Formal Methods: State of the Art and Future Directions

Edmund M. Clarke and Jeannette M. Wing
Carnegie Mellon University

We survey recent progress in the development of mathematical techniques for specifying and verifying complex hardware and software systems. Many of these techniques are capable of handling industrial-sized examples; in fact, in some cases these techniques are already being used on a regular basis in industry. Success in formal specification can be attributed to notations that are accessible to system designers and to new methodologies for applying these notations effectively. Success in verification can be attributed to the development of new tools such as more powerful theorem provers and model checkers than were previously available. Finally, we suggest some general research directions that we believe are likely to lead to technological advances. Although it is difficult to predict where the future advances will come, optimism about the next generation of formal methods is justified in view of the progress during the past decade. Such progress, however, will strongly depend on continued support for basic research on new specification languages and new verification techniques.

Categories and Subject Descriptors: B.1.2 [**Hardware**]: Control Structure Performance Analysis and Design Aids—*Formal models*; B.1.4 [**Hardware**]: Microprogram Design Aids—*Verification*; B.2.2 [**Arithmetic and Logic Structures**]: Performance Analysis and Design Aids—*Verification*; B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids—*Formal models*; B.4.4 [**Input/Output and Data Communications**]: Performance Analysis and Design Aids—*Formal models, verification*; B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Verification*; B.6.3 [**Logic Design**]: Design Aids—*Verification*; B.7.2 [**Integrated Circuits**]: Design Aids—*Verification*; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Program Verification—*Assertion checkers, correctness proofs*; D.3.2 [**Programming Languages**]: Language Classifications—*Design languages, Very high-level languages*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning

This research is sponsored in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government.
Address: Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213
Working Group Members: Rajeev Alur, Edmund Clarke (co-chair), Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jeannette Wing (co-chair), Jim Woodcock, and Pamela Zave.

About Programs—*Mechanical verification, Specification techniques*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Mechanical theorem proving*

General Terms: Software engineering, formal methods, hardware verification

Additional Key Words and Phrases: Software specification, model checking, theorem proving

1. INTRODUCTION

Hardware and software systems will inevitably grow in scale and functionality. Because of this increase in complexity, the likelihood of subtle errors is much greater. Moreover, some of these errors may cause catastrophic loss of money, time, or even human life. A major goal of software engineering is to enable developers to construct systems that operate reliably despite this complexity. One way of achieving this goal is by using *formal methods*, which are mathematically-based languages, techniques, and tools for specifying and verifying such systems. Use of formal methods does not *a priori* guarantee correctness. However, they can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompletenesses that might otherwise go undetected.

The first part of this report assesses the state of the art in *specification* and *verification*. For verification, we highlight advances in *model checking* and *theorem proving*. In the three sections on specification, model checking, and theorem proving, we explain what we mean by the general technique and briefly describe some successful case studies and well-known tools. The second part of this report outlines future directions in fundamental concepts, new methods and tools, integration of methods, and education and technology transfer. We close with summary remarks and pointers to resources for more information.

2. STATE OF THE ART

In the past, the use of formal methods in practice seemed hopeless. The notations were too obscure, the techniques did not scale, and the tool support was inadequate or too hard to use. There were only a few non-trivial case studies and together they still were not convincing enough to the practicing software or hardware engineer. Few people had the training to use them effectively on the job.

Only recently have we begun to see a more promising picture for the future of formal methods. For software specification, industry is open to trying out notations like Z to document a system's properties more rigorously. For hardware verification, industry is adopting techniques like model checking and theorem proving to complement the more traditional one of simulation. In both areas, researchers and practitioners are performing more and more industrial-sized case studies, and thereby gaining the benefits of using formal methods.

2.1 Specification

Specification is the process of describing a system and its desired properties. Formal specification uses a language with a mathematically-defined syntax and semantics. The kinds of system properties might include functional behavior, timing behavior, performance characteristics, or internal structure. So far, specification has been most successful for behavioral properties. One current trend is to integrate different specification languages, each able to handle a different aspect of a system. Another is to handle non-behavioral aspects of a system like its performance, real-time constraints, security policies, and architectural design.

Some formal methods such as Z [Spivey 1988], VDM [Jones 1986], and Larch [Guttag and Horning 1993] focus on specifying the behavior of sequential systems. States are described in terms of rich mathematical structures like sets, relations, and functions; state transitions are given in terms of pre- and post-conditions. Other methods such as CSP [Hoare 1985], CCS [Milner 1980], Statecharts [Harel 1987], Temporal Logic [Pnueli 1981; Manna and Pnueli 1991; Lamport 1984], and I/O automata [Lynch and Tuttle 1987] focus on specifying the behavior of concurrent systems; states typically range over simple domains like integers or are left uninterpreted, and behavior is defined in terms of sequences, trees, or partial orders of events. Still others such as RAISE [Nielsen et al. 1989] and LOTOS [ISO 1987] wed two different methods, one for handling rich state spaces and one for handling complexity due to concurrency. Common to all these methods is the use of the mathematical concepts of abstraction and composition.

The process of specification is the act of writing things down precisely. The main benefit in so doing is intangible—gaining a deeper understanding of the system being specified. It is through this specification process that developers uncover design flaws, inconsistencies, ambiguities, and incompletenesses. A tangible by-product of this process, however, is an artifact, which itself can be formally analyzed, e.g., checked to be internally consistent or used to derive other properties of the specified system. The specification is a useful communication device between customer and designer, between designer and implementor, and between implementor and tester. It serves as a companion document to the system's source code, but at a higher level of description.

Notable Examples

—**CICS**. Oxford University and IBM Hursley Laboratories collaborated in the 1980s on using Z to formalize part of IBM's Customer Information Control System, an on-line transaction processing system with thousands of installations worldwide [Houston and King 1991]. Measurements taken by IBM throughout the development process indicated an overall improvement in the quality of the product, a reduction in the number of errors discovered, and earlier detection of errors found in the process. IBM also estimated a 9% reduction in the total development cost of the new release. The success of this work is well-known and resulted in the Queen's Award for Technological Achievement. It inspired many others to follow suit.

—**CDIS**. In 1992 Praxis delivered to the UK Civil Aviation Authority the CCF Display Information System, a part of the new air traffic management system for London's airspace [Hall 1996]. CDIS is a distributed, fault-tolerant system implemented on nearly 100 computers linked in a dual local area network. Praxis used formal methods as an integral part of the development process and in conjunction with other software engineering, project management, and quality assurance techniques. During requirements analysis, formal description supplemented informal and structured requirements notations. At the system specification stage, an abstract VDM model was developed in conjunction with concrete user interface definitions, semi-formal definitions of the concurrent behavior, and definitions of external interfaces. During design, the abstract VDM was refined into more concrete module specifications. At a lower level, the software for the dual LAN was specified and developed formally using CCS.

Productivity on the project was the same or better than on comparable projects carried out using informal methods. There was, in other words, no net cost in using formal methods. However, the perceived and measured quality of the software was much higher. The delivered software had a defect rate of about 0.75 faults per thousand lines of code, a figure two to ten times better than that for published projects and on comparable software in air traffic control applications that did not use formal methods.

—**Lockheed C130J**. Praxis has been recently working with Lockheed on analyzing the code for the avionic software of the Lockheed C130J [Croxford and Sutton 1995] being supplied to the US Air Force and RAF. The software is coded in the SPARK-annotated subset of Ada. Specifications are written in the Software Productivity Consortium's CORE notation [SPC 1993], which is based on Parnas's tabular specifications [Heninger 1980; Janicki et al. 1996]. Many would expect that the use of SPARK would add to the cost of the software, while improving its quality. The added quality, however, decreased the overall cost of software development because of the huge savings in testing. The use of SPARK annotations to specify the behavior of the modules led to software which is close to being "correct by construction" and hence passes its tests instead of requiring expensive rework.

