Model Checking: Theory into Practice

E. Allen Emerson*

Department of Computer Sciences and Computer Engineering Research Center
The University of Texas at Austin, Austin TX-78712, USA
emerson@cs.utexas.edu
http://www.cs.utexas.edu/users/emerson/

Abstract. Model checking is an automatic method for verifying correctness of reactive programs. Originally proposed as part of the dissertation work of the author, model checking is based on efficient algorithms searching for the presence or absence of temporal patterns. In fact, model checking rests on a theoretical foundation of basic principles from modal logic, lattice theory, as well as automata theory that permits program reasoning to be completely automated in principle and highly automated in practice. Because of this automation, the practice of model checking is nowadays well-developed, and the range of successful applications is growing. Model checking is used by most major hardware manufacturers to verify microprocessor circuits, while there have been promising advances in its use in software verification as well. The key obstacle to applicability of model checking is, of course, the state explosion problem. This paper discusses part of our ongoing research program to limit state explosion. The relation of theory to practice is also discussed.

1 Introduction

There is a chronic need for more effective methods of constructing correct and reliable computer software as well as hardware. This need is especially pressing for concurrent, distributed, real-time, or, more generally, *reactive systems*, which can exhibit, unpredictably, any one of an immense set of possible ongoing, ideally infinite, behaviors. Many safety critical and economically essential real-world applications are reactive systems. Examples include: computer operating systems, computer network communication protocols, on-board avionics control systems, microprocessors, and even the internet.

The traditional approach to program verification involves the use of axioms, and inference rules, together with hand proofs of correctness. Because of the inherent difficulty and sheer tediousness of manual proof construction, this manual, *proof-theoretic* strategy has, by-and-large, fallen from favor. While this approach has introduced fundamental principles such as coherent design and compositional proofs, the evidence suggests that it does not scale to real systems.

* This work was supported in part by NSF grant CCR-980-4736 and TARP project 003658-0650-1999.

We have developed an alternative *model-theoretic* strategy for program reasoning (cf. [7]). The informal motivation is that, while manual proofs are not feasible, it ought to be possible to devise fully automated design and reasoning methods using the basic model theory of modal temporal logic. Temporal logic, a tensed form of modal logic, has been shown to provide a most useful approach to specifying and reasoning about the complex and ongoing behavior reactive systems [27]. The linear temporal logic LTL provides modalities corresponding to natural language tenses such as Fp (eventually p), Gp (henceforth p), Xp (nexttime p), and pUq (eventually q and p until then) that are well-suited to describing ongoing behavior along a (discrete) timeline corresponding to an execution sequence of a reactive program. Additional expressiveness is gained through use of the path quantifiers A (for all futures) and E (for some future) in the branching time logics CTL and CTL* (cf. [9]).

The automation of model-theoretic reasoning is permitted by the fact that core decision problems for temporal and classical modal logic are decidable. This provides a means, in principle, of letting programs reason about programs. Thus, it is significant that the following key problems associated with (propositional) temporal logic are decidable, in some cases efficiently. First, there is the *satisfiability* problem: given temporal logic specification f, does it have a model M? Second, there is the *model checking* problem: Is candidate Kripke structure model M a genuine model of temporal specification f?. Satisfiability asks if f is realizable in *some* model, and is useful in automatic program synthesis. Model checking decides if f is true in a *particular* model, and caters for automatic program verification.

We introduced model checking as an algorithmic method of verifying correctness of finite state reactive programs [7] (cf. [4], [6], [30]). Model checking has turned out to be one of the more useful approaches to verification of reactive systems, encompassing not only finite state but some finitely represented, infinite state systems as well. Our original approach is based on fixpoint computation algorithms to efficiently determine if a given finite state transition graph defines a model of correctness specification given in the temporal logic CTL. An advantage of this algorithmic approach is that it caters for both verification and debugging. The latter is particularly valuable in the early stages of systems development when errors predominate and is widely used in industrial applications. Symbolic model checking, which equals our original fixpoint based model checking algorithm plus the data structures for symbolic state graph representation using binary decision diagrams [3] (cf. [24]), is now a standard industrial tool for hardware verification.

