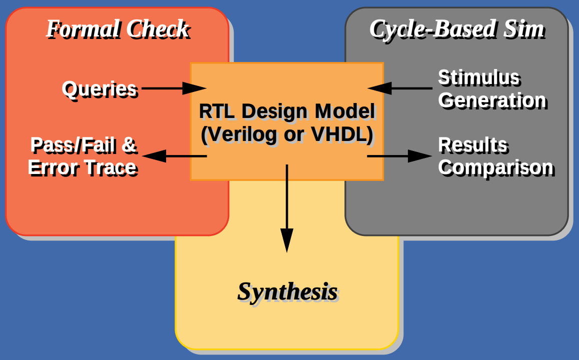
Another direction actively pursued at CMU, Bell Labs and elsewhere is a graceful integration of some possibly limited theorem-proving capabilities into the model-checking paradigm. While successes in this direction have been too limited to be able to predict much promise for this direction, the potential is large, and research in this direction is welcome.

Finally, as the field evolves, it undoubtedly will expand its influence on the evolution of the hardware description languages, leading to ones more suitable and attractive for verification. The very strong interest in software verification, as yet without a firm footing, may find its base in the hardware/software ("co-design") interface, where a number of researchers currently are working.

FormalCheck Has Been Designed to Snap Right Into Your Existing Methodology

FormalCheck accepts the synthesizable subsets of both Verilog and VHDL, ensuring that your verified HDL is exactly the one you will synthesize with no changes and no additional editing. The Queries required for verification are easily entered in minutes and replace the tedious entry of test benches and test vectors. The output FormalCheck produces for debugging is identical to the output from simulation. In other words, FormalCheck truly does snap right into your existing design flow.

FormalCheck Fits Into Existing Design Flow



References

[1] R. Alur, R. P. Kurshan, Timing Analysis in COSPAN, Springer LNCS 1066 (1996) 220-231.

[2] J. R. Burch, E. M. Clarke, D. Long, K. L. McMillan, D. L. Dill, Symbolic Model Checking for Sequential Circuit Verification, IEEE Trans. Computer Aided Design, 13 (1994) 401-424.

[3] E. M. Clarke, R. P. Kurshan, Computer-Aided Verification, IEEE Spectrum, June 1996, 61-67.

[4] R. H. Hardin, Z. Har'El, R. P. Kurshan, COSPAN, Springer LNCS 1102 (1996) 423-427.

[5] G. J. Holzmann, Design and Validation of Computer Protocols, Prentice Hall, 1991.

[6] R. P. Kurshan, Computer-Aided Verification of Coordinating Processes, Princeton Univ. Press, 1994.

[7] K. L. McMillan, Symbolic Model Checking, Kluwer, 1993.

[8] http://dimacs.rutgers.edu/Workshops/SYLA-Tutorials/program.html

Working with Intellectual Property

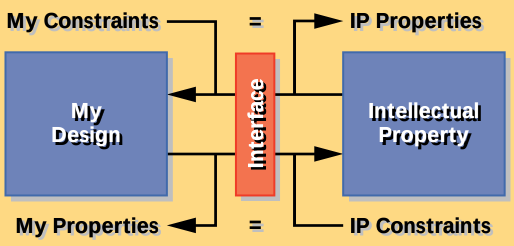
Today's designs are often centered on the re-use of previous designs, or on purchased Intellectual Property (IP), or "cores," used to speed the creation of a larger design. Buying a core or re-using a cell designed by another group introduces a new problem into the design flow: how can you be sure you are using the IP the way its designer expected it would be used?

When a core is being designed, its designer makes assumptions about how it will be used. These assumptions include details such as how data is passed and what transitions are allowed (or perhaps more importantly, illegal) in handshaking with the core. Such assumptions are often subconscious and so poorly documented in a specification. Most specs supply only "good" behaviors like timing diagrams and test vectors; the more important "bad" applications are rarely specified.