—**TCAS**. In the early 1990s, the Safety-Critical Systems Research Group at the University of California, Irvine (now at the University of Washington) produced a formal requirements specification for the Traffic Collision Avoidance System (TCAS) II, required on all commercial aircraft flying in U.S. airspace. They used the Requirements State Machine Language (RSML), which is based on Statecharts with changes made to overcome difficulties found during the specification process. Although an industry group was attempting to provide an English language specification at the same time, the complexity of TCAS impeded that process; eventually the English specification effort was abandoned and the RSML specification was adopted instead. After a group of industry and university representatives produced a first draft of the TCAS II specification, a private company on behalf of the Federal Aviation Administration took over the specification effort; official TCAS II documentation still uses RSML. Both the private company and the original university researchers have produced automated tools for RSML including simulators, test case generators and other test tools, and safety analysis tools. The TCAS II specification has been automatically checked for mathematical completeness and consistency [Heimdahl and Leveson 1996] and provably-correct code can now be automatically generated from RSML specifications.

The TCAS II project demonstrated (1) the practicality of writing a formal requirements specification for a complex, process-control system and (2) the feasibility of building a formal model of a system that is readable and reviewable by application experts without special training.

Other case studies in formal specification have been performed primarily on commercial and safety-critical systems. Some are proprietary or lack documentation that we can cite. To give the reader a sense of the applicability of formal methods, we list below some for which we can provide references.

—**Databases**. An HP Medical Instruments real-time database for storing patient monitoring information [Bear 1991].

—**Devices**. A Tektronix family of oscilloscopes [Delisle and Garlan 1990]; a Schlumberger line of household electricity meters [Arnold et al. 1996].

—**Hardware**. An INMOS floating point processor [Barrett 1989]; the virtual channel processor in INMOS's T9000 transputer [Barrett 1995]. (Also see Section 2.2.2.)

—**Medical**. The Clinical Neutron Therapy System at the University of Washington (cyclotron controller) [Jacky 1995].

—**Nuclear**. Argonne National Laboratories' work on the Reactor Safety System for the Experimental Breeder Reactor-II [Chisolm et al. 1987; Kljaich et al. 1989]; the shutdown system of the Darlington Nuclear Generating System in Canada [Archinoff et al. 1990].

—**Security**. The security policy model for the NATO Air Command and Control System [Boswell 1995]; the secure transmission of datagrams in the Multinet Gateway System [Dinolt et al. 1984]; the Token-based Access Control System of the U.S. National Institute of Standards and Technology [Kuhn and Dray 1990].

—**Telephony**. Various features of AT&T's 5ESS telephone switching system using Esterel [Jagadeesan et al. 1996] and combinations of Z and CSP [Mataga and Zave 1995; Zave 1995; Zave and Jackson 1996]; the University of Passau and Siemens Nixdorf's joint work on customizable telephone services and features [Steffen et al. 1996], recently done for Deutsche Telekom.

—**Transportation**. The automatic train protection system for the Paris Metro [Carnot et al. 1992; Guiho and Hennebert 1990]; British Rail's signaling rules [King 1994]; and the on-board avionics software for an Israel aircraft [Harel 1992].

See also [Craigen et al. 1993a; Craigen et al. 1993b; Craigen et al. 1994; Craigen et al. 1995] for a description of twelve case studies in formal methods (most cited above).

2.2 Verification

Two well-established approaches to verification are model checking and theorem proving. They go one step beyond specification; these formal methods are used to analyze a system for desired properties.

2.2.1 *Model Checking.* Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. Roughly speaking the check is performed as an exhaustive state space search which is guaranteed to terminate since the model is finite. The technical challenge in model checking is in devising algorithms and data structures that allow us to handle large search spaces. Model checking has been used primarily in hardware and protocol verification [Clarke and Kurshan 1996]; the current trend is to apply this technique to analyzing specifications of software systems.

Two general approaches to model checking are used in practice today. The first, *temporal model checking*, is a technique developed independently in the 1980s by Clarke and Emerson [Clarke and Emerson 1981] and by Queille and Sifakis [Queille and Sifakis 1982]. In this approach specifications are expressed in a temporal logic [Pnueli 1981] and systems are modeled as finite state transition systems. An efficient search procedure is used to *check* if a given finite state transition system is a *model* for the specification.[1]

In the second approach, the specification is given as an automaton; then the system, also modeled as an automaton, is compared to the specification to determine whether or not its behavior conforms to that of the specification. Different notions of conformance have been explored, including language inclusion [Har'El and Kurshan 1990; Kurshan 1994a], refinement orderings [Cleaveland et al. 1993; Roscoe 1994], and observational equivalence [Cleaveland et al. 1993; Fernandez et al. 1996; Roy and de Simone 1990]. Vardi and Wolper [Vardi and Wolper 1986] showed how the temporal-logic model-checking problem could be recast in terms of automata, thus relating these two approaches.

In contrast to theorem proving, model checking is completely automatic and fast, sometimes producing an answer in a matter of minutes. Model checking can be used to check partial specifications, and so it can provide useful information about a system's correctness even if the system has not been completely specified. Above all, model checking's *tour de force* is that it produces counterexamples, which usually represent subtle errors in design, and thus can be used to aid in debugging.

The main disadvantage of model checking is the state explosion problem. In 1987 McMillan used Bryant's *ordered binary decision diagrams* (BDDs) [Bryant 1986] to represent state transition systems efficiently, thereby increasing the size of the systems that could be verified. Other promising approaches to alleviating state explosion include the exploitation of partial order information [Peled 1996], localization reduction [Kurshan 1994a; Kurshan 1994b], and semantic minimization [Elseaidy et al. 1996] to eliminate unnecessary states from a system model.

Model checkers today are routinely expected to handle systems with between 100 and 200 state variables. They have checked interesting systems with 10^{120} reachable states [Burch et al. 1994], and by using appropriate abstraction techniques, they can check systems with an essentially unlimited number of states [Clarke et al. 1992]. As a result, model checking is now powerful enough that it is becoming widely used in industry to aid in the verification of newly developed designs.

Notable Examples

—**IEEE Futurebus+**. In 1992 Clarke and his students at Carnegie Mellon used SMV [McMillan 1993] to verify the cache coherence protocol described in the IEEE Futurebus+ Standard 896.1-1991 [Clarke et al. 1993; Long 1993]. They

[1]Exhaustive state space search, or reachability analysis, dates back to the earliest papers on Petri Nets. The term "model checking" was coined by Clarke and Emerson [Clarke and Emerson 1981].

constructed a precise model of the protocol in the SMV input language and then used SMV to show that the resulting transition system satisfied a formal specification of cache coherence. They found a number of previously undetected errors and potential errors in the design of the protocol. This appears to be the first time that an automatic verification tool has been used to find errors in an IEEE standard. Although the development of the protocol began in 1988, all previous attempts to validate it were based entirely on informal techniques.

—**IEEE SCI**. In 1992 Dill and his colleagues at Stanford developed the Murphi finite state verification system and verified the cache coherence protocol of the Scalable Coherent Interface, IEEE Standard 1596-1992 [Dill et al. 1992]. The SCI standard defines several protocols, each a subset of the next. They constructed a model of a "typical" protocol and supplied a specification of properties necessary for cache coherence. To avoid errors in the translation, they based their model directly on the C code that is given as a definition of the SCI standard. Since the number of states of the model could be very large, they verified only small instances of the system. Even with this simplification, they found several errors in the protocol, ranging from omissions of variable initializations to subtle logical errors. These errors existed in the rather basic subset that they defined, although the protocol had been extensively discussed, simulated, and even implemented.