The remainder of the paper is organized as follows. In section 2 we describe some recent technical advances in limiting state explosion. In section 3 we discuss factors relevant to the transition of model checking from theory into practice. Some closing remarks are given in section 4.

2 Limiting State Explosion

The most common calls from industrial users of formal verification tools are for (a) increased capacity of the tool to handle really large programs/designs, and for (b) increased automation. Model checking is especially popular in industrial usage because it is fully automated in principle. However, model checking's capacity is still limited. Model checking's capacity and utility are limited primarily by space complexity, and secondarily by time complexity. It is important that (the representation of) the state graph of the program being verified fit within the main memory of the computer on which the model checking tool is running, so as to avoid the slow down of many orders of magnitude which occurs when paging to disk. Of course, the underlying theoretical vexation is the well-known problem of combinatorial *state explosion*: e.g., if each sequential process of a parallel system has just 10 local states, a system with 1000 processes, could have as many as 10^{1000} global system states. A better handle on state explosion is essential to increasing the capacity of model checking tools. Thus, a good deal of effort has been devoted to limiting state explosion and making model checking more space efficient.

The technique of *abstraction* is central to the effective management of state explosion. In general terms, abstraction means the replacement of an intractably large system by a much smaller, abstract system through suppression of inessential detail and elimination of redundant information. The abstracted system should still preserve relevant information about the original system, and a desirable attribute is that the abstracted system should be equivalent in some appropriate sense to the original system. In the sequel, we will describe some of our work on abstraction utilizing the regularity of structure present in many reactive systems composed of multiple similar subcomponents, and mention some related open problems.

2.1 Symmetry

Symmetry Quotient Reduction. One useful approach to abstraction exploits the symmetry inherent in many concurrent systems composed of multiple interchangeable subprocesses or subcomponents. Using symmetry we have been able to verify a resource controller system with 150 processes and about 10^{47} states in roughly an hour on a Sparc. We developed a "group-theoretic" approach to symmetry reduction in [14]. The global state transition graph M of such systems often exhibits a great deal of symmetry, characterized by the the group of graph automorphisms of M. The basic idea is to reduce model checking over the original, intractably large structure M to model checking over the smaller quotient structure $\overline{M}$, where symmetric states are identified. The quotient graph can be exponentially smaller. Technically, let $\mathcal{G}$ be any group contained in $Aut\ M \cap Auto\ f$, where $Aut\ M$ is the group of permutations of process indices defining automorphisms of M and $Auto\ f$ is the group of permutations that leave f and crucial subformulas thereof invariant. We define an equivalence relation on states of M so that $s \equiv_{\mathcal{G}} t$ iff $t = \pi(s)$ for some permutation $\pi \in \mathcal{G}$.

The quotient $\overline{M}$ is obtained by identifying states that are $\equiv_{\mathcal{G}}$-equivalent. Then, because $\mathcal{G}$ respects both the symmetry of the structure M and the ("internal") symmetry of the CTL* specification formula f, we can show $M, s \models f$ iff $\overline{M}, \bar{s} \models f$, where $\bar{s}$ represents the equivalence class $[s]$ of states symmetric to s. Since it turns out, in many practical cases, to be both easy and efficient to construct the quotient, we have reduced the problem of model checking CTL* properties over an intractably large structure to model checking over a relatively small structure, with the proviso that the symmetry of the structure is appropriately respected by the specification. Other work on symmetry may be found in, e.g., [21], [19], [5].

