DIMACS Technical Report 96-15
June, 1996

Workshop Summary: Computational and Complexity Issues in Automated Verification

by

Robert Brayton
EECS Department
University of California
Berkeley, CA 94720
brayton@ic.eecs.berkeley.edu

Allen Emerson
Computer Science Department
University of Texas
Austin, TX 78712
emerson@cs.utexas.edu

Joan Feigenbaum[1]
AT&T Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
jf@research.att.com

[1]Permanent Member of DIMACS

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Laboratories, Bell Laboratories, and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

The correctness of computer hardware and software is an area of growing theoretical interest and practical importance. It is now widely acknowledged that effective reasoning about program correctness requires: (i) the use of appropriate formalisms, such as temporal logic or automata, for rigorously specifying correct behavior, and (ii) the use of mechanical reasoning algorithms, such as model checkers, to permit proofs of correctness that could not reasonably be constructed by hand owing to the intractable amount of tedious detail to be constructed automatically. A principal limiting factor in automated verification is the computational complexity of various associated mechanical reasoning problems. This complexity might be prohibitively high because of the combinatorial state explosion problem, which appears in many guises.

The DIMACS Workshop on Computational and Complexity Issues in Automated Verification, held at Rutgers University on March 25-28, 1996, and organized by Bob Brayton (University of California at Berkeley), Allen Emerson (University of Texas), and Joan Feigenbaum (AT&T Laboratories), featured both theoretical and practical work in this important area of computer science. This report contains short abstracts of all talks given at the workshop, as well as pointers to sources of more information.

1 Workshop Program

1.1 Monday, March 25, 1996

1.1.1 Tutorial and Survey Talks

09:00-10:00 A. Pnueli: *Temporal Logic: Entering the Age of Integration*

10:00-11:00 E. Clarke: *Exploiting Abstraction, Compositional Reasoning, and Symmetry in Temporal Logic Model Checking*

11:15-12:15 R. Bryant: *Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification*

1.1.2 BDDs

01:30-02:00 S. Ponzio: *Lower Bounds for Restricted Branching Programs*

02:00-02:30 F. Somenzi: *Reordering Variables in Decision Diagrams*

02:30-03:00 R. Ranjan: *Exploiting Memory Hierarchy for High Performance BDD Manipulation*

1.1.3 Partial Order Verification

03:30-04:30 D. Peled: *Overview of Partial-Order Verification*

04:30-05:00 P. Wolper: *Partial Order Methods or How Independence Defeats Complexity*

1.2 Tuesday, March 26, 1996

1.2.1 Complexity and Expressiveness of Logical Theories

09:00-10:00 N. Immerman: *Descriptive Complexity and Model Checking*

10:00-10:30 M. Vardi: *Verification = Logic + Algorithmics*

11:00-11:30 A. Emerson: *Tree Automata*

1.2.2 Program Checking (aka The Blum School)

01:30-02:30 S. Kannan: *Overview of Checking, Self-Testing, and Self-Correcting*

02:45-03:15 R. Rubinfeld: *Robust Functional Equations with Applications to Program Testing and Correcting*

03:15-03:45 F. Ergun: *Testing Multivariate Functions*

04:00-04:30 R. Kumar: *Issues in Approximate Self-Testing*

04:30-05:00 H. Wasserman: *Checking the Fourier Transform: A Historical Update*

1.3 Wednesday, March 27, 1996

1.3.1 Synthesis

09:00-10:00 N. Bjorner: *A Survey of Reactive Synthesis*

10:00-10:30 A. Aziz: *Optimization of Digital Circuits*

1.3.2 Language Containment

11:00-12:00 R. Kurshan: *Complexity Reduction for Language Containment Tests*

12:00-12:30 O. Kupferman: *Experiences with Reduction Hierarchies*

1.3.3 Theory and Practice

02:00-02:30 R. Alur: *Principles for Hierarchical Reasoning*

02:30-03:00 S. German: *Verifying the SRT Division Algorithm*

03:00-03:30 A. Sistla: *Symmetry and Model Checking*

1.3.4 Integrating Theorem Proving and Model Checking

04:00-04:30 N. Shankar: *The Integration of Model Checking with Theorem Proving in PVS**

04:30-05:00 C. Seger: *An Idiot Savant Theorem Prover for Hardware Verification*

1.4 Thursday, March 28, 1996

1.4.1 Systems and Industrial Experience, Part I

09:00-10:00 K. McMillan: *Design-by-refinement Using the SMV System*

10:00-10:30 D. Dill: *Experience with Murphi*

11:00-12:00 G. de Palma: *FormalCheck: The Productization of COSPAN*

12:00-12:30 T. Shiple: *VIS: A System for Verification and Synthesis*

1.4.2 Systems and Industrial Experience, Part II

02:00-03:00 G. Berry: *Automatic Verification of Esterel Programs*

03:00-03:30 C. Pixley: *Case Studies in Formal Verification of Design Equivalence*

04:00-04:30 Xudong Zhao: *Word-Level Model Checking*

04:30-05:00 C. Eisner: *Formal Verification in IBM: RuleBase, an Industrial-Strength Model-Checking Tool*

05:00-05:30 H. Sipma: *STeP and its Applications*

2 Talk Abstracts

This section contains short abstracts of all of the talks given at the workshop, along with pointers to more complete sources of information. The abstracts are listed alphabetically by speakers' last names.

2.1 Rajeev Alur, Bell Laboratories

Principles for Hierarchical Reasoning

In the top-down design methodology, the designer constructs models of the system at different levels of abstraction. For correctness, it is necessary to prove a refinement relation between successive levels. A careful choice of the underlying modeling paradigm allows decomposition of the proof using compositional and assume-guarantee rules. We present a new concurrency model that supports such reasoning principles. The model represents synchronous and asynchronous components in a uniform framework; such an ability to describe and compose modules with different synchrony assumptions is essential for hardware-software codesign. While synchronous models are based on a notion of atomic computation step, and asynchronous models remove that notion by introducing stuttering, our model is based on a flexible notion of what constitutes a computation step: by applying an abstraction operator to a system, arbitrarily many consecutive steps can be collapsed into a single step. The abstraction operator, which may turn an asynchronous system into a synchronous one, allows us to describe systems at various levels of temporal detail. For describing systems at various levels of spatial detail, we use a hiding operator that may turn a synchronous system into an asynchronous one. We illustrate the model with diverse examples from synchronous circuits, asynchronous shared-memory programs, and synchronous message passing.

This talk is based on joint work with Tom Henzinger (UC Berkeley). More information can be obtained by sending email to `alur@research.bell-labs.com`.

2.2 Adnan Aziz, UT Austin

Optimization of Digital Circuits

The advent of modern VLSI CAD tools has radically changed the process of designing digital systems. The first CAD tools automated the final stages of design, such as placement and routing. As the low level steps became better understood, the focus shifted to the higher stages. In particular logic synthesis, the science of optimizing designs (for various measures such as area, speed, or power) specified at the gate level, has shifted to the forefront of CAD research.

Logic synthesis algorithms originally targeted the optimization of PLA implementations; this was followed by research in synthesizing more general multi-level logic implementations. Currently, the central thrust in logic synthesis is sequential synthesis, i.e., the automatic optimization of the entire system. This can be at the logic level (i.e., the input is a netlist of gates and latches) or the state transition graph level. Designs invariably consist of a set of interacting components. Natural questions related to such systems are what is the optimal choice of a component, and automatically deriving a component so as to satisfy given properties.

Typically, the synthesis process has two stages: First, the set of all possible implementations is characterized (which is the topic of this talk), and then one is chosen according to some optimality criteria. Using the sequential calculus S1S as our basic tool we demonstrate that the set of all implementations can always be captured by a single finite-state automaton, which can be generated automatically. However, in practise such an automaton may be prohibitively large to construct. We suggest an approach that alleviates this problem by defining simpler automata that capture only a subset of possible implementations.

The complete version of this paper is [2]. More information about the speaker's work in general can be found on the Web page `http://www-cad.eecs.berkeley.edu/~adnan`.

2.3 Gerard Berry, Ecole des Mines and Inria

Automatic Verification of Esterel Programs

Esterel is a synchronous language dedicated to reactive systems: software or hardware controllers, embedded systems, communication protocols, etc. The language is imperative and well-adapted to control-dominated applications. Conceptually, Esterel processes evolve synchronously and communicate using zero-delay signal broadcasting. The language primitives include sequencing, concurrency, and process preemtion mechanisms. The semantics is defined in a mathematical way. Software or hardware implementation is based on the finite-state character of control in the language: the generated code or circuit is composed of a FSM that possibly drives a table of data computing action. The FSM can be either represented explicitly as a state graph or implicitly as a boolean circuit.