—**Stereo components**. One of the emerging application domains of automatic verification is the design of hybrid systems, which consist of both discrete and continuous components. In 1994, Bosscher, Polak, and Vaandrager won a best-paper award for proving manually the correctness of a control protocol used in Philips stereo components [Bosscher et al. 1994]. In 1995, Ho and Wong-Toi verified an abstraction of the protocol using the symbolic model checker HyTech and inferred, fully automatically, a more efficient timing of the protocol than the one used by Philips [Ho and Wong-Toi 1995]. Also in 1995, Daws and Yovine used the verification tool Kronos [Daws and Yovine 1995] to check automatically all the properties stated and handproved by Bosscher et al. In 1996, Bengtsson and his colleagues model checked the entire protocol, thus completing the quest of fully automating a human proof that as little as two years ago was considered far out of reach for algorithmic methods [Bengtsson et al. 1996].

—**ISDN/ISUP**. The NewCoRe Project was the first full-scale application of formal verification methods in a routine software design project within AT&T [Chaves 1992; Holzmann 1994]. The project lasted from 1989 until 1992. Formal modeling and automated verification were applied to the development of the International Telecommunications Union (formerly CCITT) ISDN/IUPP (ISDN User Part Procotol). A team of five "verification engineers" formalized 145 requirements in temporal logic, and rendered the proofs with the help of a special-purpose model checker [Holzmann 1992; Holzmann and Patti 1989]. A total of 7,500 lines of Specification and Description Language (SDL) source code (excluding comments) was verified; 112 errors were revealed (and fixed) in the high-level designs; approximately 55% of the original design requirements were discovered to be logically inconsistent.

—**HDLC**. A High-level Data Link Controller (HDLC) transmitter core was being designed at the Bell Labs Microelectronics Design Center in Madrid, Spain for an Application-Specific Integrated Circuit library of telecommunication macrocells. The standard design process included capture at the register-transfer level using VHDL, simulation, and synthesis. In 1996, late in the process, the formal verification team at Bell Labs offered to run some additional functional verification on the design [Calero et al. 1997]. Since this design was considered to be practically finished, it was not expected that any errors would be found. Within five hours of work, six properties were specified and five were verified, using the FormalCheck verification tool [DePalma and Glaser 1996]. The sixth property was found by FormalCheck to fail, uncovering a bug that would have at least reduced the throughput of the HDLC channel. More likely, this bug would have confused the higher level protocols causing lost transmissions. It took just a few minutes to identify and propose a fix for a design error that managed to escape many hours of logic simulation. The error was corrected and the correction was formally verified using FormalCheck. Plans are now in the works at the Madrid design center to include model checking as part of the standard design process.

—**PowerScale** In 1995 a group at Bull in collaboration with researchers of the Verimag Laboratory used LOTOS to describe the processors, memory controller, and bus arbiter of the multiprocessor architecture called PowerScale. This architecture is based on IBM's PowerPC microprocessor and is used in Bull's Escala series of servers and workstations[2]. They identified four correctness properties, which express the essential requirements for a proper functioning of the arbitration algorithm, and formalized the properties and algorithm in terms of bisimulation relations (modulo abstractions) between finite labelled transition systems. Using the compositional and on-the-fly model checking techniques implemented in the CÆSAR/ALDÉBARAN Development Package (CADP) toolbox, the correctness of the arbitration algorithm was established automatically in a few minutes [Chehaibar et al. 1996].

—**Buildings.** In 1995 civil engineers at North Carolina State University used the Concurrency Workbench to analyze the timing properties of a distributed active structural control system [Elseaidy et al. 1996]. The system in question was designed to make buildings more resistant to earthquakes by sampling the forces being applied to the structure and using hydraulic actuators to exert countervailing forces. The engineers first coded their design in a timed version of the CCS language; the resulting model contained in excess of $2.12 * 10^{19}$ states and was not directly analyzable. However, by using the semantic minimization feature of the Concurrency Workbench, they were able to construct automatically a much smaller system with the same timing properties that could be analyzed. In the course of their analysis they uncovered an error in a timer setting that, if undetected, could have caused the active structural control component to worsen, rather than dampen, the vibration experienced by buildings during earthquakes.

Other successful industrial-sized case studies in model checking are too numerous to list. Evidence that model checking has "come-of-age" is that industry is building their own model checkers or simply using existing ones. Listed below are some well-known model checkers, roughly categorized according to whether the specification

[2]PowerScale and Escala are registered trademarks of Bull.

they check is given as a logical formula or as a machine:

—**Temporal logic model checkers**. The very first two model checkers were EMC [Clarke and Emerson 1981; Clarke et al. 1986; Browne et al. 1986] and CÆSAR [Queille and Sifakis 1982; Fernandez et al. 1996]. SMV [McMillan 1993] is the first model checker to use BDDs. The Spin system [Gerth et al. 1995; Holzmann 1991] uses partial order reduction to reduce the state explosion problem [Holzmann and Peled 1994; Peled 1996]. Murphi [Dill et al. 1992] and UV [Kaltenbach 1994] are based on the Unity programming language [Chandy and Misra 1988]. The Concurrency Workbench [Cleaveland et al. 1993] verifies CCS processes for properties expressed as mu-calculus formulas. SVE [Filkorn et al. 1994], FORMAT [Damm et al. 1995; Damm and Delgado-Kloos 1996], and CV [Déharbe and Borrione 1995] all focus on hardware verification. HyTech [Alur et al. 1996] is a model checker for hybrid systems; Kronos [Daws and Yovine 1995; Henzinger et al. 1994], for real-time systems.

—**Behavior conformance checkers**. The Cospan/FormalCheck system [DePalma and Glaser 1996; Har'El and Kurshan 1990] is based on showing inclusion between omega automata. FDR [Roscoe 1994] checks refinement between CSP programs; most recently, it has been used to verify and debug the Needham-Schroeder authentication protocol [Lowe 1996]. The Concurrency Workbench [Cleaveland et al. 1993] checks a similar notion of refinement between CCS programs; it and the tool Auto [Roy and de Simone 1990] may also be used to minimize systems with respect to observational equivalence and to determine if two systems are observably equivalent.

—**Combination checkers**. Berkeley's HSIS [Hojati et al. 1993] combines model checking with language inclusion; Stanford's STeP [Bjørner et al. 1996] system, with deductive methods; and VIS [Brayton et al. 1996], with logic synthesis. The PVS theorem prover [Owre et al. 1992] has a model checker for the modal mu-calculus [Rajan et al. 1995]. METAFrame [Steffen et al. 1996] is an environment that supports model checking in the entire software development process.

2.2.2 *Theorem Proving.* Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. This logic is given by a *formal system*, which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas. While proofs can be constructed by hand, here, we focus only on machine-assisted theorem proving. Theorem provers are increasingly being used today in the mechanical verification of safety-critical properties of hardware and software designs.

Theorem provers can be roughly classified in a spectrum from highly automated, general-purpose programs to interactive systems with special-purpose capabilities. The automated systems have been useful as general search procedures and have had noteworthy success in solving various combinatorial problems. The interactive systems have been more suitable for the systematic formal development of mathematics and in mechanizing formal methods.

In contrast to model checking, theorem proving can deal directly with infinite state spaces. It relies on techniques like structural induction to prove over infinite domains. Interactive theorem provers, by definition, require interaction with a human, so the theorem proving process is slow and often error-prone. In the process of finding the proof, however, the human user often gains invaluable insight into the system or the property being proved.

Notable Examples

—**SRT division algorithm**. In 1995 Clarke, German, and Zhao used automatic theorem-proving techniques based on symbolic algebraic manipulation to prove the correctness of an SRT division algorithm similar to the one in the Pentium [Clarke et al. 1996]. This verification method runs automatically and could have detected the error in the Pentium, which was caused by a faulty quotient digit selection table. Later Rueß, Shankar, and Srivas used SRI's general-purpose theorem prover, PVS [Owre et al. 1992], on this same example [Rueßet al. 1996].