Annotated Symmetry Quotient Reductions. One can also "trade group theory for automata theory". In [15] we introduced a powerful alternative method. This method is also more uniform in that it permits use of a *single* annotated quotient $\overline{M} = M/Aut\ M$ for model checking for *any* specification f, without computing and intersecting with $Auto\ f$. The idea is to augment the quotient with "guides", indicating how coordinates are permuted from one state to the next in the quotient. An automaton for f designed to run over paths through M, can be modified into another automaton run over $\overline{M}$ using the guides to keep track of shifting coordinates. This automata-theoretic method is much more more expressive than the purely "group-theoretic approach" above. In particular, we can show that the automata-theoretic approach makes it possible to efficiently handle reasoning under fairness assumption, unlike the purely "group-theoretic" approach above. Another augmented "model-theoretic" approach is introduced in [16], accommodating both fairness and discrete real-time in the uniform framework of the Mu-calculus.

Open problems related to quotient symmetry reduction. Despite the expressive power of annotated symmetry quotient reduction, efficiently handling of general specifications in which all process indices participate along a path still seems a difficult one. It would be interesting, and useful for hardware verification, to see if the work on fair scheduling (e.g., $GFex_1 \wedge \ldots GFex_n$) could be adapted to efficiently handle, e.g., the tighter requirement of round-robin scheduling ($(ex_1; ex_2; \ldots ex_n)^\omega)$) of an otherwise fully symmetric system. Such a composite seems to possess only cyclic symmetry, thereby limiting compression.

Simple Symmetry. Another important type of syntactic reasoning based on symmetry we call *simple symmetry* (originally dubbed state symmetry in [14]). It simplifies reasoning based on the symmetry of the specification, the structure, and individual states, and does not entail calculating a quotient structure. For instance, with an appropriately symmetric start state and with appropriate global symmetry, an *individual* process (or component) is correct iff *all* processes are correct. It permits "quantifier inflation" and "quantifier reduction": $M, s \models f_1$ iff $M, s \models \bigwedge_i f_i$, provided all f_j are identical up to re-indexing. Quantifier inflation via simple symmetry was used in [26] to facilitate verification of memory arrays, and more recently to radically reduce case analysis in [25]. In [16] we provided some powerful instances of the application of simple symmetry. Model checking $E(FP_1 \wedge \ldots \wedge FP_n)$ is an NP-complete problem, apparently requiring time exponential in n in the worst case. We use simple symmetry to show that for a system that (a) is fully symmetric; (b) has fully symmetric start state s_0; and (c) is already known to be *re-settable* so that $AGEFs_0$ holds, the above property then amounts to EFP_1, which is in polynomial time. This can be quite useful, since many systems do possess such symmetry and re-settability is a common requirement for many hardware circuits and embedded systems anyway. Many more useful applications of simple symmetry are possible.

Approximate Symmetry. In conversations we have had with industrial hardware engineers, it comes out that while symmetry reduction is often applicable due to the presence of many similar subcomponents, there are also many instances where it is not — quite — applicable. That is, the systems are not genuinely symmetric but "approximately" symmetric, for example, because of one different component or slight differences among all components. This limits the scope of utility of symmetry reduction techniques.

In [17] and [10] we proposed and formalized three progressively "looser" notions of approximate symmetry: *near* symmetry, *rough* symmetry, and, finally, *virtually* symmetry. Each can be applied to do group-theoretic quotient reduction to various asymmetric systems. The correspondence established in each case between the original large structure and the small quotient structure is exact, a bisimulation (up to permutation), in fact. Near symmetry can permit symmetry reduction on systems comprised of 2 similar but not identical processes. Rough symmetry can accommodate, e.g., for fixed $k \geq 2$, systems with k similar but not identical process, prioritized statically. Rough symmetry can handle the readers writers problem for k readers and ℓ writers. Virtual symmetry is yet more general and can accommodate dynamically varying priorities.