Since Esterel is applied in areas where bugs are very expensive, automatic bug detection is a primary concern for users. Several formal verification systems have been developed for Esterel:

- The Auto tool family performs bisimulation reduction of an explicit state graph w.r.t. a list of input / output signals. This technique allows the user to visualize reduced models of the automaton, hiding the signals he is not interested in. Thanks to the definition of bisimulation, the reduced model is a full and faithfull image of the original system modulo the input / output abstraction. A BDD-based version is currently under experimentation to avoid state space explosion in building the initial state graph of a program.

See `ftp://cma.cma.fr/pub/verif`.

- The TiGeR BDD system of Madre, Coudert, and Touati implements symbolic algorithms to compute the reachable state space of the control circuit and supports verification of elementary temporal properties. The reachability algorithms have been specifically tailored for controllers and use mixes enumerative/symbolic techniques to compute the reachable state space in a stratified way.

- The TempEst system of L. Jagadeesan (Bell Laboratories) and C. Puchol (Univ. Texas) translates a past temporal logic formula into an Esterel program called an observer to be put in parallel with the original program. With each safety property is associated an output signal of the global system. Proving the property amounts to prove that the signal is never emitted, which is done by explicit or implicit reachability analysis techniques.

  See `http://www.cs.utexas.edu/users/cpg/TempEst/`.

Several verifications of large controllers have been performed in an insustrial context. At Dassault Aviation (avionics), about 15 safety properties have been checked using TiGeR for a landing-gear controller. The proofs were carried out on an Alpha machine with 512 MBytes, using several hours of CPU time. Some requirements involving time counters could not be proved using our rather brute-force technique. At AT&T, similar proofs have been performed using TempEst on a comparatively smaller part of a switch.

Formal verification is also heavily used for regression tests of the Esterel compiler: for each new release, FSM comparison is used on a database of test programs to check that compiler updates do not modify the generated FSMs.

More information about the speaker's work can be found on the Web page
`http://cma.cma.fr/Personnel/Berry.html`.

2.4 Nikolaj Bjørner, Stanford University

A Survey of Reactive Synthesis

The different synthesis problems summarized in Figure 1 were surveyed in the talk. Functional synthesis addresses the problem of synthesizing a functional program from a first or higher-order logical specification of the form $\forall x \exists y \cdot \varphi(x, y)$. A functional or sequential program f is extracted from a proof whose inference rules correspond to programming constructs. The program f satisfies $\forall x \cdot \varphi(x, f(x))$.

The challenge in reactive synthesis is to extract programs realizing specifications in temporal logics. We focus on temporal logics that can express progress properties.

Derivation of synchronization skeletons for closed systems is directly related to the satisfiability problem for temporal logics. Thus, the synchronization skeletons correspond to constructive elimination of the existential quantifiers in the specification $\exists y \cdot \varphi(y)$.

Open synchronous reactive synthesis is the temporal counterpart of functional synthesis. A temporal specification of the form $\forall x \exists y \cdot \varphi(x, y)$ is realized by a module f satisfying $\forall x \cdot \varphi(x, f(x))$. Requirements like: *$f$ may not depend on the future* and *f may not depend on the far past* are imposed to ensure the extracted module can be implemented. We summarize three approaches to this problem: *(i)* the tree-automata, *(ii)* the game automata, and *(iii)* the control theoretic based approach. We demonstrate how the approaches are easily inter-reducible. The main difficulty encountered in the realizability problem is rather the computational complexity of extracting f as well as the resulting size of f. In the branching time spectrum we consider the control theoretic approach for branching time specifications.

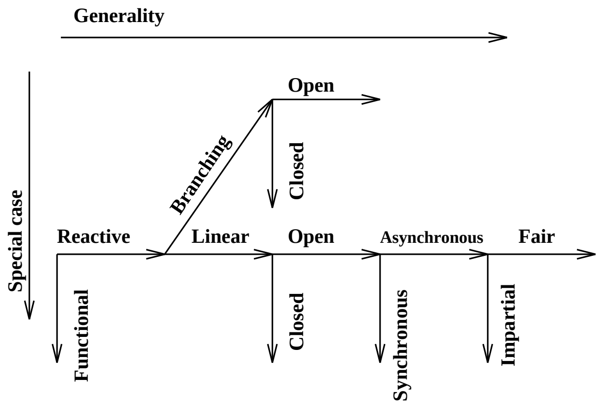


Figure 1: Road map of the reactive synthesis talk

In asynchronous synthesis one allows the environment and synthesized module to interact by interleaving. A third player, the scheduler, takes the role of controlling the synthesized module's behavior. By imposing fairness constraints on the scheduler one faces the *fair* synthesis problem.

The talk concludes with a summary of open theoretical and practical problems. Though the most impressive work has been carried out in the theoretical domain the field seems to need more ambitious examples that will demonstrate how reactive synthesis can be applied in praxis.

A copy of the slides including a list of continuously updated references can be obtained via the web address below.

More information about this and other work of the speaker can be found on the Web page
`http://theory.stanford.edu/people/nikolaj/index.html`.

2.5 Randal Bryant, Carnegie Mellon University

Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification

Ordered Binary Decision Diagrams (OBDDs) have found widespread use in CAD applications such as formal verification, logic synthesis, and test generation. OBDDs represent Boolean functions in a form that is both canonical and compact for many practical cases. They can be generated and manipulated by efficient graph algorithms. Researchers have found that many tasks can be expressed as series of operations on Boolean functions, making them candidates for OBDD-based methods.

The success of OBDDs has inspired efforts to improve their efficiency and to expand their range of applicability. Techniques have been discovered to make the representation more compact and to represent other classes of functions. This has led to improved performance on existing OBDD applications, as well as enabled new classes of problems to be solved.

We provide an overview of the state of the art in graph-based function representations, focussing on several recent advances of particular importance for formal verification. First, dynamic variable ordering, in which the total ordering of variables in the OBDD representation changes as the BDD manipulations proceed, has proved an effective technique for controlling BDD memory requirements. By implementing the reordering as swaps of adjacent variables, this reordering can be implemented efficiently without changing the view presented to the application programmer. Second, we consider methods of relaxing the total variable ordering requirement of OBDDs. One compromise is to allow variables to appear in different orders along different paths, but to maintain the restriction that a given variable appear at most once along any path. This "free ordering" convention is particularly effective in representing verifying systems where the interpretation of different state variables depends on a context that can be determined by decoding a small number of bits of control state. Third, we consider variations of BDDs where the nodes are interpreted according to alternate decompositions of Boolean functions.

Finally, we consider extensions of graph-based function representations beyond Boolean functions to cases where the function arguments are Boolean, but the range is numeric. These "word-level" functions are particularly useful when verifying arithmetic circuits. The use of Binary Moment Diagrams (BMDs) has enabled verification of multiplier and divider circuits, bypassing the exponential complexity encountered representing these functions with BDDs.

More information and this and other work of the speakercan be found on the Web page `http://www.cs.cmu.edu/ bryant`.

2.6 Edmund Clarke, Carnegie Mellon University

Exploiting Abstraction, Compositional Reasoning, and Symmetry in Temporal Logic Model Checking

Model checking is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. Specifications are expressed in temporal logic, and the reactive system is modeled as a state-transition graph. An efficient search procedure is used to determine whether or not the state-transition graph satisfies the specifications.

In this lecture, we briefly describe the basic model checking algorithm and show how it can be used with binary decision diagrams to verify properties of large state-transition graphs. We also discuss how techniques based on abstraction, compositional reasoning, and symmetry can be used to extend the power of the basic algorithm to handle more complex reactive systems.

More information can be found on the Web page `http://www.cs.cmu.edu/ modelcheck`.

2.7 Gary De Palma, Lucent Technologies

FormalCheck, the Productization of COSPAN

Good technology is not enough to get the design community to adopt formal methods. There must be value. Value is measured as the benefits gained from the adoption minus the cost in money, effort, and time. Model checking offers very high benefits, but a change in methodology is a very high cost. We must do everything possible to make it easy for the user to gain access to the technology.

To increase the benefits of using FormalCheck, we have:

- Solved real problems by including an automatic reduction algorithm to increase the size of the problems FormalCheck can attack.
- Included a state enumeration algorithm to make the tool more usable for raw or small designs. There is no BDD "compile" time.
- Included high levels of automation: Reduction is fully automatic, regression testing automatically looks to see whether any part of the design used in a previous verification run has changed. If not, the run can be bypassed.

To minimize the effort to adopt FormalCheck, we have:

- Focused only on RTL verification: There is no design process change; the user just snaps in our tool instead of his RTL simulator.
- Included support for full, synthesizable subsets of Verilog and VHDL – no changes needed.
- Attempted to work with simulation, not replace it. This lowers risk, because simulation is still available if needed.
- Formatted output in a simulator viewer for comfort.
- Aided debugging by always producing a counter example if a property fails.
- Mapped compiled designs back into users' hierarchy and signal names so that their views do not change.

Any commercial formal verification tool needs to make a similar effort to attract the early adopter and prevent the technology from getting a black eye.