When FormalCheck is used to verify a core, part of the process requires that these assumptions be specifically stated as constraints on the environment. Unlike a specification, with FormalCheck these assumptions cannot be understated. If they are, the verification of the core will fail. The format of these "constraints" is exactly the same as the format for the properties (behaviors) the designer wants to verify. If these constraints are passed with the core to the designer of the larger chip, then the constraints can be verified as required behavior of the larger design.

This duality of constraints and properties gives FormalCheck an advantage in smoothing out the problems engineers are finding at the interface between their design and the IP their design incorporates.

FormalCheck Supports IP and Reuse



The FormalCheck Architecture

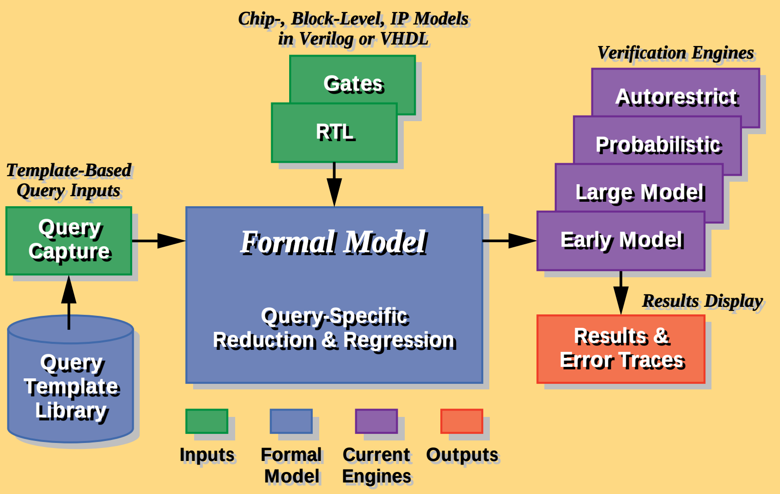
FormalCheck requires only two inputs from the user: the design and the queries. The design is exactly what will be passed on to the synthesis tool to create the gate level model; it requires no additional work by the user. The queries represent the behaviors (properties in FormalCheck speak) which need to be verified, and the assumptions about the environment (constraints in FormalCheck speak) under which the design is expected to exhibit proper behavior. Entering the queries is simplified by the use of a template library.

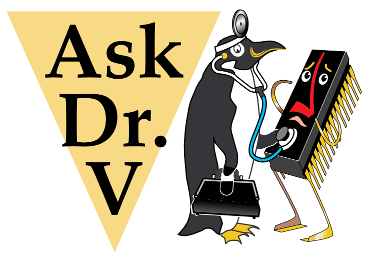
These pieces are automatically combined into the formal model, a formal statement of the problem to be solved. At this point, FormalCheck applies any reduction in the model consistent with the query. This lets the tool handle larger designs and run faster. Also at this point, if the query has been previously verified and is being run on an updated design for regression purposes, FormalCheck automatically checks to see if the design changes could have altered the results of the previous verification. If not, the query is marked as re-verified in a minute or so instead of re-running the full verification algorithm. If the regression test algorithm cannot re-verify the design, the full algorithm is run.

The user can select any of FormalCheck's four verification algorithms. The early model algorithm uses explicit state enumeration and is good at finding errors in a model which is still very green. As the design becomes more robust, the large model algorithm based on BDD's becomes the most efficient. For models which are too large to verify flat, the probabilistic verification is excellent at finding errors. Since each state transition is tested only once, it is much more efficient than simulation using random vectors. Finally, the auto-restrict algorithm combines the probabilistic and BDD-based algorithms, again to speed error detection in very large models.

The output from FormalCheck is pass or fail. That is, the user receives either a verification that the behavior is true under all allowed input conditions, or a simulation-like error trace which can be used to debug the design. That eliminates the possibility of missing the bad behavior because it was buried in millions of cycles of simulation output.

FormalCheck Architecture





Questions about design verification? Our verification expert, Dr. V, wants to help! Just submit your questions to info@blda.lucent.com and Dr. V will provide the answers.

Q: They give out Ph.D.'s in formal verification. Why do you think I'll be able to use FormalCheck?