Additional open problems related to symmetry. There are number of important unsolved problems here. One is to broaden the scope of approximate symmetry reduction as much as possible. Obviously, a system that is "absolutely" asymmetric is going to lack sufficient regularity and redundancy to permit identification of approximate symmetries for reduction. Finding a sufficiently broad notion of approximate symmetry that is flexible enough to cover every useful applications is open, and perhaps an inherently ill-posed problem. But a practical solution is highly desirable. One possibility is to formulate a very general notion of approximate symmetry parameterized by degree of divergence from actual symmetry. We might look for a notion that is universally applicable in principle, but which (a) may in general result in a collapsed graph that is only loosely corresponds, say by a conservative or liberal abstraction, to the original structure; but (b) which progresses toward exact abstraction as the system approaches genuine symmetry. Our previous work with virtual general symmetry suggests a measure based on the number of "missing" arcs. Another technical problem that would have significant practical ramifications would be to extend approximate symmetry to annotated quotient structures (cf. [15], [16]).

2.2 Parameterized Verification

Symmetry quotient reductions, and many other abstraction methods, address reasoning about systems with k processes for a possibly large natural number constant k. It is often more desirable to reason about systems comprised of n homogeneous processes, where n is a natural number parameter. This gives rise to the *Parameterized Model Checking Problem (PMCP)*: decide whether a temporal property is true for all (sufficiently large) size instances n of a given system. The advantage is that one application of PMCP settles all cases of interest; there is then no need to be concerned whether a system known to be correct for 100 processes might fail for 101 processes or 1000 processes. In general, PMCP is undecidable (cf. [AK86]). But because of its practical importance many researchers have addressed this problem, obtaining interesting results and partial solutions (cf., e.g., [23], [2], [18], [22]). But most of these have potentially serious limitations, e.g., they require human assistance ("process invariants", etc.), are only partially automated (may not terminate), are sound but not complete, or lack a well-defined domain of applicability.

Parameterized Synchronous Systems. In [12] we formulated an algorithm for determining if a parameterized synchronous system satisfies a specification. The method is based on forming a single finite *abstract graph* (cf. [Lu84]) which encodes the behavior of concrete systems of all sizes n. An abstract state $\overline{s}$ records for a state s, which process locations are occupied. The theory we developed in [12] we later applied in [13] to verify correctness of the Society of Automotive Engineers SAE-J1850 protocol. This protocol operates along a single wire bus in (Ford) automobiles, and coordinates the interactions of (Motorola) micro-controllers distributed among the brake units, the airbags, the engine, etc. The general goal is to verify parameterized correctness so that the bus operates correctly no matter how many units are installed on the bus. (Thus, the same bus architecture could be used in small cars and large trucks.) The specific property we verified, by using a meta-tool built on-top of SMV, was that higher priority messages could not be overtaken by lower priority messages.

Open problem related to parameterized synchronous systems. A restriction on the mathematical model used in [12] is that conventional mutual exclusion algorithms cannot be implemented in it (cf. [18]), roughly because a system with 1 processes in a local state is, by definition of the abstract graph, indistinguishable from one with strictly more than 1 processes in that local state. It would be desirable to overcome this limitation. One possibility is to refine the abstract graph to distinguish between exactly 1 vs. 2 vs. 3 or more processes in an abstract state, to accommodate mutual exclusion. Or perhaps this method could be adjoined with use of a signal token as in [11], possession of which arbitrates among competing processes.

Reducing Many Processes to Few. In recent work [EK00] we give a fully automatic (algorithmic), sound and complete solution to PMCP in a rather broad framework. We consider asynchronous systems comprised of many homogeneous copies of a generic process template. The process template is represented as a synchronization skeleton [8], where enabling guards have a special character, either disjunctive or conjunctive. Correctness properties are expressed using CTL*\X (CTL* minus the nexttime X) We reduce model checking for systems of arbitrary size n to model checking for systems of size (up to) a small *cutoff* size c. This establishes decidability of PMCP as it is only necessary to model check a finite number of relatively small systems. In a number of interesting cases, we can establish polynomial time decidability. For example, we can reduce the problem of checking mutual exclusion for a critical section protocol to reasoning about systems with 2 processes. We emphasize this *algorithmically* establishes correctness for systems of *all* sizes n. This method generalizes and has been applied to systems comprised of multiple heterogeneous classes of processes, e.g., m readers and n writers.