For more information about the FormalCheck project, contact the speaker at `gdp@mhcnet.att.com`.

2.8 David Dill, Stanford University

Experience with Murφ

Murφ is both a description language and a verifier for finite state concurrent systems. It is appropriate for protocols and finite-state systems which can reasonably be modelled as a collection of processes that run at arbitrary speeds, where the steps of the processes interleave (only one process takes a step at any time), and where the processes interact by reading and writing shared variables.

The Murφ verifier works by explicitly generating states and storing them in a hash table. It is an "on-the-fly" verifier: It checks for errors when each new state is generated and reports an error as soon as it is detected. We have found that explicit state verification is sometimes advantageous over BDD-based verification because the performance is more predictable and sometimes better (unless extreme cleverness is used in the BDD-based method).

Over the last few years, we have developed several more advanced verification techiques. The first of these was to exploit symmetry in a description. When states behave equivalently up to permutation of the elements of certain finite types, the verifier generates states in the quotient graph, not the original state graph. This has resulted in savings of a factor of 100 in space and time when verifying some cache coherence protocols.

More recently, we have implemented an improvement on a probabilistic verification method of Wolper and Leroy. We have also implemented a reduction which abstracts away from the exact number of some components in the description, so that the verification results are valid for, say, any number of concurrent processes.

Murφ has been used in both industry and academia. Our most successful application area has been multiprocessor cache coherence. Murφ was used to verify cache coherence protocols in the Stanford DASH and FLASH multiprocessors, a multiprocessor at Sun, the Sun Ultrasparc processor, and a multiprocessor at HaL computing. It was also integrated into the University of Wisconsin Teapot system to validate user-defined cache protocols.

For more information about Murφ, contact the speaker at `dill@cs.stanford.edu`.

2.9 Cindy Eisner, IBM

Formal Verification in IBM: RuleBase, an Industrial-Strength Model-Checking Tool

RuleBase is an industrial-strength formal verification tool developed at the IBM Haifa Research Laboratory. It is the result of three years of experience in practical formal verification of hardware.

Industrial-strength requirements include significant size (up to a few thousands of latches), a fully automated path from RTL to verification, usability by non-experts, and debugging aids which allow insight into the design. Each of these issues is addressed.

While a few thousands of latches cannot be model-checked (because of size problems), we can use formal verification techniques on units of this size. The idea is to restrict the environment of the unit under test (typically a complicated piece of control logic), and then perform automatic reductions that bring the size of the logic to be model-checked down to manageable proportions. The core model checker (SMV) is then employed.

RuleBase provides a fully automated path from RTL to verification. The inputs to RuleBase are the implementation in the user's HDL, an environment in an HDL-like language (a dialect of SMV), and the specification in Sugar/CTL, the RuleBase specification language. The three different types of inputs are then translated by RuleBase into SMV/CTL for model checking. The HDL translation path uses the user's own synthesis tool under a technology- independent synthesis. This allows RuleBase to support multiple HDLs (VHDL, Verilog, DSL) without needing to address the semantic issues of the RTLs themselves.

RuleBase overcomes the most difficult usability issue in traditional model checking - the specification language - by providing syntactic sugaring of the temporal logic CTL in the specification language Sugar. In Sugar, for instance, the Pentium Bus rule "last_data should be asserted with the fourth datum after a request" is expressed as "$AG(request \rightarrow next\_event(data)[4](last\_data))$." The equivalent CTL is "$AG(request \rightarrow\ !E[!dataU(data\ \&\ EXE[!dataU(data\ \&\ EXE[!dataU(data\ \&\ EXE[!dataU(data\ \&\ !last\_data)])])])])$."

RuleBase has been used for over three years for the unit verification of various designs in IBM. Tens of bugs per unit not found using traditional simulation techniques are usually found using formal verification. In addition, RuleBase has been applied to the verification of cache coherence protocols. Formal verification has found several specification holes in two cache coherence protocols, as well as one coherence violation resulting from a very specific sequence of events.

In addition to three years experience by expert users, RuleBase has been used for one and a half years by designers of the IBM Haifa Design Group to verify their own designs. Technology transfer to a number of other IBM laboratories is expected within a few months. In addition, discussions are underway with outside companies interested in using the tool.

For more information about RuleBase, contact the speaker at `cindye@vnet.ibm.com` or Ilan Beer at `beer@vnet.ibm.com`.

2.10 Alan Emerson, University of Texas at Austin

Tree Automata

Finite state automata on infinite trees play a fundamental role in computer aided verification using branching time temporal logic. Many such logics can be viewed as alternating tree automata; once converted to nondeterministic form using the tableau construction from modal logic and, as needed, determinization of automata on infinite strings, we can reduce the model checking and satisfiability problems for such branching time logics as CTL and CTL*. We give the relevant technical definitions and outline the application of this machinery to model checking and satisfiability testing.

For more information about tree automata, contact the speaker at `emerson@cs.utexas.edu`.

2.11 Funda Ergun, Cornell University

Testing Multivariate Functions

Self-testing programs provide an alternative approach to the problem of program correctness that has the advantage of being program independent. One can construct self-testers by exploiting the set of properties that uniquely define the function that the program purportedly computes and testing that they hold at random inputs. The previous testers, even though they introduce very small overhead while testing univariate functions, become significantly more costly in the case of multivariate functions, since the number of properties that define the function grows infeasibly large in those instances. In this talk we mention techniques for finding a much smaller set of such properties, which lead to more efficient testers for certain multivariate functions. We give general techniques to test linear and multilinear multivariate functions, as well as techniques for specific nonlinear functions. Our techniques yield efficient self-testers for the following linear functions: FFT, evaluation of polynomials, dot product, differentiation, integration, high order differentiation and operators in elementary Jordan canonical normal form, and finite extensions of finite fields. For multilinear applications, our methods yield a tester for polynomial multiplication that makes $O(1)$ calls to the program, in contrast to $O(\log n)$ of the best previously known tester, and we present a new tester for matrix multiplication, as well as a tester for inner product and linear functions over finite extensions of finite fields. We demonstrate how to test nonlinear functions such as polynomial exponentiation and composition.

All of the testers we construct make $O(1)$ calls to the program that is being tested, therefore the asymptotic complexity of the whole operation remains unchanged after the addition of the testing overhead..

This is joint work with Ravi Kumar and D. Sivakumar. For more information, contact the speaker at `ergun@cs.cornell.edu`.

2.12 Steven M. German, IBM

Verifying the SRT Division Algorithm

Proving the correctness of arithmetic operations has always been an important problem. The importance of this problem has been recently underscored by the highly-publicized division error in the Pentium processor. We use automatic theorem proving techniques to verify a floating-point division circuit. The circuit uses a radix four SRT division algorithm that is similar to the one used in the Pentium. The algorithm looks ahead to find the next quotient digit in parallel with the generation of next partial remainder. An 8-bit ALU estimates the next remainder's leading bits. A quotient digit look-up table generates the next quotient digit depending on the leading bits of the estimated remainder and the leading bits of the divisor.

In our approach to verification, we formalize the circuit and its correctness conditions as a set of algebraic relations over the real numbers. These algebraic relations correspond closely to the bit-level structure of the circuit, and could have been generated mechanically from a hardware description. Most of the hardware for the SRT algorithm can be described by linear inequalities.

The proof is carried out fully automatically by a powerful theorem prover called Analytica that we have developed. Analytica is based on the symbolic computation system Mathematica. Analytica uses the Sup-Inf algorithm to decide the truth of formulas containing linear inequalities.

This is joint work with Ed Clarke and Xudong Zhao. The paper appears in the proceedings of the 1996 CAV conference, and a reprint is available from the speaker at `german@watson.ibm.com`.

2.13 Neil Immerman, University of Massachusetts at Amherst

Descriptive Complexity and Model Checking

Computational complexity was originally defined in terms of the natural entities of time and space, and the term complexity was used to denote the time or space used in the computation. Yet to a mathematician rather than checking whether an input satisfies a property S, a more natural question might be what is the complexity of expressing the property S? These two issues – checking and expressing – are closely related. It is startling how closely tied they are when the latter refers to expressing the property in first-order logic of **finite** and **ordered** structures.

We explain some of the tight relationships between describing versus computing. For example, NP equals the set of second-order existential properties; P equals the set of properties describable in first-order logic plus inductive definitions; and, for all k, DSPACE(n^k) is equal to the set of properties describable by a uniform sequence of first-order sentences using $k+1$ variables.

There are close relationships between model checking and descriptive complexity. The models used in model checking are finite logical structures. The μ-calculus used in model checking is the monadic part of the inductive language that captures polynomial time. In the second part of the talk, we explore these connections.