—**Processor designs**. The Verity verification tool [Kuehlmann et al. 1995] is widely used within IBM in the design of many processors such as the PowerPC and System/390. Applied in a hierarchical manner, the tool can handle entire processor designs containing millions of transistors [Appenzeller and Kuehlmann 1995]. Using this tool, the functional behavior of a hardware system at the register transfer level, gate level, or transistor level, is modeled as a boolean state transition function. Algorithms based on BDDs are used to check the equivalence of the state transition functions for different design levels.

—**Motorola 68020**. In 1991 Boyer and Yu constructed an Nqthm [Boyer and Moore 1979; Boyer and Moore 1988] specification of the Motorola 68020 microprocessor (including 80% of the user-mode instructions) [Boyer and Yu 1996]. They used the specification to prove the correctness of many binary machine code programs produced by commercial compilers from source code in such high-level languages as Ada, Lisp, and C. For example, Yu verified the MC68020 binary code produced by the "gcc" compiler for 21 of the 22 C programs in the Berkeley string library.

—**AMD5K86**. In 1995 Moore and Kaufmann of Computational Logic, Inc., and Lynch of Advanced Micro Devices, Inc., collaborated to prove the correctness of Lynch's microcode for floating point division on the AMD5K86. Starting from an informal proof of correctness they formalized their argument in the ACL2 logic [Kaufmann and Moore 1995] and checked it with the ACL2 mechanical theorem prover. Gaps and mistakes were found in the informal "proof" but in the end the microcode was mechanically shown to be correct [Moore et al. 1996]. The entire effort took about nine weeks. The mechanical proof ended doubt of the code's correctness and allowed testers to focus on other routines. In 1996 Russinoff used ACL2 to check the correctness of the floating point square root microcode [Russinoff 1996]. He found bugs in the microcode itself; after they were fixed, the final version of the square root microcode was also mechanically proved correct.

—**Motorola CAP**. During 1992-1996 Brock of Computational Logic, Inc., working in collaboration with Motorola designers, developed an ACL2 specification of the entire Motorola Complex Arithmetic Processor (CAP), a microprocessor for digital signal processing (DSP). The CAP is the most complicated microprocessor yet formalized, with a three stage pipeline, six independent memories, four multiplier-accumulators, over 250 programmer-visible registers, and an instruction set allowing the simultaneous modification of well over 100 registers in a single instruction. The formal specification tracked the evolving design and included a simpler non-pipelined view that was proved equivalent on a certain class of programs. Finally, Brock used ACL2 to verify the binary microcode for several DSP algorithms [Brock et al. 1996].

—**AAMP5**. During 1993-1995 Srivas of the Stanford Research Institute and Miller of Rockwell International collaborated on the specification and verification of the Collins Commercial Avionics AAMP5 microprocessor. They used PVS to specify 108 of the 209 AAMP5 instructions and verified the microcode for 11 representative instructions [Miller and Srivas 1995].

As with model checking, an increase in the number and kinds of theorem provers provides evidence for a growing interest in theorem proving. There has been a corresponding increase in the number and kinds of examples to which theorem provers have been applied. Below is a list of some well-known theorem provers, categorized roughly by their degree of automation:

—**User-guided automatic deduction tools**. Systems like ACL2 [Kaufmann and Moore 1995], Eves [Craigen et al. 1988], LP [Garland and Guttag 1988], Nqthm [Boyer and Moore 1979], Reve [Lescanne 1983], and RRL [Kapur and Musser 1987] are guided by a sequence of lemmas and definitions but each theorem is proved automatically using built-in heuristics for induction, lemma-driven rewriting, and simplification. Nqthm, the Boyer-Moore theorem prover, has been used to check a proof of Gödel's first incompleteness theorem, and in a variety of large-scale verification efforts.

—**Proof checkers**. Examples include Coq [Cornes et al. 1995], HOL [Gordon 1987], LEGO [Luo and Pollack 1992], LCF [Gordon et al. 1979], and Nuprl [Constable et al. 1986]. They have been used to formalize and verify hard problems in mathematics and in program verification.

—**Combination provers**. Analytica [Clarke and Zhao 1993], which combines theorem proving with the symbolic algebra system Mathematica, has successfully proved some hard number-theoretic problems due to Ramanujam. Both PVS [Owre et al. 1992] and STeP [Bjørner et al. 1996] combine powerful decision procedures and model checking with interactive proof. PVS has been used to verify a number of hardware designs and reactive, real-time, and fault-tolerant algorithms.

3. FUTURE DIRECTIONS

The overarching goal of formal methods is to help engineers construct more reliable systems. Formal methods is thus an area that cuts across almost all other areas in Computer Science. Its foundations lie squarely in mathematics, its intended applications are hardware and software systems, and its potential users are all developers involved in the system engineering process.

Tremendous advances in the past decade have been made on all fronts. As technology improves, it becomes more feasible to attack harder and larger problems.

Progress in the area depends on doing fundamental research, inventing new methods and building new tools, integrating different methods to work together, and making concerted efforts by researchers to work with practitioners to transfer technology effectively.

3.1 Fundamental Concepts

Significant advances in the practical use of formal methods have relied on fundamental results drawn from all areas in Computer Science, not necessarily directly intended for formal methods. Further work needs to be done in the areas of

—**Composition**. We need to understand how to compose methods, compose specifications, compose models, compose theories, and compose proofs.

—**Decomposition**. We need to develop more efficient methods for decomposing a computationally demanding global property into local properties whose verification is computationally simple (e.g., the task decomposition and localization reduction methods of [Kurshan 1994b]).

—**Abstraction**. Real systems are difficult to specify and verify without abstractions. We need to identify different kinds of abstractions, perhaps tailored for certain kinds of systems or problem domains, and we need to develop ways to justify them formally, perhaps using mechanical help.

—**Reusable models and theories**. Rather than defining models and theories from scratch each time a new application is tackled, it would be better to have reusable and parameterized models and theories.

—**Combinations of mathematical theories**. Many safety-critical systems have both digital and analog components. These hybrid systems require reasoning about both discrete and continuous mathematics.
System developers would like to be able to predict how well their system will operate in the field. Indeed they often care more about performance than correctness. Performance modeling borrows strongly from probability, statistics, and queueing theory.

—**Data structures and algorithms**. To handle larger search spaces and larger systems, new data structures and algorithms, e.g., more concise data structures for representing boolean functions, are needed.

3.2 Methods and Tools

No one method or tool can serve all purposes. We need to support all different kinds. From past experience, we have learned what kinds can have the most impact. To be attractive to practitioners, methods and tools should satisfy the following criteria. We realize that some of these criteria are ideals, but they are still good to strive for.

—**Early payback**. Methods and tools should provide significant benefits almost as soon as people begin to use them.

—**Incremental gain for incremental effort**. Benefits should increase as developers get more adept or put more effort into writing specifications or using tools.

—**Multiple use**. It should be possible to amortize the cost of a method or tool over many uses. For example, it should be possible to derive benefits from a single specification at several points in a program's life cycle: in design analysis, code optimization, test case generation, and regression testing.

—**Integrated use**. Methods and tools should work in conjunction with each other and with common programming languages and techniques. Developers should not have to "buy into" a new methodology completely to begin receiving benefits. The use of tools for formal methods should be integrated with that of tools for traditional software development, e.g., compilers and simulators.

—**Ease of use**. Tools should be as easy to use as compilers, and their output should be as easy to understand.

—**Efficiency**. Tools should make efficient use of a developer's time. Turnaround time with an interactive tool should be comparable to that of normal compilation. Developers are likely to be more patient, however, with completely automatic tools that perform more extensive analysis.