3 Theory and Practice in Model Checking

In this section we discuss factors related to the usefulness of and use of model checking. The core theoretical principles underlying model checking rest on a few basic ideas including, e.g. some from modal logic (the finite model theorem), lattice theory (the Tarski-Knaster theorem permitting branching time modalities to be calculated easily), and automata theory (the language containment approach). These ideas are well-known. It has also become increasingly clear that model checking is useful in practice. Model checking is employed by most major hardware companies, e.g., Cadence, IBM, Intel, and Motorola, for verification and debugging of microprocessors circuits. Model checking is showing promise for the verification of software, and is being used or is under investigation by, e.g., Lucent, Microsoft, NASA. We would now like to offer an account of why it is that model checking turns out to be useful in practice. It involves the following factors.

Model checking = search ∧ efficiency ∧ expressiveness. Plainly, efficiency is key. Impressive progress has been made with symbolic data structures such as BDDs, and significant advances with various forms of abstraction are ongoing. But expressiveness is also key. An efficient automated verification method that could not express most of the important correctness properties of interest would be of little use, as would a method that lacked the modeling power to capture reactive systems of interest. It seems model checking fares well on these fronts.

Temporal logic provides a powerful and flexible language for specification. Overall, it is quite adequate to the task of reasoning about reactive systems. This is perhaps a bit surprising when we recall that we are using *propositional* temporal logic. But the work of Kamp [20], showing that LTL is essentially equivalent to the First Order Language of Linear Order, does provide a measure of expressive completeness. It is certainly the case that there are properties such as "P holds at every even moment, and we don't care about the odd moments" which are not expressible in LTL. However, they are expressible in the framework of ω-fsa's which means that our technical model checking machinery is still applicable. The author, in fact, views automata as just generalized formulae of temporal logic. Automata and temporal logic, broadly speaking, are the same thing. The advantage of temporal logic, which is sometimes important in practice, is the close connection with tensed natural language.

A finite state framework suffices. At least it does in practice for most of the applications most of the time. This is genuinely surprising, but, there are several reasons for it. First, from the standpoint of specifications, most all propositional modal and temporal logics, including LTL and CTL, have the (bounded) *finite model property.* If a formula f in such a logic is satisfiable in any model, then it is satisfiable in a finite model (of size bounded by some function of the length of f, e.g., $exp(\ |f|\ )$). If a system can be specified in propositional temporal logic, it can be realized by a finite state program. Two decades of experience shows that many (of at least the crucial parts of) reactive systems can be specified in propositional temporal logic, and hence should be realized by a finite state program. Second, from the standpoint of the reactive programs themselves, we see that most solutions to synchronization problems presented in the literature are, indeed, finite state. Typically, in a concurrent program, we can cleanly separate out the finite state synchronization skeletons [7] of the individual processes, which synchronize and coordinate the processes, from the sequential code. For example, in the synchronization skeleton for the solution to the mutual exclusion problem, we abstract out the details of the code manipulating the critical section and obtain just a single node. In the field we are starting to see a trend in model checking of software of abstracting out the irrelevant sequential parts and boiling down the program to a system of finite state synchronization skeletons.

Model checking is highly automated. It is fully automated in principle and highly automated in practice. Human intervention is sometimes required in practice for such things as determining a good variable ordering when doing BDD-based symbolic model checking. It can also be required in doing certain abstractions to get the model to be checked. For instance, in the case where one is verifying a module in isolation, typically a human must understand the environment in which the module operates and provide an abstraction of the environment for the module to interact with. Still, model checking is highly automated even in practice. Model checking seems to be more popular in industrial usage than theorem proving because of the high degree of automation. Theorem provers, while having in principle unbounded capacity, require human expertise to supply key lemmas, and, a skilled "operator" of the tool. In practice, the operator is usually someone with a Ph.D. in CS, EE, or, quite often, Mathematics. For this reason, deployment of theorem provers may be hindered in an industrial setting. Due to the automation of model checkers, they can successfully be used by engineers and programmers at the M.S. or B.S. level. In management's view, this facilitates the wide-scale deployment of verification technology in the organization.