In particular, there is a linear-time algorithm to translate any formula from branching time temporal logic (CTL or CTL$*$) into an equivalent expression in transitive closure logic FO(TC). Traditionally, temporal logic has been used to express queries for CAV model checking and then translated into μ-calculus for evaluation. However, evaluation of μ-calculus formulas is complete for time polynomial in the (typically huge) number of states in the Kripke structure. Thus, this is often not feasible, not parallelizable, and incremental strategies are unlikely to exist. By contrast, evaluation of any formula in FO(TC) requires only NSPACE$(\log n)$. This means that the space requirements are manageable, the entire computation is parallelizable, and dynamic evaluation may be possible.

The results of this talk will appear in a joint paper with Moshe Vardi. Related work, including surveys of descriptive complexity, can be found on the speaker's homepage:
`http://www.cs.umass.edu/ thtml/people/fac/immerman.html`.

2.14 Sampath Kannan, University of Pennsylvania

Overview of Checking, Self-Testing, and Self-Correcting

Program checking is a paradigm for proving the correctness of the output of programs. The program is treated as a black box whose output is checked each time it is used. To contrast checking with verification – verification proves the program correct by looking at the internals of the program while checking only shows that the output is correct. Checking is often easier to do than verification. While a checker can be designed for a problem and then check every program for that problem, verification has to be done for each program separately.

To produce mathematical guarantees the checker should be used each time a program is used. The checker is allowed to be probabilistic and is allowed to interact with the program, *i.e.*, to feed other inputs to the program and perform some computation on all of the outputs produced by the program. We want to ensure that the checker does not simply recompute the function being computed by the program. In order to enforce this, we require that the checker's running time be little-oh of the running time of the program being checked where calls to the program being checked count as one step. If we want efficient, practical, checkers we should insist on including the actual times taken for the calls, but theoretically the model where calls are counted as one step is more interesting.

Program checkers have been designed for a number of problems such as extended GCD, matrix multiplication, graph isomorphism, various problems in arithmetic, group theory, linear algebra, and combinatorial optimization. In addition, extensions of the basic model have been used in designing checkers for parallel programs, programs that generate probability distributions, programs that store and retrieve data from memory, and programs that compute over the reals.

A stronger notion than program checking is the notion of a self-testing/self-correcting pair. A self-tester ensures that a program is correct with high probability on a chosen distribution for the inputs and the self-corrector uses a program that has been self-tested to compute the correct answer on any input with high probability. Self-tester/self-corrector pairs have been designed for several numerical problems and the concept also finds important application in the theory of probabilistically checkable proofs.

2.15 S. Ravi Kumar, Cornell University

Issues in Approximate Self-Testing

In this talk, we explore various issues in approximate self-testing. The notion of self-testing (*i.e.*, using the program itself to ensure that it computes the correct function) can be extended to the case when the program is imprecise due to the limitations of floating point arithmetic. The self-tester should be resilient to these approximations. Previous work by Gemmell *et al.* and Ar *et al.* gave some initial results for testing programs in this setting. However, other than testing functions that can be viewed as homomorphisms, very little is known. We present results in this direction for linear functions, multilinear functions, and polynomials in the case when the domain is not closed (*e.g.*, rational domains).

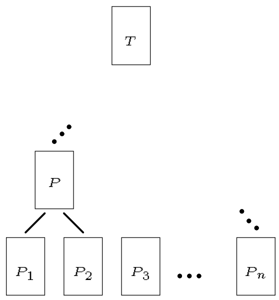
This is joint work with Funda Ergun and Ronitt Rubinfeld. For more information, contact the speaker at `ravi@cs.cornell.edu`.

2.16 Orna Kupferman, Bell Laboratories

Experiences with Reduction Hierarchies

In the automata-theoretic approach to computer aided verification of coordinating processes, we check whether n processes $P_1, P_2, \ldots, P_n$ perform a certain task T by solving the language containment problem $L(P_1) \cap L(P_2) \cap \cdots \cap L(P_n) \subseteq L(T)$. Conventional methods for solving this problem are doomed to fail, as the size of the product $P_1 \times P_2 \times \cdots \times P_n$ is very big (at least exponential in n). Kurshan has suggested the method of *Reduction Hierarchies* as a heuristic for solving this problem [10].

In the reduction-hierarchy method, as demonstrated in the figure on the right, we place the processes $P_1, P_2, \ldots, P_n$ at the bottom of the hierarchy, place the task T at its top, and try "to bridge the gap". This is done by repeatedly replacing two processes A_1 and A_2 in the hierarchy by a new process A that meets the following two conditions.



1. $L(A) \supseteq L(A_1) \cap L(A_2)$.
2. A is small (same order as A_1 and A_2).

Clearly, if we succeed in constructing such a hierarchy (that is, we are left with two processes A_1 and A_2 whose intersection is empty), then the product $P_1 \times P_2 \times \cdots \times P_n$ is contained in T.

We distinguish between two types of hierarchies. One is *tree hierarchies*, where each process can be used at most once. The second is *DAG hierarchies*, where the number of times we can use each process is not limited. In the talk, we referred mainly to tree hierarchies. As we showed in several examples, reduction tree hierarchies are often very effective. To check how effective they are, we considered the following three questions.

1. Does a hierarchy alway exist?

2. When a hierarchy is likely to exist?

3. How to find a hierarchy?

From examples that answer the first question to the negative, we tried to learn the answers to the other questions. We showed that a crucial point in constructing a hierarchy is picking the processes A_1 and A_2 wisely: the more variables the processes A_1 and A_2 share, the bigger are our chances to replace them with a small process. We showed that searching for a process A to replace A_1 and A_2, the language of A should not only contain $A_1 \cap A_2$, but should also be contained in the union of T with the complements languages of processes that were not used in the construction of A. These two observation are now guiding us when we turn to implement in COSPAN a procedure that searches for hierarchies.

More information can be obtained from the speaker at `ok@research.bell-labs.com`.

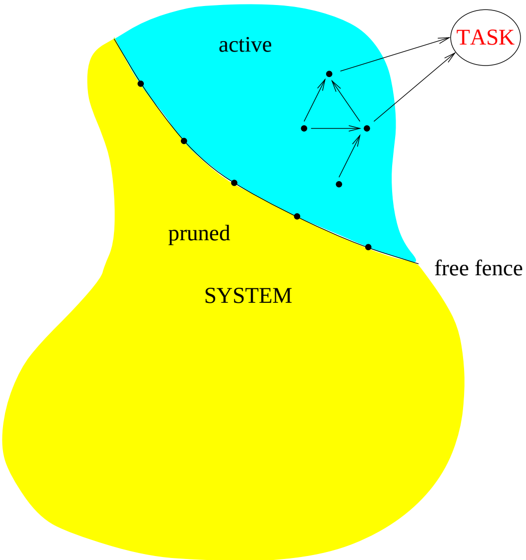
2.17 Robert P. Kurshan, Bell Laboratories

Complexity Reduction for Language Containment Tests

Algorithmic verification based on ω-automata consists of testing the language containment $\mathcal{L}(P) \subset \mathcal{L}(T)$, where P is an automaton which models a system and T is an automaton which captures the property with respect to which P is to be verified. Commonly, P is expressed in terms of a parallel composition $P = \otimes P_i$ of component automata P_i, satisfying the *language intersection property* $\mathcal{L}(P) = \cap \mathcal{L}(P_i)$. The well-known problem with algorithmic verification is that the complexity of the language containment test in terms of the number of components is PSPACE-complete.

One heuristic for dealing with this complexity problem is *localization reduction*, wherein the system model P is reduced to a smaller model P' *relative to* the property T. For example, P' may be a composition of a subset of the P_i's. On account of the language intersection property, $\mathcal{L}(P') \subset \mathcal{L}(T) \Rightarrow \mathcal{L}(P) \subset \mathcal{L}(T)$. This reduction may be used in conjunction with other reductions such as symbolic verification, abstraction and compositional reasoning.

Localization reduction moreover guarantees the reverse implication, through an iterative algorithm. Assume that P is defined in terms of program variables, partitioned among the P_i's, and that T refers to these variables. Following the variable dependency graph rooted at T, guess a P' (say, start with the automaton defined by the empty set of variables, defining the universal language, or any other set of variables chosen by the designer). Variables in P' are the "active" variables of the model. Variables not in P', but upon which the variables of P' depend are "freed" (treated as primary inputs to P') and form a "fence" separating the active variables from all other variables, which are irrelevant to the behavior of P' and hence may be "pruned" (removed).



If $\mathcal{L}(P') \subset \mathcal{L}(T)$ fails, check if the counterexample may be extended to an element of $L(P)$ (computationally simpler than the language containment test). If yes, it's a counterexample to $\mathcal{L}(P) \subset \mathcal{L}(T)$. Otherwise, use the counterexample to find some P_i's which together are inconsistent with the counterexample, and compose them with P', to form a new candidate P'. Iterate.

This algorithm works well when the property T is *local* in the sense that it depends on only a small portion of P, so that the localization reduction terminates within the available space. Lacking this, the objective is to *decompose* T into a parallel composition, each of whose components is local. The next talk discusses this decomposition problem in the more general framework of *reduction hierarchies*.

