Formal Verification In a Commercial Setting

R. P. Kurshan
Bell Laboratories
Murray Hill, NJ 07974
k@research.bell-labs.com

Abstract

This tutorial addresses the following questions:

- *why* do formal verification?
- *who* is doing it today?
- *what* are they doing?
- *how* are they doing it?
- what about the *future*?

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 in an otherwise lack-luster job market.

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 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) was pitifully small. 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

Design Automation Conference ®

0-89791-847-9/97/0006/$3.50 DAC 97 - 06/97 Anaheim, CA, USA

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) [7], 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 sky-rocketing 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 [3, 9] made the application of verification technology simpler and simpler: what needed days and advanced techniques only a few years earlier, now could be done automatically in a few hours. The race was on.

The *what*, the *how* and the *future* 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.

Paradigms

Formal verification, even verification applied in commercial projects spans a spectrum from the highly expressive but not-so-automated automated theorem-provers, to the significantly restricted but highly automated model-checkers [2, 5]. The theorem-provers have been around for over 35 years, and definitely have their staunch adherents. They have been used extensively in government pilot projects, notably in NASA using SRI's PVS system, but even much earlier with famous theorem-provers like Gypsy. The Boyer-Moore theorem-prover may have been the first to be commercialized, by Computational Logic, Inc.

In spite of impressive demonstrations in the hardware domain and elsewhere, the theorem-provers never achieved the broad level of acceptance for which their advocates had hoped. The reason undoubtedly lies in the need for expert users, and an application cycle which evolves generally slower than a normal product design cycle, so even just keeping up with the project development schedule is a problem. More than that is the difficulty to "sell" the technology to designers. If you tell a designer you have just "verified" her design, she likely will yawn with a (justified!) combination of disbelief and uncertainty of what it means (if I pull the plug, will it keep working??). On the other hand, show her a *bug* in her new design, and she immediately understands the value of your tool, although she may have little idea how the bug was discovered. Model-checkers automatically produce error tracks when the property under check is found by the tool to fail. Theorem-provers by construction are generally incapable of doing that. This is a major practical difference in the two technologies.

On the other hand, model-checking cannot even attempt to verify that a recursively defined FFT algorithm is correct. This is a feasible undertaking for a theorem-prover, at least conceptually. The domain of model-checking is mainly limited to *control-intensive* designs– ones that commonly are described in terms of state machines. Among these are cache coherence protocols, bus controllers, telephone switches, arbiter circuits and communication protocols.

Within model-checking, there are two main approaches: logic-based, and more recently, automaton-based. These two are not at all orthogonal and each can be described in terms of the other. However, the primary logic-based paradigm, based on the logic CTL [4] and the primary automaton-based paradigm, based on ω-automata [8], are to a large extent incompatible with one another, and their differences are important and fundamental. CTL, forming the basis of the CMU and Siemens core technologies mentioned in the Introduction, utilizes a very useful *existential path quantifier*, which enables a designer to check, for example, if it is always possible for his design to be reset. Such a property cannot be posed using ω-automata. On the other hand, ω-automaton reduction algorithms and refinement methodologies often are necessary to deal with commercial-sized designs. These algorithms are intrinsically incompatible with existential path quantification, and thus not viable with CTL-based technologies.

In practice, it is possible to bridge these difficulties both in CTL-based tools and in ω-automaton-based tools. In CTL, some weaker forms of reduction are possible, and if the logic is weakened by removing the existential path quantifier, the automaton-based reductions are largely possible too. For ω-automata, important existential properties such as the reset property, can be offered as a "sanity" check of a reduced design, without the need to require that the property be inherited by the original (unreduced) model. Nonetheless, the two methodologies remain quite distinct. Developing a design through stepwise refinement, with a hierarchy of increasingly more detailed design models is an important strategy for complex design development, and may be implemented using ω-automata; but it is not feasible in the CTL approach. Conversely, the flexibility in CTL to express properties through a general logic formula, is absent with ω-automata.

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 to check designs with 5 K latches and 100 K combinational variables (counting busses and enumerated types as single variables), but for some properties even 500 latches and 50 K 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. (This is reminiscent of the same lesson in the realm of automated theorem-proving.)

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* [8], in which the parts of the design model which are irrelevant to the property being checked, are (automatically) abstracted away. In COSPAN [6], 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 [8].

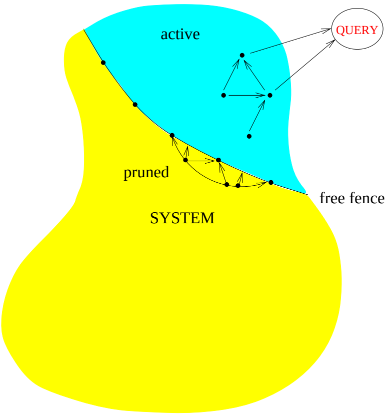

Figure 1: The COSPAN *Localization Reduction* algorithm, through which a design model is reduced dynamically, relative to the query being checked.

Interfaces

A vital part of any commercial verification tool is its user interface. Until recently, the issue of the user interface was largely ignored by the academic community (it was *boring* research!) and to a great extent, this retarded the acceptance of model-checking by industry. Ironically, now that industry itself has focused on the interface as a critical issue and in fact has been the leader in interface construction, university resources are belatedly and thus possibly unadvisedly being applied to this problem.