—**Ease of learning**. Notations and tools should provide a starting point for writing formal specifications for developers who would not otherwise write them. The knowledge of formal specifications needed to start realizing benefits should be minimal.

—**Error detection oriented**. Methods and tools should be optimized for finding errors, not for certifying correctness. They should support generating counterexamples as a means of debugging.

—**Focused analysis**. Methods and tools should be good at analyzing at least one aspect of a system well, e.g., the control flow of a protocol. They need not be good at analyzing all aspects of a system.

—**Evolutionary development**. Methods and tools should support evolutionary system development by allowing partial specification and analysis of selected aspects of a system.

More ambitiously, rather than build a single tool, we can build "meta-tools" which themselves produce tools customized for a particular problem domain [Steffen et al. 1996], formal notation [Cleaveland et al. 1995], or logic [Gordon 1987; Kindred and Wing 1996]. These meta-tools, like compiler generators, provide an automatic way to build specialized model checkers or proof checkers.

Finally, for any new method or tool, its developer should state explicitly what its strengths, limitations, modeling assumptions, ease of integration with other methods and tools, and start-up costs are. Clear selection criteria help potential users decide what method or tool is most appropriate for the problem at hand.

3.3 Integration of Methods

Given that no one formal method is likely to be suitable for describing and analyzing every aspect of a complex system, a practical approach is to use different methods in combination. When combining methods it is important to consider both:

—Finding a suitable style for using different methods together; and

—Finding a suitable meaning for using different methods together.

Very often neither is addressed adequately. Failure to find a suitable style misses out on the true advantages of combining methods. For example, the Z school stresses the importance of presentation of specifications in an accessible form, with plenty of natural language. This emphasis has helped in popularizing its notation. Any combination must preserve this style of presentation.

Failure to attend to the theoretical foundations of the combination misses out on the true advantages of formality. In chemistry, a distinction is drawn between a *mixture* and a *compound*. In a mixture, the ingredients merely mingle together; in a compound, the ingredients become chemically united. So it is with combining different formal methods. If the meaning of the combination is not properly explained, then the result is merely a mixture: nothing more can be deduced from the joint description than from the separate ones. If the meaning of the combination is explained, then the result is much more powerful. It then becomes possible to have two views of a system specification, and to reason with and refine one view, and to understand the consequences in the other view.

3.3.1 *Model Checking and Theorem Proving.* One of the most promising directions in method integration is in combining model checking and theorem proving [Kurshan and Lamport 1993; Rajan et al. 1995; Bjørner et al. 1996], ideally to benefit from the advantages of both approaches. One way is to employ model checking as a decision procedure within a deductive framework, as is done in tools such as PVS and STeP. For example, a sufficiently expressive logic can be used to define temporal operators over finite state transition systems in terms of maximal or minimal fixed points. For finite state transition systems, these fixed points can be evaluated using a model checker as a decision procedure. For structures with unbounded state spaces, the temporal properties can be verified by means of fixed point induction.

Another way of combining deductive and model checking approaches is to use deduction to obtain a finite state abstraction of an implementation that can be verified using model checking. Such abstractions are commonly used in preparing a problem for model checking but are seldom rigorously verified. Deduction can also be used to verify assumption-commitment proof obligations generated by composing component implementations that have been separately verified by means of model checking. Induction can be combined with model checking to verify systems composed of networks of finite state processes.

3.3.2 *Integration with the System Development Process.* Formal methods can complement less formal methods that are used in the overall system development process. They could be used not instead of, but in addition to, informal methods, as was done by Praxis in the CDIS example. So far formal methods have shown their strength in their use in specification and verification. It is worth exploring how they can be used in requirements analysis, refinement, and testing.

—**Requirements analysis** necessarily deals with customers who often have an imprecise idea of what they want; formal methods can help customers nail down their system requirements more precisely.

—**Refinement** is the reverse of verification; it is the process of taking one level of specification (or implementation) and through a series of "correctness-preserving transformations" synthesizing a lower-level specification (or implementation). Although much theoretical work on refinement has been done, the results have not transferred to practice yet.

—**Testing** is an area that is one of the most costly in all software projects. Formal methods can play a role in the validation process, e.g., using formal specifications to generate test suites [Richardson et al. 1989], and using model and proof checking tools to determine formal relationships between specifications and test suites and between test suites and code.

3.4 Education and Technology Transfer

Education is vital to the success of the formal methods. There are different kinds of audiences:

—**Our research peers**. Some of our greatest skeptics are our own colleagues. We can overcome this skepticism by collaborating with them and their students on systems that they care about.

—**Practitioners**. Technology transfer should be taken very seriously from the very beginning. The recent spread of formal methods is directly related to efforts made by researchers in teaching their techniques to industry.
For effective technology transfer, however, we must keep in mind that success for industry depends on: timely delivery, continuously-enhanced functionality, understanding customers' needs, re-use of legacy code, commitment to quality, elimination of errors, cost-effective development, and real-time performance.

—**Students, at all levels**. Some graduate programs now incorporate formal methods in their curricula [Garlan et al. 1995; Oxford 1996]. Educators are starting to consider teaching formal methods at the undergraduate level. Students need to understand not just how to build single stand-alone programs from scratch, but also how to construct large systems, perhaps using off-the-shelf components and how to maintain legacy code; they need to know not just how to code, but also how to do high-level system design.

4. CONCLUDING REMARKS

Commercial pressure to produce higher-quality software is always increasing. Formal methods have already demonstrated success in specifying commercial and safety-critical software; and in verifying protocol standards and hardware designs. In the future, we expect that the role of formal methods in the entire system development process will increase, especially as the tools and methods successful in one domain carry over to others. Progress, however, will strongly depend on continued support for basic research on new specification languages and new verification techniques.

Ideally, system developers would all be trained sufficiently well that they would not even think that they are using a formal method or tool. They would routinely use the mathematics underlying the notation of a formal specification language as simply a means of communicating ideas to others on their team or of documenting their own design decisions. They would routinely use tools like model and proof checkers with as much ease as they use compilers. Therefore, as researchers in and educators of formal methods, we should strive to make our notations and tools accessible to non-experts.

Towards this ideal, however, it makes sense to cultivate a new career path for specialists in formal methods. They could be experts in the use of one method or tool, or they could be knowledgeable in many, offering their advice on which to use for a given application. Wing envisioned over ten years ago the idea of *specification firms* [Wing 1985], analogous to architecture and law firms, whose employees would be hired for their skills in formal methods. This vision has been realized by the growth both in the number of in-house teams that consult on projects within large corporations (e.g., AT&T and Intel) and in the number of independent companies (e.g., Computational Logic, Inc., Kestrel Institute, and ORA) that specialize in the use of formal methods and do contract work for industry and government agencies. Some companies, such as Praxis, use formal methods as a routine part of their development process.

Finally, for further reading, see the April 1996 issue of *IEEE Computer*, which contains a roundtable discussion on formal methods, and the June 1996 issue of *IEEE Spectrum*, which gives an overview of model checking. On-line forums include the net newsgroup, `comp.specification`, and its subnewgroups for specific methods; and the formal methods mailing list, `fsdm@cs.uq.oz.au`. The Oxford University's web page

`http://www.comlab.ox.ac.uk/archive/formal-methods.html`

points to a wealth of information about formal methods, including papers, reports, tools, conferences, journals, projects, and people.

REFERENCES

ALUR, R., HENZINGER, T., AND HO, P.-H. 1996. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering 22*, 3, 181–201.

APPENZELLER, D. P. AND KUEHLMANN, A. 1995. Formal verification of a PowerPC microprocessor. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'95)* (Austin, TX, Oct. 1995), pp. 79–84.

ARCHINOFF, G. ET AL. 1990. Verification of the shutdown system software at the Darlington Nuclear Generating System. In *Intl. Conf. on Control and Instrumentation in Nuclear Installations* (Glasgow, Scotland, May 1990).