A thorough discussion of the speaker's approach to verification can be found in [11].

2.18 Ken McMillan, Cadence Berkeley Labs

Design-by-refinement Using the SMV System

SMV is a tool for automatic formal hardware verification. It is based on a technique called "symbolic model checking," whereby a finite state model of a system is checked relative to a specification in temporal logic, without actually constructing the state graph of the model.

Using SMV, the hardware model to be verified is described in a simple equational language with nondeterministic choice. This language is supplemented with "syntactic sugar" that supports reusable modules, conditionals, parallel loops, defaults, arrays, binary arithmetic, and other constructs necessary for hardware descriptions. Specifications may be written in temporal logic, and verified (or falsified) automatically using symbolic "search" techniques. The transition relation of the model and various sets of states are represented indirectly using boolean formulas, which are in turn represented by binary decision diagrams, and other compact canonical forms for boolean functions.

This use of symbolic representations rather than direct state space search can greatly increase the size, in terms of states, of models that can be verified. However, this is not in itself sufficient to allow the verification of detailed descriptions of large systems. For this purpose, SMV provides two mechaninisms for "compositional" verification, in which properties proved about small parts of a system can be used to prove properties of the system as a whole.

The basic proof decomposition mechanism is "assume/guarantee" reasoning. In this method, one component of the system is shown to satisfy temporal property P. Property P is then used as an assumption when verifying that another component of the system satisfies property Q. Based on this general mechanism, the system supports a more structured style of decomposition. A language construct called "refinement" allows several levels of definition to be written for the same collection of signals. Each level of definition must logically imply the level above (in other words, it must refine the possible behaviors). This simple mechanism makes it possible to formally define interfaces between low-level components, and to define arbitrary refinement maps that relate signals at two levels of abstraction. Maps between levels of description can be created by simply introducing new signals for the lower level implementation, and defining these (nondeterministically) in terms of the level above.

Because refinement maps give low-level signals in terms of high-level signals, they can be used to decompose the verification problem into small, local verification problems. This is done using a specialized assume/guarantee rule that allows higher level definitions to be used as the "environment" when verifying the lower level definitions.

A mechanism for inheritance from high-level definitions to low-level definitions allows one level to be defined in terms of a set of modifications to the level above. This supports design by a sequence of small refinements. The final design will contain definitions of abstract signals which do not affect the actual system outputs, but only appear as part of the refinement process. Because of the structure of the language, however, these definitions may be removed without altering the observable behavior. Thus, no "hiding" mechanism is needed or provided in the language.

The use of refinement and inheritance is illustrated by the design of a simple RISC propcessor pipeline. The top level of description is a purely sequential implementation of the instruction set architecture, that executes one instruction at each step. In the next level of decription, the complete state of this model is delayed through a sequence of stages, simulating the operation of the pipeline without actually performing any operations. The lowest level of description refines this simulated pipeline by inserting actual operations into the pipeline stages.

This method provides two advantages. First, it allows the pipeline to be described in a clean a concise manner, as a set of local modifications to a very regular structure. This description is easily modified, for example, to move operations from one stage of the pipeline to another. Second, it allows verification of the pipeline to be localized – each stage of the pipeline can be verified separately, in context of the abstract instruction-set-architecture model. The abstractions used to prove these local properies are derived automatically from the structure of the definitions. This contrasts with other pipeline verification techniques, which are based on computing a global abstraction function by flushing all pending operations from the pipeline.

We also observe how the forwarding paths and interlocks in the pipeleine can be derived by a sequence of model checking "experiments," in which each model checking counterexample leads to the addition of a bypass or a stall in the pipeline. Because of the reduction in verification complexity obtained by the refinement method, these experiments require only about ten seconds to perform. The result is a convenient methodology for designing pipelines that insures that all pipeline hazards are handled correctly.

For more information about the SMV system, contact the speaker at `mcmillan@cadence.com`.

2.19 Doron Peled, Bell Laboratories

Overview of Partial-Order Verification

Concurrent systems verification can employ either a total order semantics, also called *interleaving semantics*, or a partial order semantics. Total order semantics views each execution of a concurrent system as a sequence of events, where actions executed concurrently appear according to some arbitrary order. Partial order semantics allows events to appear either ordered or unordered, disallowing causality cycles. Total order semantics are traditionally considered easier to work with, as they lend themselves to simple representations, *e.g.*, using finite state machines. Partial order semantics is more recent in modeling concurrent programs.

Until recently, using partial order semantics in verification was scarce, as it was considered to be more complicated. In recent years, new research showed that partial order semantics can be as simple to use as total order semantics, and in many cases partial order verification methods perform significantly better than total order based methods. This is based on the observation that execution sequences of a concurrent program can be grouped together into equivalence classes that are indistinguishable by the property to be checked. This class corresponds to a single partial order execution. Applying the reduction constructs a reduced state-space that generates at least one representative for each equivalence class. The reduction approach is also extended to branching specifications.

Partial order reduction techniques are used to alleviate the state-space explosion in automatically verifying concurrent programs [9, 6, 19, 13, 14, 8, 5]. The reduction is based on a modified depth-first-search, where at each searched state only a subset of the enabled transitions is selected. Selecting such a subset is based on simple which can differ according to the specification language, *e.g.*, for linear temporal logic [14] or for branching temporal logic [5].

Using the partial order reduction techniques, it has become possible to analyze problems of larger size, which did not lend themselves to automatic verification before. The simplicity of the principles behind these methods suggest that they can be integrated into any state-based automatic verification tool. The SPIN model-checking tool already employs partial order reduction [8].

2.20 Carl Pixley, Motorola

Case Studies in Formal Verification of Design Equivalence

Deciding design equivalence continues to be an important problem in industry. In a recent microprocessor design, fourteen "custom cells" required comparison. The Register Transfer Level (RTL) models were written at one design center and the schematic (actually, gate level) models were implemented at another design center. The two views were then brought together for comparison. None of the cells employed latches with hardware reset and none came equipped with an initial state or a synchronizing sequence. In each case, Motorola's Mapover Equivalence Tool (MET) was first used to attempt to determine equivalence. MET is a logic checker which requires an exact correspondence between storage elements to succeed. MET automatically deduces which storage elements need to be compared using a proprietary algorithm – patent pending. However, in the case of several custom cells, MET could not be used because the RTL and gate models did not have the same numbers of latches. For example, the RTL used edge-triggered flip-flops. Internally, flip-flops are composed of master and slave latches which are transparent on opposite phases of the clock. In some cases, the gate version of a custom cell separated the master latch from the slave latch and moved the slave across logic. Thus breaking the correspondence between storage elements needed for logic checking to succeed. Therefore, a sequential analysis and verification tool was needed to compare the designs.

Motorola's sequential analysis tool is Verdict, a CTL model checker based originally on Carnegie Mellon's SMV and the University of Colorado implementation. Verdict is similar to Berkeley's VIS program. Verdict takes Verilog input, has waveform output display, handles transparent latches, and multiphase clocks. Verdict was used to check Sequential Hardware Equivalence (SHE). SHE was reported in 1990 at the CAV workshop held at DIMACS [15, 16]. An implementation of CTL using Binary Decision Diagrams (BDDs) to represent transition relations and reachable sets and to compute fixed point algorithms was presented in this DIMACS paper.

Verdict, SMV and VIS implicitly evaluate CTL properties with respect to all initial states. In the case of sequential comparison, the product of two designs is formed by instantiating the two designs and identifying corresponding inputs. A pair of states of the two designs is equivalent if, starting from that pair of states, the designs have the same I/O behavior for valid input sequences. The CTL property $AG(outputs - RTL = outputs - Gate)$ is TRUE for a state pair if and only if the state pair is equivalent. To check for the existence of an Equivalent State Pair (ESP) the initial state set is declared to be *all* state pairs and the property $!AG(outputs - RTL = outputs - Gate)$ is evaluated. If the property is TRUE, then there are no equivalent states; if FALSE, then Verdict will print one equivalent state pair using its counterexample generation feature. If the two designs have equivalent states, the proposition $(EF\ AG(outputs - RTL = outputs - Gate))$ is then checked. If the EF-AG property is true then both design are resettable, if not, then at least one design cannot be reset [15].

When a design pair does not compare favorably with SHE, an initialization sequence $(in_1, in_2, ..., in_k)$ is provided by the design team. A very significant implementation enhancement was observed. There are two ways of checking initialization. In the first way, starting from all initial states, the following property is evaluated.

$$(in_1 \rightarrow AX(in_2 \rightarrow AX(\ldots AG(in_k \rightarrow AG(outputs - RTL = outputs - Gate)))))$$

A second method is to construct an initialization module as part of the design. The initialization module emits the input sequence to the design pair during the first k cycles of the design's operation after which a "ready" condition is asserted. Typically this initialization module employs a counter. A sequential verification tool could, in fact, generate this sequence as part of its reachability procedure. The property

$$(ready \rightarrow AG(outputs - RTL = outputs - Gate))$$

is checked from the initial state set in which the counter is 0 but state pairs are otherwise unconstrained. It was observed that the initializing model method evaluated CTL formulas more that an order of magnitude faster than the method of initializing within the CTL formula.