A: Because we've hidden the details behind our intuitive user interface. All you need to know is the behavior you want to verify, the event that triggers it, and the event that signals when it's no longer required. Pardon the expression, but it's as easy as one, two, three.

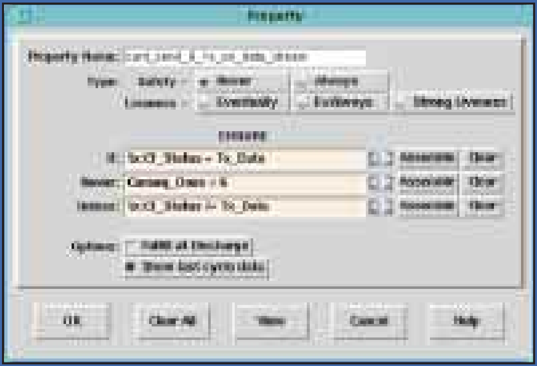
Q: Are there any bugs FormalCheck can find that vector-based solutions can't?

A: Technically, no. Any bug found with FormalCheck can be fed into a simulator and re-found. It only takes the right set of vectors. The real question is: how do you find the right vectors? FormalCheck exhaustively checks every reachable state transition exactly once and, with its symbolic states and reduction algorithms, can often verify more state transitions in an hour than could be run with years of simulation. So, unless your project has a deadline, the answer to this question is NO.

Q: I understand that FormalCheck verifies behaviors under all possible conditions. Should I throw out my HDL simulators?

A: No. These tools complement each other. Here's why. Some circuits, like ALUs, adders and multipliers, are

FormalCheck Is Easy to Use— You Don't Have to Be an Expert



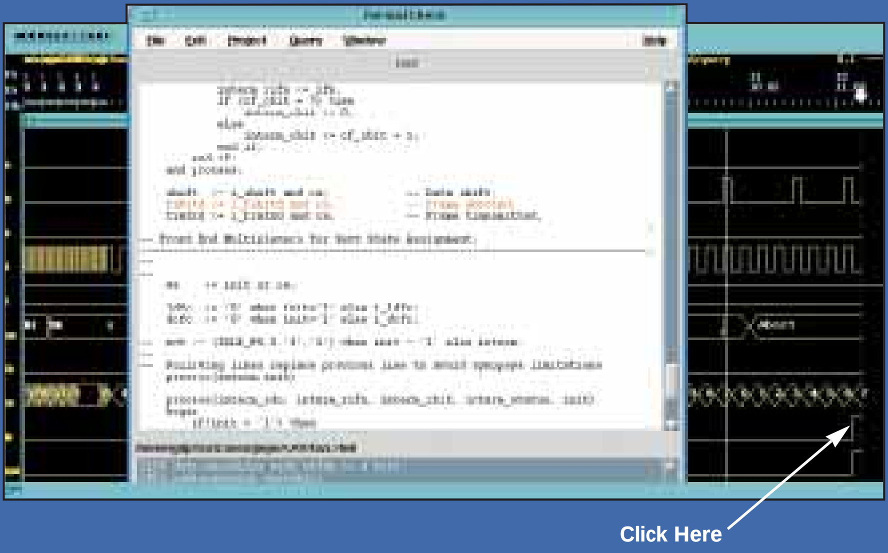
Much of BLDA's recent work on FormalCheck has been to make the tool easy-to-use for the mainstream design engineer. Queries are template-based, so there's no need to learn a new verification language, and most queries take only minutes to write.

All queries are expressed by using only five simple concepts: never, always, eventually, eventually always and strong liveness. Each of these properties can be controlled by an enabling condition and a discharging condition. The enabling condition is a trigger that must be met before the fulfilling behavior is required and the discharging condition describes an event that signals when the behavior is no longer needed.

Enabling: After Data Transmission Starts
Fulfilling: Never Six Consecutive Ones Transmitted
Discharging: Unless Transmission Ended

Automatic Back Referencing to Source Code

That's it. Once the queries are entered, the tool does the rest. FormalCheck searches the reachable state space of the design using this query and exhaustively searches for problems. Failed queries generate error traces which are viewed in a simulation viewer. You can back reference from the traces to the source code line which caused the transition by just clicking on the unexpected value in the error trace.