ARNOLD, A., BEGAY, D., AND RADOUX, J.-P. 1996. The embedded software of an electricity meter: An experience in using Formal Methods in an industrial project. *Science of Computer Programming*.

BARRETT, G. 1989. Formal methods applied to a floating-point number system. *IEEE Trans. on Soft. Eng. 15*, 5 (May), 611–621.

BARRETT, G. 1995. Model checking in practice: The t9000 virtual channel processor. *IEEE Trans. on Soft. Eng. 21*, 2 (Feb.), 69–78.

BEAR, S. 1991. An overview of HP-SL. In *Proc. of VDM'91: Formal Development Methods*, Volume 551 of *Lecture Notes in Computer Science* (1991). Springer-Verlag.

BENGTSSON, J., GRIFFIOEN, W., KRISTOFFERSEN, K., LARSEN, K., LARSSON, F., PETTERSSON, P., AND YI, W. 1996. Verification of an audio protocol with bus collision using UppAal. In R. ALUR AND T. HENZINGER Eds., *Computer-Aided Verification '96*, Lecture Notes in Computer Science 1102, pp. 244–256. Springer-Verlag.

BJØRNER, N. ET AL. 1996. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Proc. of the 8th International Conference on Computer-Aided Verification*, Number 1102 in Lecture Notes in Computer Science (July 1996), pp. 415–418. Springer-Verlag.

BOSSCHER, D., POLAK, I., AND VAANDRAGER, F. 1994. Verification of an audio-control protocol. In H. LANGMAACK, W.-P. DE ROEVER, AND J. VYTOPIL Eds., *FTRTFT 94: Formal Techniques in Real-time and Fault-tolerant Systems*, Lecture Notes in Computer Science 863, pp. 170–192. Springer-Verlag.

BOSWELL, A. 1995. Specification and validation of a security policy model. *IEEE Trans. on Software Engineering 21*, 2 (Feb.), 63–68.

BOYER, R. AND YU, Y. 1996. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM 43*, 1 (January), 166–192.

BOYER, R. S. AND MOORE, J. S. 1979. *A Computational Logic*. Academic Press, New York.

BOYER, R. S. AND MOORE, J. S. 1988. *A Computational Logic Handbook*. Academic Press, New York.

BRAYTON, R. ET AL. 1996. VIS: A system for verification and synthesis. In *Proc. of the 8th International Conference on Computer-Aided Verification*, Number 1102 in Lecture Notes in Computer Science (July 1996), pp. 423–427. Springer-Verlag.

BROCK, B., KAUFMANN, M., AND MOORE, J. S. 1996. Heavy inference: Theorems about commercial microprocessors. In M. SRIVAS AND A. CAMILLERI Eds., *Formal Methods in Computer-Aided Design (FMCAD'96)* (November 1996), pp. (to appear). Springer-Verlag.

BROWNE, M. C., CLARKE, E. M., DILL, D. L., AND MISHRA, B. 1986. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers C-35*, 12, 1035–1044.

BRYANT, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers C-35*, 8.

BURCH, J. R., CLARKE, E. M., LONG, D. E., MCMILLAN, K. L., AND DILL, D. L. 1994. Symbolic model checking for sequential circuit verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 13*, 4 (April), 401–424.

CALERO, J., ROMAN, C., AND PALMA, G. D. 1997. A practical design case using formal verification. In *Proc. of Design-SuperCon'97* (1997). To appear.

CARNOT, M., DASILVA, C., DEHBONEI, B., AND MEIJA, F. 1992. Error-free software development for critical systems using the B-methodology. In *Third International IEEE Symposium on Software Reliability Engineering* (1992).

CHANDY, K. AND MISRA, J. 1988. *Parallel Program Design*. Addison-Wesley.

CHAVES, J. 1992. Formal methods at AT&T: An industrial usage report. In *Proc. Formal Description Techniques IV - 1991* (North-Holland, 1992), pp. 83–90.

CHEHAIBAR, G., GARAVEL, H., MOUNIER, L., TAWBI, N., AND ZULIAN, F. 1996. Specification and verification of the powerscale bus arbitration protocol: An industrial experiment with LOTOS. In *Proceedings of FORTE/PSTV'96* (Kaiserslautern (Germany), 1996). Chapman & Hall.

CHISOLM, G., KLJAICH, J., SMITH, B., AND WOJCIK, A. 1987. An approach to the verification of a fault-tolerant, computer-based reactor safety system: A case study using automated reasoning (volume 1, interim report). Technical Report NP-4924 (Jan.), Electric Power Research Institute, Palo Alto, CA. Prepared by Argonne National Laboratory.

CLARKE, E., GERMAN, S., AND ZHAO, X. 1996. Verifying the SRT division algorithm using theorem proving techniques. In *Proc. of the 8th International Conference on Computer-Aided Verification*, Number 1102 in Lecture Notes in Computer Science (July 1996), pp. 111–122. Springer-Verlag.

CLARKE, E. AND KURSHAN, R. (1996). Computer-Aided Verification. *IEEE Spectrum 33*, 6, 61–67.

CLARKE, E. AND ZHAO, X. 1993. Analytica: A theorem prover for Mathematica. *The Mathematica Journal*, 56–71.

CLARKE, E. M. AND EMERSON, E. A. 1981. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, Volume 131 of *Lecture Notes in Computer Science* (1981). Springer-Verlag.

CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS 8*, 2, 244–263.

Clarke, E. M., Grumberg, O., Hiraishi, H., Jha, S., Long, D. E., McMillan, K. L., and Ness, L. A. 1993. Verification of the Futurebus+ cache coherence protocol. In *Proc. CHDL* (1993).

Clarke, E. M., Grumberg, O., and Long, D. E. 1992. Model checking and abstraction. In *Proc. of Principles of Prog. Lang.* (1992).

Cleaveland, R., Madelaine, E., and Sims, S. 1995. Generating front ends for verification tools. In E. Brinksma, R. Cleaveland, K. Larsen, and B. Steffen Eds., *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '95)*, Volume 1019 of *Lecture Notes in Computer Science* (Aarhus, Denmark, May 1995), pp. 153–173. Springer-Verlag.

Cleaveland, R., Parrow, J., and Steffen, B. 1993. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM TOPLAS 15*, 1 (Jan.), 36–72.

Constable, R. et al. 1986. *Implementing Mathematics with the NuPRL Proof Development Environment.* Prentice-Hall.

Cornes, C., Courant, J., Filliâtre, J.-C., Huet, G., Manoury, P., Paulin-Mohring, C., Munoz, C., Murthy, C., Parent, C., Saïbi, A., and Werner, B. 1995. The coq proof assistant reference manual version 5.10. Technical Report 177 (July), INRIA. `http://pauillac.inria.fr/coq/systeme_coq-eng.html`.

Craigen, D., Gerhart, S., and Ralston, T. 1993a. An international survey of industrial applications of formal methods. Technical Report NIST GCR 93/626 (vols. 1 and 2) (March), U.S. National Institute of Standards and Technology. Also published by the U.S. Naval Research Laboratory (Formal Rep. 5546-93-9582, Spet. 1993), and the Atomic Energy Control Board of Canada.

Craigen, D., Gerhart, S., and Ralston, T. 1993b. Observations on industrial practice using formal methods. In *Proc. 15th Int. Conf. on Software Eng.* (May 1993).

Craigen, D., Gerhart, S., and Ralston, T. 1994. Formal methods in critical systems. *IEEE Software 11*, 1 (Jan.).

Craigen, D., Gerhart, S., and Ralston, T. 1995. Formal methods reality check: Industrial usage. *IEEE Trans. on Software Engineering 21*, 2 (Feb.), 90–98.