These methods are illustrated on two of the 14 designs: a register file in which each of the RTL and the Gate models have over 350 latches and a barrel shifter in which each of the RTL and gate had about 34 latches. Verdict's design scaling and constant propogation options were used. As a result, in both designs, input signal protocols were discovered; in the register file an output don't care was discovered; and a design bug was found in the barrel shifter.

More information can be obtained from the speaker at `pixley@math.sps.mot.com`.

2.21 Amir Pnueli, Weizmann Institute

Temporal Logic: Entering the Age of Integration

This presentation is intended as a tutorial of temporal logic, surveying its application to the specification, verification, systematic development, and synthesis of reactive systems. A point strongly emphasized in the talk is that the formal community should present its (potential) customers with a unified theory, containing specialized methods for special circumstances (rather than a collection of unrelated approaches).

To achieve this goal, it is necessary to integrate several of the existing approaches. In particular, we should integrate

1. Branching with Linear temporal logics.

2. Algorithmic (model checking) with deductive (theorem proving) verification methods.

3. Automata (including alternating) with the logical approaches and methods.

Consistently with this view, the logic we presented was CTL*, the most expressive propositional temporal logic, extended by first-order variables, predicates, functions, and rigid quantification. The models for this logic are Kripke structures extended with variables and their valuations over possibly infinite domains.

The survey intended to demonstrate that we already have all the tools necessary in order to perform deductive verification over this powerful logic, as well as to carry out algorithmic verification for the special cases that the extended Kripke's structure turns out to be finite state.

More information can be obtained from the speaker at `amir@wisdom.weizmann.ac.il`.

2.22 Stephen Ponzio, Hebrew University

Lower Bounds for Restricted Branching Programs

This talk presents a summary of the techniques for proving lower bounds on the size of restricted branching programs for explicit functions. By "restricted branching programs," we refer to OBDD's and their variants, defined below. Because of their restricted nature, these branching programs are easily manipulated and tested for equivalence, a useful property that makes them central to a very common technique for circuit verification. However, these restrictions also render them a weak model of computation – unable to compute in polynomial size functions such as multiplication – and we would like to understand their computational power.

A *branching program* is a *binary decision diagram (BDD)*; it is *oblivious* if every path reads the variables in the same order; it is *read-k* if on each path, each variable appears at most k times. Thus, an OBDD is an oblivious read-once branching program.

We begin with an example, describing completely Bryant's lower bound of $2^{n/8}$ on the size of OBDD's computing multiplication (specifically, the middle bit in the product of two n-bit integers). The argument is nicely viewed as a reduction to a problem of communication complexity between one player who knows the values of the variables in the top half of the OBDD, and another player who knows the rest of the variables. Given this analysis, it is easy to see that the same lower bound holds for various nondeterministic OBDD's: with OR nodes, with AND nodes, or with XOR nodes. Further, it is also easy to see that the communication complexity argument holds for "k-OBDD's," in which every path reads the variables k times and in the same order, which must be a single permutation repeated k times: While an OBDD of width w yields a one-way (one-round) communication protocol of $\lg w$ bits, a k-OBDD of width w yields a two-way k-round protocol of $2k \lg w$ bits. This argument can be carried even one step further, generalizing to any oblivious branching program of linear (kn) length, by using a lemma of Alon and Maass, obtaining a lower bound of $2^{\Omega(n)}$ for constant k, again even if the program is nondeterministic.

Having seen that oblivious programs are utterly incapable of computing multiplication, we turn our attention to *non-oblivious*, or simply read-once branching programs, sometimes called "free BDD's," where different paths may read the variables in different orders, but still only once each. Such programs have been well studied by the theory community; the techniques for proving lower bounds are different. We begin by describing a function due to Alon and Maass that best exhibits the severity of obliviousness: it has simple read-once programs but requires exponential-size oblivious programs without restrictions on the variable ordering, and even if their length may be superlinear, up to $o(n \lg n)$.

Though this indicates that non-oblivious programs may be much stronger, an exponential lower bound for multiplication also holds in this model. As time does not permit the presentation of this proof, we give instead a very nice proof of a lower bound for read-once programs that determine whether a 0,1-matrix is a permutation matrix. This proof gives some of the flavor of read-once lower bounds and we also note that this function is in AC^0, while in the other direction, there are functions that are easy for read-once programs (even for OBDD's) but *not* in AC^0 (parity, for example). In fact, the class of functions computable by OBDD's of polynomial size cannot be easily bounded within the usual complexity hierarchy, as there is a logspace-complete function which is easy for OBDD's.

We conclude with some well known open questions: Is every function computable by poly-size read-k programs also computable by poly-size read-$(k-1)$ programs ($k > 2$ a constant)? Define *semantic* read-k programs to be those in which every *consistent* path (not containing both a variable and its negation) contains each variable at most k times. Give an explicit function that requires exponential-size *semantic* read-twice programs.

For more information, including an extensive list of references, see the speaker's PhD thesis, which can be found at `http://www.cs.huji.ac.il/ ponzio/`.

2.23 Rajeev Ranjan, University of California at Berkeley

Exploiting Memory Hierarchy for High Performance BDD Manipulation

The success of binary decision diagram (BDD) based algorithms for synthesis and/or verification depend on the availability of a high performance package to manipulate very large BDDs. State-of-the-art BDD packages, based on the conventional depth-first technique, limit the size of the BDDs due to a disorderly memory access pattern that results in unacceptably high elapsed time when the BDD size exceeds the main memory capacity. We present a high performance BDD package that enables manipulation of very large BDDs by using an iterative breadth-first technique directed towards localizing the memory accesses to exploit the memory system hierarchy. Our novel contributions are:

- data structures and memory management techniques to preserve the locality of reference,

- a technique to perform multiple, independent BDD operations simultaneously (*superscalarity*), and
- a technique to perform multiple BDD operations even when the operands of some BDD operations are the result of some other BDD operations yet to be completed (*pipelining*).

Superscalarity and pipelining are targeted towards deriving higher performance from the memory system hierarchy by exploiting the locality of reference in the memory access pattern across several BDD operations. To our knowledge, this is the first complete BDD package incorporating a comprehensive set of BDD algorithms designed around obtaining higher performance by targeting the memory system hierarchy. We achieve high performance on large BDDs without sacrificing performance on small and medium sized BDDs. In fact, our performance exceeds that of a state-of-the-art depth-first package even for many BDDs that fit within the main memory. For BDD sizes that do not fit within the main memory, a performance improvement of more than a factor of 150 is achieved.

This is joint work with Jagesh V. Sanghavi, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. More information can be found on the web page
`http://www-cad.eecs.berkeley.edu/Respep/Research/cal_bdd`.

2.24 Ronitt Rubinfeld, Cornell University

Robust Functional Equations with Applications to Program Testing and Correcting

Given a functional equation, such as $\forall x, y\ f(x)+f(y) = f(x+y)$, we study the following general question: When can the "for all" quantifiers be replaced by "for most" quantifiers without essentially changing the functions that are characterized by the property? When "for most" quantifiers are sufficient, we say that the functional equation is robust. Examples of robust functional equations have been given in several preceding works. We initiate a general study aimed at characterizing properties of functional equations that determine whether or not they are robust. We then study a general class of functional equations, which are of the form $\forall x, y\ \ F[f(x-y), f(x+y), f(x), f(y)] = 0$, where F is an algebraic function. We give conditions on such functional equations that imply robustness.

Our results have applications to the area of self-testing/correcting programs — we provide results which show that the concept of self-testing/correcting has much broader applications than we previously understood. We show that self-testers and self-correctors can be found for many functions satisfying robust functional equations, including algebraic functions of trigonometric functions such as $\tan x$, $\frac{1}{1+\cot x}$, $\frac{Ax}{1-Ax}$, $\cosh x$.

For more information, contact the speaker at `ronitt@cs.cornell.edu`.

2.25 Natarajan Shankar, SRI International

The Integration of Model Checking with Theorem Proving in PVS*

Automated proof checking tools have traditionally been employed for the verification of digital systems. These tools have the advantage of being completely general but suffer from the inherent limits to the efficient automation of expressive logics. By constraining the expressiveness, one can define certain useful fragments of logic for which efficient decision procedures can be found. The model checking paradigm yields an important class of decision procedures for establishing temporal properties of finite-state machines. Model checking is effective for automatically verifying finite automata with relatively small state spaces, but it is inadequate when the state spaces are either too large or unbounded. For this reason, it is useful to integrate the complementary technologies of model-checking and proof checking. Such an integration has to be carried out in a delicate manner in order to be more than just the sum of the techniques. We describe an approach for such an integration where a BDD-based checker for Park's μ-calculus has been used as a decision procedure within the framework of the PVS proof checker [17]. We argue that our approach fits in nicely with the design philosophy of PVS where highly efficient decision procedures are used as the workhorses of an interactive proof checker. The utility of the combination of the two techniques is illustrated by an example of an N-process mutual exclusion protocol that is proved using induction and model checking. A second example demonstrates the application of abstraction and model checking in the verification of a communication protocol, where theorem proving is used to ensure that the abstraction preserves the correctness property [7].