How simple is simple?

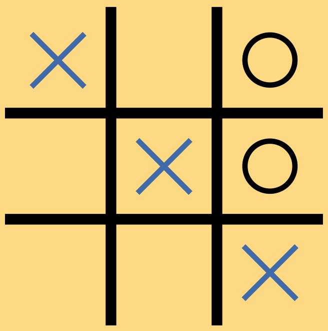
Exhaustive simulation is a myth. The smallest of real designs cannot be truly exhaustively simulated in our lifetimes. Even a trivial example requires millions of vectors for exhaustive simulation. As an example, we give you a Tic-Tac-Toe design and ask, how may vectors are needed to simulate every possible, legal Tic-Tac-Toe game.

Count the vectors this way. Two vectors are needed at the beginning of each game to reset the board, cycle the clock and set who moves first, X or O. Then each move constitutes two vectors (to cycle the clock). A move adds one symbol to an empty square alternating between X and O. The game ends when either player gets three identical symbols in a row either horizontally, vertically or diagonally, or when the board is full and no one has three identical symbols in a row. Thus the shortest game takes 12 vectors (a reset and 5 moves) and the longest game takes 20 vectors.

Check out our web page for a more detailed description of the problem and how you can gain notoriety by submitting the right answer. The results and real answers will be posted in July.

Our web site is at http://www.bell-labs.com/formalcheck

Tic-Tac-Toe



close to pure combinational logic. To exhaustively verify these circuits, you need only verify the behavior under all possible input combinations. Simulators, because of years of tuning and hardware acceleration, will still out-perform FormalCheck for these circuits. However, at the other end of the scale are the control circuits. The behavior of these circuits depends greatly on their history (current state) and exhaustive verification requires trying all possible input combinations from all possible current states. FormalCheck, because it understands state machines and only verifies each state transition once, out-performs simulation on these circuits. Using simulation to find bugs in control circuits is like going skeet shooting with a b-b gun. You might get lucky.

Q: What about the size of my design? How big is too big for FormalCheck?

A: FormalCheck can verify larger designs than any other model checker on the market today. It has verified properties on designs of up to 5,000 latches and 100,000 combinational variables. To tackle such designs, FormalCheck uses a set of automatic reduction algorithms that verify as small a subset of the design as possible where the desired behavior still holds. Meanwhile, the size of the largest verifiable design continues to grow rapidly.

Q: I understand that checking for deadlock at the block level is not enough and that deadlock-free blocks may deadlock when hooked together. Can't this reduction algorithm hide system level deadlock?

A: No. FormalCheck's automatic reduction algorithm always does the reduction based on the specific query. It never reduces the design in any way which can change the outcome of the verification. Thus, the amount of reduction achieved for a design will vary from query to query.

Q: My team is focused on building designs by reusing previous designs or purchased IP. How does FormalCheck work in this environment.

A: FormalCheck supports re-use in two ways. First, since the properties are written using concepts like "eventually" or "always," they usually do not contain the cycle by cycle implementation details found in vectors or test benches. Thus they are more easily reused when verifying a modified version of the IP or previous work. Secondly, FormalCheck requires the IP designer to formalize his assumptions about how the core is to be used. These assumptions can be verified on the larger design. (see the IP sidebar)

How to Reach Us

Sales Offices:
600 Mountain Avenue
Room 3B-431
P.O. Box 636
Murray Hill, NJ 07974-0636
Phone: 908-582-5724
Fax: 908-582-5145

9430 Research Boulevard
Echelon IV, Suite 431
Austin, TX 78759
Phone: 512-343-3618
Fax: 512-342-1985

2600 San Tomas Expwy.
Santa Clara, CA 95051
Phone: 408-562-1358
Fax: 408-562-1301

Lucent Technologies
Bell Labs Design Automation
info@blda.lucent.com
www.bell-labs.com/org/blda
800-875-6590

Lucent Technologies
Bell Labs Innovations