Craigen, D., Kromodimoeljo, S., Meisels, I., Neilson, A., Pase, B., and Saaltink, M. 1988. m-EVES: A tool for verifying software. In *Proceedings of the 10th International Conference on Software Engineering* (Singapore, April 1988), pp. 324–333.

Croxford, M. and Sutton, J. 1995. Breaking through the V and V bottleneck. In *Proceedings of Ada in Europe 1995* (1995). Springer-Verlag.

Damm, W. and Delgado-Kloos, C. 1996. *Practical Formal Methods for Hardware Design.* Lecture Notes in Computer Science. Springer-Verlag. To appear.

Damm, W., Josko, B., and Schlör, R. 1995. *Specification and Validation methods for Programming Languages and Systems*, Chapter Specification and verification of VHDL-based system-level hardware designs, pp. 331–410. Oxford University Press.

Daws, C. and Yovine, S. 1995. Two examples of verification of multirate timed automata with KRONOS. In *Proc. 1995 IEEE Real-Time Systems Symposium, RTSS'95* (Pisa, Italy, Dec. 1995). IEEE Computer Society Press.

Déharbe, D. and Borrione, D. 1995. Semantics of a verification-oriented subset of VHDL. In P. Camurati and H. Eveking Eds., *CHARME'95, Correct Hardware Design and Verification Methods*, Volume 987 of *Lecture Notes in Computer Science* (Frankfurt, Germany, Oct. 1995), pp. 293–310. Springer-Verlag.

Delisle, N. and Garlan, D. 1990. A formal specification of an oscilloscope. *IEEE Software 7*, 5 (Sept.), 29–36.

DePalma, G. and Glaser, A. 1996. Formal verification augments simulation. *Electronic Engineering Times*, 56.

Dill, D. L., Drexler, A. J., Hu, A. J., and Yang, C. H. 1992. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors* (1992), pp. 522–525.

Dinolt, G. et al. 1984. Multinet gateway—towards A1 certification. In *IEEE Symp. on Security and Privacy* (1984).

Elseaidy, W., Cleaveland, R., and Baugh, J. 1996. Modeling and verifying active structural control systems. *Science of Computer Programming*. To appear. A preliminary version of this paper appears in the *Proceedings of the 1994 Real-Time Systems Symposium*.

Fernandez, J.-C., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., and Sighireanu, M. 1996. CADP (CÆSAR/ALDEBARAN development package): A protocol validation and verification toolbox. In *Proc. of the 8th International Conference on Computer-Aided Verification*, Number 1102 in Lecture Notes in Computer Science (July 1996). Springer-Verlag.

Filkorn, T., Schneider, H., Scholz, A., Strasser, A., and Warkentin, P. 1994. SVE User's Guide. Technical Report ZFE BT SE 1-SVE-1, Siemens AG, Corporate Research and Development, Munich.

Garlan, D., Abowd, G., Jackson, D., Tomayko, J., and Wing, J. 1995. The CMU Master of Software Engineering Core Curriculum. In *Proceedings of the Eighth SEI Conference on Software Engineering Education (CSEE)*, Volume 895 of *Lecture Notes in Computer Science* (New Orleans, March 1995), pp. 65–86. Springer-Verlag.

Garland, S. J. and Guttag, J. V. 1988. Inductive methods for reasoning about abstract data types. In *Proc. of the 15th Symposium on Principles of Programming Languages* (1988), pp. 219–228.

Gerth, R., Peled, D., Vardi, M. Y., and Wolper, P. 1995. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. IFIP/WG6.1 Symp. on Protocol Specification, Testing, and Verification* (Warsaw, Poland, June 1995).

Gordon, M. 1987. HOL: A proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis* (1987). Kluwer.

Gordon, M. J., Milner, A. J., and Wadsworth, C. P. 1979. *Edinburgh LCF*, Volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag.

Guiho, G. and Hennebert, C. 1990. SACEM software validation. In *Twelfth International Conf. on Software Engineering* (1990).

Guttag, J. and Horning, J. 1993. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag. Written with S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing.

Hall, A. 1996. Using formal methods to develop an ATC information system. *IEEE Software 12*, 6 (March), 66–76.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 231–274. Preliminary version: Tech. Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.

Harel, D. 1992. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer 25*, 1 (Jan.), 8–20.

Har'El, Z. and Kurshan, R. P. 1990. Software for analytical development of communications protocols. *AT&T Bell Laboratories Technical Journal 69*, 1 (Jan.–Feb.), 45–59.

Heimdahl, M. and Leveson, N. 1996. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering SE-22*, 6 (June), 363–377.

Heninger, K. 1980. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. on Soft. Eng. 6*, 1 (Jan.), 2–13.

Henzinger, T. A., Nicollin, X., Sifakis, J., and Yovine, S. 1994. Symbolic model checking for real-time systems. *Information and Computation 111*, 111–244.

Ho, P.-H. and Wong-Toi, H. 1995. Automated analysis of an audio control protocol. In P. Wolper Ed., *Computer-Aided Verification '95*, Lecture Notes in Computer Science 939, pp. 381–394. Springer-Verlag.

Hoare, C. A. R. 1985. *Communicating Sequential Processes*. Prentice-Hall International.

Hojati, R., Brayton, R., and Kurshan, R. 1993. BDD-based debugging of designs using language containment and fair CTL. In C. Courcoubetis Ed., *Proceedings of the 5th International Conference on Computer-Aided Verification*, Number 697 in Lecture Notes in Computer Science (1993), pp. 41–57. Springer-Verlag.

Holzmann, G. 1991. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey.

Holzmann, G. 1992. Practical methods for the formal validation of SDL specifications. *Computer Communications*. Special issue on Practical Uses of FDT's.

Holzmann, G. 1994. The theory and practice of a formal method: NewCoRe. In *Proc. IFIP World Computer Congress* (Hamburg, Germany, August 1994).

Holzmann, G. and Patti, J. 1989. Validating SDL specifications: An experiment. In C. Vissers and E. Brinksma Eds., *Proc. 9th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP* (Twente, Neth., June 1989).

Holzmann, G. and Peled, D. 1994. An improvement in formal verification. In *Proc. FORTE94* (Berne, Switzerland, October 1994).

Houston, I. and King, S. 1991. CICS project report: Experiences and results from using Z. In *Proc. of VDM'91: Formal Development Methods*, Volume 551 of *Lecture Notes in Computer Science* (1991). Springer-Verlag.

ISO. 1987. Information Systems Processing–Open Systems Interconnection—LOTOS. Technical report, International Standards Organization DIS 8807.

Jacky, J. 1995. Specifying a safety-critical control system in Z. *IEEE Trans. on Software Engineering 21*, 2 (Feb.), 99–106.

Jagadeesan, L., Puchol, C., and Olnhausen, J. V. 1996. A formal approach to reactive systems software: A telecommunications application in Esterel. *Formal Aspects of Computing 8*, 2 (March), 123–151.

Janicki, R., Parnas, D. L., and Zucker, J. 1996. Tabular representations in relational documents. In C. Brink Ed., *Relational Methods in Computer Science*. Springer-Verlag. To appear.

Jones, C. B. 1986. *Systematic Software Development Using VDM.* Prentice-Hall International, New York.

Kaltenbach, M. 1994. Model checking for UNITY. Technical Report TR94-31 (Dec.), The University of Texas at Austin.

Kapur, D. and Musser, D. 1987. Proof by consistency. *Artificial Intelligence 31*, 125–157.

Kaufmann, M. and Moore, J. S. 1995. *ACL2: A Computational Logic for Applicative Common Lisp, The User's Manual (Version 1.8).* ftp://ftp.cli.com/pub/acl2/v1-8/acl2-sources/doc/HTML/acl2-doc.html.