This is joint work with S. Rajan, M. Srivas, and K. Havelund. More information can be found on the speaker's homepage: `http://www.csl.sri.com/"shankar/shankar.html`.

2.26 Thomas Shiple, University of California at Berkeley

VIS : A System for Verification and Synthesis

VIS (Verification Interacting with Synthesis) is a tool that integrates the verification, simulation, and synthesis of finite-state hardware systems. We briefly describe the salient features of VIS.

Verilog front end VIS operates on an intermediate format called BLIF-MV, which is an extension of BLIF, the intermediate format for logic synthesis. VIS includes a stand-alone compiler from Verilog to BLIF-MV, called VL2MV, which supports a synthesizable subset of Verilog. VL2MV extracts a set of interacting finite state machines that preserves the behavior of the source Verilog program defined in terms of simulated results. We have extended the syntax of Verilog to allow the specification of nondeterminism, and symbolic variables.

Hierarchy and initialization When a BLIF-MV description is read into VIS, it is stored hierarchically as a tree of modules, which in turn consist of sub-modules. This hierarchy can be traversed in a manner similar to traversing directories in UNIX. Simulation and verification operations can be performed at any subtree of the hierarchy. It is possible to replace the subhierarchy rooted at the current node with a new hierarchy specified by a new BLIF-MV file, which might be a synthesized module or a manually abstracted module.

Interaction with synthesis VIS can interact with SIS to optimize the existing logic by reading and writing the BLIF format, which SIS recognizes. Synthesis can be performed on any node of the hierarchy.

Abstraction Manual abstraction can be performed by giving a file containing the names of variables to abstract. For each variable appearing in the file, a new primary input node is created to drive all the nodes that were previously driven by the variable. Abstracting a net effectively allows it to take any value in its range, at every clock cycle.

Fair CTL model checking and language emptiness check VIS performs fair CTL model checking under Büchi fairness constraints. In addition, VIS can perform language emptiness checking by model checking the formula $EG\ \ true$. The language of a design is given by sequences over the set of reachable states that do not violate the fairness constraint. The language emptiness check can be used to perform language containment by expressing the set of bad behaviors as another component of the system. If model checking or language emptiness fail, VIS reports the failure with a counterexample.

Equivalence checking VIS provides the capability to check the combinational equivalence of two designs. An important usage of combinational equivalence is to provide a sanity check when re-synthesizing portions of a network. VIS also provides the capability to test the sequential equivalence of two designs. Sequential verification is done by building the product finite state machine, and checking whether a state where the values of two corresponding outputs differ, can be reached from the set of initial states of the product machine. If this happens, a debug trace is provided. Both combinational and sequential verification are implemented using BDD-based routines.

Simulation VIS also provides traditional design verification in the form of a cycle-based simulator that uses BDD techniques. Since VIS performs both formal verification and simulation using the same data structures, consistency between them is ensured. VIS can generate random input patterns or accept user-specified input patterns. Any subtree of the specified hierarchy may be simulated.

More information can be found on the web page
`http://www-cad.eecs.berkeley.edu/Respep/Research/vis`.

2.27 Henry Sipma, Stanford University

STeP and its Applications

The Stanford Temporal Prover, STeP, supports the computer-aided formal verification of reactive, real time (and, in particular, concurrent) systems based on temporal specifications. Reactive systems maintain an ongoing interaction with their environment; their specifications are typically expressed as constraints on their behavior over time. Unlike most model checking systems for temporal verification, STeP is not restricted to finite-state systems, but combines model checking with deductive methods to allow the verification of a broad class of systems, including parameterized (N-component) circuit designs, parameterized (N-process) programs, and programs with infinite data domains.

The deductive methods of STeP verify temporal properties of systems by means of verification rules and verification diagrams. *Verification rules* are used to reduce temporal properties of systems to first-order verification conditions. *Verification diagrams* provide a visual language for guiding, organizing, and displaying proofs. Verification diagrams allow the user to construct proofs hierarchically, starting from a high-level, intuitive proof sketch and proceeding incrementally, as necessary, through layers of greater detail.

The system implements powerful techniques for automatic *invariant generation* Deductive verification almost always relies on finding, for a given program and specification, suitably strong (inductive) invariants and intermediate assertions. The user can typically provide an intuitive, high-level invariant, from which the system derives stronger, more detailed, *top-down invariants*. Simultaneously, *bottom-up invariants* are generated automatically by analyzing the program text. By combining these two methods, the system can often deduce sufficiently detailed invariants to carry through the entire verification process.

The system also provides an integrated suite of simplification and decision procedures for automatically checking the validity of a large class of first-order and temporal formulas. This degree of automated deduction is sufficient to handle most of the verification conditions that arise in deductive verification.

An educational version of STeP is available. The distribution includes a comprehensive user manual and a tutorial, as well as 40 example programs with specifications from the textbook that are ready to be loaded. For many programs, ready-to-load verification diagrams have been included as well. STeP is implemented in Standard ML of New Jersey, using CML and eXene for its X-windows user interface. For information on obtaining the system, send e-mail to `step-request@cs.stanford.edu`. This is joint work with Nikolaj Bjørner, Anca Browne, Eddie Chang, Michael Colón, Arjun Kapur, Zohar Manna, and Tomás Uribe.

2.28 A. Prasad Sistla, University of Illinois at Chicago

Symmetry and Model Checking

Over the last decade much research has been conducted on using temporal logic model checking for automatic verification of finite state concurrent programs. In this approach, the global state graph of a concurrent program is constructed (explicitly or implicitly) and then its correctness against a temporal logic specification is checked using efficient algorithms. The main bottleneck for this approach is the state explosion problem, *i.e.*, the rapid explosion (exponential) in the number of states to be explored. In this talk we will present an approach, that exploits symmetries in the system, for containing the state explosion problem. These symmetries induce an equivalence relation on the state space. Our approach constructs a quotient structure corresponding to these equivalence relations and does modelchecking on the quotient structure. In order to make this approach work, we need to consider the symmetries in the temporal specification also. In the talk, we present various results relating the correctness of a temporal formula in the original structure to its correctness in the quotient structure. We also present an automata-theoretic approach for this problem.

The research on symmetries is joint work with Allen Emerson. For more information, contact the speaker at `sistla@surya.eecs.uic.edu`.

2.29 Fabio Somenzi, University of Colorado at Boulder

Reordering Variables in Decision Diagrams

The size of a Binary Decision Diagram (BDD) depends, sometimes dramatically, on the order in which the variables are tested along the paths of the diagram. The success of BDDs in solving seemingly intractable problems has brought considerable attention to the problem of determining good orders. Optimum orders can be found by dynamic programming, but the approach is only viable for BDDs with 20 variables or less. Much of the research has therefore concentrated on heuristic techniques. Some of these try to guess a good order before the BDDs are built. Other techniques try to reduce the size of an existing BDD by gradually changing the variable order. The latter techniques go under the name of "dynamic reordering," and have been so successful as to become an integral part of almost every package for the manipulation of BDDs.

Dynamic reordering techniques employ a variety of local search schemes. The most effective do date is called sifting and is due to R. Rudell. In sifting a variable is moved up and down in the order until it finds a locally optimal position. Another variable is then chosen, and the process repeated until some stopping criterion is met. The efficient implementation of sifting relies on the ability to swap two adjacent variables in a BDD in time that depends only on the number of nodes involved in the swap. The use of reference counts and of a hash table organized by variable make the efficient swap possible. With minor modifications, sifting works for most known variants of ordered decision diagrams.

Sifting performs a number of swaps that tends to grow quadratically with the number of variables. Several devices are employed to keep that number low and replace costly swaps with cheaper operations. For instance, it is possible to keep an "interaction matrix" that says whether there may be arcs joining nodes of the two variables being swapped. If that is the case, the swap can be performed in constant time. Also, lower bounds on the size of the BDD can be computed, so that sifting is terminated when the probability of improvement is null or low.

Sifting is based on the idea that each variable can find a good position in the order that is more or less independent of the positions of the other variables. In some cases, however, this assumption does not hold. For instance, symmetric variables tend to strongly "attract" each other. Sifting cannot move a badly placed group of symmetric variables, because it tries to move one variable at the time. To overcome this limitation, group sifting allows more than one variable to be moved at one time. Heuristics aggregation criteria are used to identify what variables should be joined into a group.