For years, the academic community split hairs over proposals for the most sublime hardware description language. This pursuit not only reached no consensus, it actually diverged, as its councils grew. Meanwhile, for largely base reasons related more to practice and government sanctions than technical justification, the two arguably poor hardware description languages VHDL and Verilog became industry standards. VHDL was mandated by certain government contracts. Verilog was supported by a growing number of CAD tools (more for historical reasons than an infatuation with the language). If verification were to be broadly used in industrial applications, it became clearer and clearer that the hair-splitting over hardware description languages must give way to a pragmatic adoption of VHDL or Verilog (or preferably, both) as the input language to the verification tool. Today, there is little argument over this view. The moral of the story may be that language crafting is like the proverbial tar-baby which will suck you in, when you'd better be out chasing rabbits.

At the back end of a model-checker, where design bugs are demonstrated through error tracks, there was a spontaneous consensus to represent an error track through the classical simulation test representation of a waveform as shown in Fig. 2.

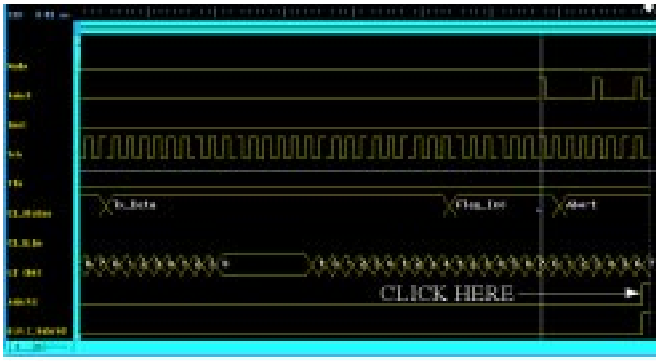

Figure 2: The FormalCheck error track waveform panel.

In the FormalCheck tool, a *back-referencing* utility illustrated in Fig. 3 permits the user to click on an error in the error track waveform, and get a pop-up window of the VHDL or Verilog source, with the cursor on the assignment which gave the indicated variable the indicated value.

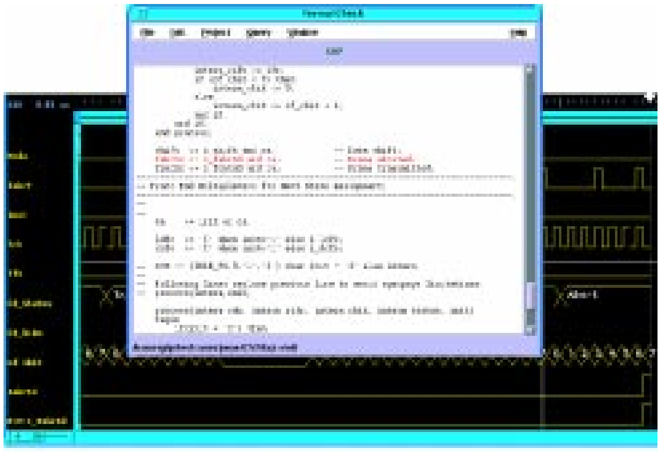

Figure 3: The FormalCheck *Back-Reference* utility through which a click on an error in the waveform pops up the source with the cursor on the line which caused the error.

The hardest and most individual part of the user interface relates to that single aspect of model-checking which is unlike any established practice in design development: defining the property to be checked. With the exception of FormalCheck (and naturally, the equivalence checkers), all the commercial model-checkers use some form of CTL to define properties. The idea of using a logic was discarded by the FormalCheck team, on the premise that this would be alien and therefore unacceptable to many hardware designers. Instead, in FormalCheck, each property is defined using one of a small set of templates, each with a clear, intuitive and simple semantics, and collectively as expressive as the class of ω-automata. The template shown in Fig. 4 defines a property which checks that after each time the designated enabling condition is enabled, the designated fulfilling condition holds continuously unless the discharging condition becomes true. Of course, what is gained in simplicity is lost in flexibility, and there always will remain those who prefer to specify properties in a logic. In FormalCheck, design constraints are defined using a companion set of templates (property templates and constraint templates are paired), and each check of a design model is performed in the context of a set of properties and constraints termed a *query*.

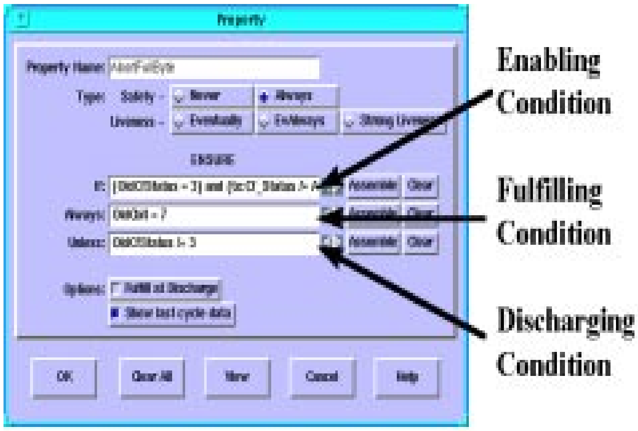

Figure 4: A FormalCheck *Property* panel through which a property to be checked may be posed simply, without the need for a logic.

Support

Critical to the success of a model-checker, or any commercial tool, is support. This includes documentation, tutorials, an active helpline 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 [8]. 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).

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 [10].

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 [8]. 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 [5], 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.

REFERENCES

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

[2] B. Brock, M. Kaufmann, J S. Moore, ACL2 Theorems about Commercial Microprocessors, in M. Srivas and A. Camilleri (eds) Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96), Springer-Verlag, 1996.

[3] 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.

[4] E. M. Clarke, E. A. Emerson, A. P. Sistla, Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, ACM Trans. Prog. Lang. Syst. 8, 1986, 244–263.

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

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

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

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

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

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