Kindred, D. and Wing, J. 1996. Fast, automatic checking of security protocols. In *Proc. of the USENIX Workshop on Electronic Commerce Protocols* (1996). To appear.

King, T. 1994. Formalising British Rail's signalling rules. In *FME'94: Industrial Benefit of Formal Methods*, Volume 873 of *Lecture Notes in Computer Science* (1994), pp. 45–54. Springer-Verlag.

Kljaich, J., Smith, B., and Wojcik, A. 1989. Formal verification of fault tolerance using theorem-proving techniques. *IEEE Transactions on Computers 38*, 366–376.

Kuehlmann, A., Srinivasan, A., and LaPotin, D. P. 1995. Verity - a formal verification program for custom CMOS circuits. *IBM Journal of Research and Development 39*, 1/2, 149–165.

Kuhn, D. and Dray, J. 1990. Formal specification and verification of control software for cryptographic equipment. In *Sixth Computer Security Applications Conference* (1990).

Kurshan, R. and Lamport, L. 1993. Verification of a Multiplier: 64 Bits and Beyond. In C. Courcoubetis Ed., *Computer Aided Verification*, Volume 697 of *Lecture Notes in Computer Science* (1993), pp. 166–179. Springer-Verlag.

Kurshan, R. P. 1994a. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press.

Kurshan, R. P. 1994b. The Complexity of Verification. In *Proc. 26th ACM Symposium on Theory of Computing (STOC)* (Montreal, 1994), pp. 365–371.

Lamport, L. 1984. The temporal logic of actions. *ACM TOPLAS*, 872–923.

Lescanne, P. 1983. Computer experiments with the REVE term rewriting system generator. In *Proceedings of the 10th Symposium on Principles of Programming Languages*

(Austin, Texas, Jan. 1983), pp. 99–108.

LONG, D. L. 1993. Model checking, abstraction, and compositional reasoning. Ph. D. thesis, Carnegie Mellon Computer Science Department.

LOWE, G. 1996. Breaking and fixing the Needham-Schroder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, Volume 1055 of *Lecture Notes in Computer Science* (March 1996). Springer-Verlag.

LUO, Z. AND POLLACK, R. 1992. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211 (May), Computer Science Dept., University of Edinburgh.

LYNCH, N. AND TUTTLE, M. 1987. Hierarchical correctness proofs for distributed algorithms. Technical report (April), MIT Laboratory for Computer Science, Cambridge, MA.

MANNA, Z. AND PNUELI, A. 1991. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York.

MATAGA, P. AND ZAVE, P. 1995. Multiparadigm specification of an AT&T switching system. In M. G. HINCHEY AND J. P. BOWEN Eds., *Applications of Formal Methods*, pp. 375–398. Prentice-Hall International.

MCMILLAN, K. L. 1993. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers.

MILLER, S. P. AND SRIVAS, M. 1995. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques* (Boca Raton, FL, 1995), pp. 2–16. IEEE Computer Society.

MILNER, A. 1980. *A Calculus of Communicating Systems*, Volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag.

MOORE, J. S., LYNCH, T., AND KAUFMANN, M. 1996. A mechanically checked proof of the correctness of the AMD5K86 floating point division algorithm. `http://devil.ece.utexas.edu:80/ lynch/divide/divide.html`.

NIELSEN, M., HAVELUND, K., WAGNER, K., AND GEORGE, C. 1989. The RAISE language, method and tools. *Formal Aspects of Computing 1*, 85–114.

OWRE, S., RUSHBY, J., AND SHANKAR, N. 1992. PVS: A prototype verification system. In D. KAPUR Ed., *11th International Conference on Automated Deduction (CADE)*, Volume 607 of *Lecture Notes in Artificial Intelligence* (June 1992), pp. 748–752. Springer-Verlag.

OXFORD. 1996. `http://www.comlab.ox.ac.uk/igdp/` . Master's of Science in Software Engineering.

PELED, D. 1996. Combining partial order reductions with on-the-fly model-checking. *Journal of Formal Methods in Systems Design 8 (1)*, 39–64. Also appeared in the *Proc. of the 6th International Conference on Computer Aided Verification 1994*, Stanford CA, USA, Lecture Notes in Computer Science 818, Springer-Verlag, 377-390.

PNUELI, A. 1981. A temporal logic of concurrent programs. *Theor. Comp. Sci. 13*, 45–60.

QUEILLE, J. AND SIFAKIS, J. 1982. Specification and verification of concurrent systems in CÆSAR. In *Proc. of Fifth ISP* (1982).

RAJAN, S., SHANKAR, N., AND SRIVAS, M. 1995. An integration of model-checking with automated proof checking. In P. WOLPER Ed., *Computer-Aided Verification '95*, Volume 939 of *Lecture Notes in Computer Science* (Liege, Belgium, June 1995), pp. 84–97. Springer-Verlag.

RICHARDSON, D., O'MALLEY, T., AND MOORE, C. T. 1989. Approaches to specification-based testing. In *ACM SIGSOFT 89: Third Symposium on Software Testing, Analysis, and Verification* (Dec. 1989).

ROSCOE, A. 1994. Model-checking CSP. In A. ROSCOE Ed., *A Classical Mind: Essays in Honour of C.A.R. Hoare* (1994). Prentice-Hall.

ROY, V. AND DE SIMONE, R. 1990. Auto/Autograph. In E. CLARKE AND R. KURSHAN Eds., *Computer-Aided Verification '90*, Volume 3 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science* (Piscataway, NJ, June 1990), pp. 235–250. American Mathematical Society.

RUESS, H., SHANKAR, N., AND SRIVAS, M. 1996. Modular verification of SRT division. In *Proc. of the 8th International Conference on Computer-Aided Verification*, Number 1102 in Lecture Notes in Computer Science (July 1996), pp. 123–134. Springer-Verlag.

RUSSINOFF, D. 1996. A mechanically checked proof of the correctness of the AMD K5 floating-point square root algorithm. Submitted.

SPC. 1993. Consortium requirements engineering guidebook. Technical Report SPC-92060-CMC version 01.00.09, Software Productivity Consortium, Herndon, VA.

SPIVEY, J. M. 1988. *Introducing Z: a Specification Language and its Formal Semantics.* Cambridge University Press, Cambridge.

STEFFEN, B., MARGARIA, T., CLASSEN, A., AND BRAUN, V. 1996. The Meta '95 environment. In *Computer-Aided Verification '96*, Lecture Notes Computer Science (New Brunswick, NJ, July 1996). Springer-Verlag. Experience Report for the Industry Day.

STEFFEN, B., MARGARIA, T., CLASSEN, A., BRAUN, V., AND REITENSPIESS, M. 1996. An environment for the creation of intelligent network services. In I. E. CONSORTIUM Ed., *Intelligent Networks: IN/AIN Technologies, Operations, Services, and Applications – A Comprehensive Report* (Chicago IL, 1996), pp. 287–300. Invited contribution. Also invited to the *Annual Review of Communications*, IEC, 1996, pp. 919–935.

VARDI, M. Y. AND WOLPER, P. 1986. An automata-theoretic approach to automatic program verification. In *Proc. of Logic in Computer Science* (1986).

WING, J. 1985. Specification firms: A vision for the future. In *Proceedings of the Third International Workshop on Software Specification and Design* (London, Aug. 1985), pp. 241–243.

ZAVE, P. 1995. Secrets of call forwarding: A specification case study. In *Proceedings of the Eighth International IFIP Conference on Formal Description Techniques for Distributed Systems and Communications Protocols (FORTE '95)* (1995), pp. 153–168. Chapman & Hall.

ZAVE, P. AND JACKSON, M. 1996. Where do operations come from? A multiparadigm specification technique. *IEEE Transactions on Software Engineering 22*, 7 (July), 508–528.