Sifting can be used to do more than just variable reordering. For instance, the most efficient algorithm for symmetry detection known to date is based on sifting. Moreover, Hett, Drechsler, and Becker have recently proposed the use of reordering techniques as a general paradigm for BDD manipulation. Though the experimental results reported so far are not conclusive, the new method has several attractive features, including the ability to perform in-place manipulation of BDDs, and the ability to combine several operations into one.

For more information, see [12, 18].

2.30 Moshe Vardi, Rice University

Verification = Logic + Algorithmics

We describe an automata-theoretic approach to the automatic verification of finite-state programs. The basic idea underlying this approach is that, for any linear temporal formula, we can construct an alternating automaton on infinite words that accepts precisely the computations that satisfy the formula. The simple combinatorial structures that emerge from the automata-theoretic approach decouple the logical and algorithmic components of finite-state-program verification and yield clear and general verification algorithms.

For more information, contact the speaker at `vardi@cs.rice.edu`.

2.31 Hal Wasserman, University of California at Berkeley

Checking the Fourier Transform: A Historical Update

In collaboration with Hughes Aircraft, we are embedding "simple checkers" in real-world software. These checkers can then serve as extremely reliable testing oracles and may have other uses as well. A central problem in this research has been that of checking the Fourier Transform.

A sequel to [3], the current talk will give an informal narrative of our recent work on this problem. The talk may include:

1. A review of the real-number Fourier Transform checker we introduced in [3].
2. Comparison with the real-number matrix-multiplication checker of [1].
3. Our struggle with a seemingly trivial probability question which caused difficulties in both [3] and [1]. Including: a solution proposed by Joel Spencer; an alternative approach developed in our revised version of [3].
4. The recent implementation of our checker by David Shreve at Hughes Aircraft.

This is joint work with Manuel Blum. For more information, see [1, 3, 4].

2.32 Pierre Wolper, University of Liege

Partial-Order Methods or How Independence Defeats Complexity

Partial-Order approaches make it possible to check meaningful properties of a state space while only considering a limited number of interleavings of independent concurrent actions, and hence avoiding much of the wastefulness of representing concurrency by interleaving. In practice they work by doing a state-space search in which only selected transitions are explored from each state. This talk first presents two central techniques for choosing the set of explored transitions: persistent sets and sleep sets. It then turns to the problem of using partial-order methods in the context of temporal logic model checking. After outlining the main results for linear-time model checking, it describes a simple argument that shows how these results can be lifted to obtain partial-order algorithms for branching-time model checking.

For more information, contact the speaker at `pw@montefiore.ulg.ac.be`.

2.33 Xudong Zhao, Carnegie Mellon University and Intel

Word-Level Model Checking

Proving the correctness of arithmetic operations has always been an important problem. The importance of this problem has been recently emphasized by the highly-publicized division error in the Pentium. Although symbolic model checking techniques based on Binary Decision Diagrams (BDDs) have been successful in verifying control logic, these techniques cannot be directly used for verifying arithmetic circuits.

One of the main reasons that the symbolic model checking systems cannot handle arithmetic circuits is the lack of a concise representation for expressions that involve words with integer values. We have used hybrid decision diagrams to represent the integer functions that occur in the arithmetic circuit verification. For the state variables corresponding to data bits, our representation behaves like a BMD while for the state variables corresponding to control signals, it behaves like an MTBDD. By using this representation, we are able to handle circuits with both control logic and wide data paths. We have extended the symbolic model checking system SMV so that it can also handle properties involving relationships among data words. In the original SMV system, atomic formulas could only contain state variables. In the extended system, we allow atomic formulas to be equations or inequalities between expressions as well. These expressions are represented as hybrid decision diagrams.

In the word-level model checking system, propositions denoting nodes in circuits are represented as BDDs and are computed in exactly the same way as in the original symbolic model checking system. Words are arrays of propositions, each of which corresponds to a single bit. Expressions are composed of arithmetic operations applied to words. Hybrid decision diagrams can be computed for words and expressions using the algorithms for arithmetic operations. Atomic formulas can be relations between expressions, and their BDD representations can be computed by the algorithm that handles arithmetic relations. After the BDD representations for the atomic formulas are generated, the BDDs for static formulas and temporal formulas are computed in the same way as in ordinary model checking. In particular, the fixpoint computations are exactly the same in both cases.

As an example to illustrate the power of these verification techniques, we verify a division circuit that implements the IEEE floating point standard. The circuit uses a radix 4 SRT division algorithm that is similar to the one used in the Pentium. The algorithm looks ahead to find the next quotient digit in parallel with the generation of next partial remainder. An 8-bit ALU estimates the next remainder's leading bits. A quotient digit look-up table generates the next quotient digit depending on the leading bits of the estimated remainder and the leading bits of the divisor.

By using the word-level model checking system, we have successfully verified the division circuit. We have been able to handle both the control logic and the data paths. All states in the finite state machine for the control logic have been verified. Moreover, we have proved invariant properties that guarantee the correctness of the data values and ensure that overflow do not occur. The total number of state variables exceeds 600 (which is, to our knowledge, much larger than any circuit previously checked by SMV).

This is joint work with E. Clarke and M. Khaira. For more information, contact the speaker at `Xudong.Zhao@cs.cmu.edu`.

References

[1] S. Ar, M. Blum, B. Codenotti, and P. Gemmell, "Checking Approximate Computations over the Reals," in *Proc. of the 25th ACM Symposium on Theory of Computing*, 786–795, 1993.

[2] A. Aziz, R. K. Brayton, F. Balarin and A. L. Sangiovanni-Vincentelli, "Sequential Synthesis Using `S1S`," in *Proc. Intl. Conf. on Computer-Aided Design*, Santa Clara CA, November 1995.

[3] M. Blum and H. Wasserman, "Program Result-Checking: a Theory of Testing Meets a Test of Theory," in *Proc. of the 35th IEEE Symposium on Foundations of Computer Science*, 382–392, 1994. Revised version: "Software Reliability via Runtime Result-Checking," `http://http.cs.berkeley.edu/ halw/`.

[4] M. Blum and H. Wasserman, "Reflections on the Pentium Division Bug," `http://http.cs.berkeley.edu/ halw/`. To appear in *IEEE Transactions on Computers.*

[5] R. Gerth, R. Kuiper, D. Peled, and W. Penczek, "A Partial Order Approach to Branching Time Logic Model Checking," in *Proc. of the 3rd Israel Symposium on Theory on Computing and Systems*, 130-139, IEEE, 1995, Tel Aviv, Israel.

[6] P. Godefroid and P. Wolper, "A partial approach to model checking," in *Proc. of the 6th Annual IEEE symposium on Logic in Computer Science*, Amsterdam, 406–414.

[7] K. Havelund and N. Shankar, "Experiments in theorem proving and model checking for protocol verification," in *Proc. of the 1996 Formal Methods Europe conference*, LNCS 1051, Springer-Verlag, 662–681, 1996, Oxford, UK.

[8] G. Holzmann and D. Peled, "An Improvement in Formal Verification," *FORTE'94, Formal Description Techniques 1994*, Chapman & Hall, 197-211, 1994, Bern, Switzerland.

[9] S. Katz and D. Peled, "Verification of Distributed Programs Using Representative Interleaving Sequences," *Distributed Computing*, 6 (1992), 107–120.

[10] R. P. Kurshan, "The Complexity of Verification," in *Proc. of the 26th ACM Symposium on Theory of Computing*, 365–371, 1994.

[11] R. P. Kurshan, **Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach**, Princeton University Press, Princeton NJ, 1994.

[12] S. Panda and F. Somenzi, "Who are the variables in your neighborhood," in *Proc. of the 1995 International Conference on Computer-Aided Design*, 74–77, 1995, San Jose, CA.

[13] D. Peled, "All from One, One from All: on Model Checking using representatives, " in *Proc. of the 1993 CAV conference*, LNCS 697, Springer-Verlag, 409–423, 1993, Elounda Crete, Greece.

[14] D. Peled, "Combining Partial Order Reductions with On-the-fly Model-Checking," *Journal of Formal Methods in Systems Design*, 8 (1996), pp. 39–64.

[15] C. Pixley, "A Computational Theory and Implementation of Sequential Hardware Equivalence," in *Proc. of the 1990 CAV conference*, DIMACS Series on Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 293–320, 1990, New Brunswick, NJ.

[16] C. Pixley, "A Theory and Implementation of Sequential Hardware Equivalence," *IEEE Transactions on Computer-Aided Design*, 11 (1992), 1469–1478.

[17] S. Rajan, N. Shankar, and M.K. Srivas, "An integration of model-checking with automated proof checking," in *Proc. of the 1995 CAV conference*, LNCS 939, Springer-Verlag, 84–97, 1995, Liege, Belgium.

[18] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *Proc. of the 1993 International Conference on Computer-Aided Design*, 42–47, 1993, Santa Clara, CA.

[19] A. Valmari, "A stubborn attack on state explosion," *Journal of Formal Methods in System Design*, 1 (1992), 297–322.