4 Conclusion

There is nowadays widespread interest in *Computer Aided Verification* of reactive systems, as evidenced by the attention paid to the topics at FST-TCS, as well as such conferences as CAV, TACAS, CONCUR, FMCAD, etc. The reason is that such automated techniques as model checking as well as partially automated theorem proving have by now been shown to actually work on a variety of "industrial strength" examples. Pnueli [28] argues that, due to the success of techniques such as model checking on actual applications, we are on the verge of an era of *Verification Engineering*. Of course, there is still a gulf between what we need to do and what we currently have the capacity to do. Basic advances as well as concerted engineering efforts are called for. One popular, and valuable, idea is the integration of theorem provers with model checkers; a number of researchers are pursuing this topic. For the present, however, this researcher's primary interest is still to try to push the idea of model-theoretic automation as far as possible, aspiring to *Completely Automated Verification*.

References

1. K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 15, pages 307-309, 1986.
2. M.C. Browne, E.M. Clarke and O. Grumberg. Reasoning about Networks with Many Identical Finite State Processes. *Information and Control*, 81(1), pages 13-31, April 1989.
3. R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation" *IEEE Transactions on Computers*, 35(8): 677-691 (1986).
4. E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic", *Logics of Programs Workshop*, IBM Yorktown Heights, New York, Springer LNCS no. 131, pp. 52-71, 1981.
5. E. M. Clarke, R. Enders, T. Filkorn, S. Jha, "Exploiting Symmetry In Temporal Logic Model Checking", *Formal Methods in System Design*, vol. 9, no. 1/2, pp. 77-104, Aug. 96.
6. E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", *ACM Trans. on Prog. Lang. and Sys (TOPLAS)* 8(2): 244-263 (1986).
7. E. A. Emerson, *Branching Time Temporal Logic and the Design of Correct Concurrent Programs*, Ph.D. Dissertation, Harvard University, 1981.
8. E. A. Emerson, E M. Clarke, "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons", *Science of Computer Programming*, 2(3): 241-266 (1982)
9. E. A. Emerson, J. Y. Halpern: " 'Sometimes' and 'Not Never' Revisited: On Branching versus Linear Time Temporal Logic", *Journal of the Assoc. Comp. Mach. (JACM)*, 33(1): 151-178 (1986).
10. E. A. Emerson, J. Havlicek, and R. J. Trefler, "Virtual Symmetry", LICS'00, pp. 121-132.
11. E.A. Emerson and K.S. Namjoshi. Reasoning about Rings. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85-94, 1995.
12. E.A. Emerson and K.S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems. In *Computer Aided Verification, Proceedings of the 8th International Conference*. LNCS , Springer-Verlag, 1996.
13. E. A. Emerson and K. S. Namjoshi, "Verification of Parameterized Bus Arbitration Protocol", Conference on Computer Aided Verification (CAV), pp. 452–463, 1998.

14. E.A. Emerson and A.P. Sistla,. Symmetry and Model Checking. In *Formal Methods in System Design*, vol. 9, no. 1/2, pp. 105-131, Aug. 96.
15. E. A. Emerson and A. P. Sistla, "Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach", *ACM Trans. on Prog. Lang. and Systems (TOPLAS)*, pp. 617–638, vol. 19, no. 4, July 1997.
16. E. A. Emerson and R. J. Trefler, "Model Checking Real-Time Properties of Symmetric Systems", MFCS 1998: 427-436.
17. E. A. Emerson, R. J. Trefler, "From Asymmetry to Full Symmetry: New Techniques for Symmetry Reduction in Model Checking", CHARME 1999: 142-156.
18. S.M. German and A.P. Sistla. Reasoning about Systems with Many Processes. *J. ACM*,39(3), July 1992.
19. C. Ip and D. Dill. Better verification through symmetry. In *Formal Methods in System Design*, vol. 9, no. 1/2, pp. 41-76, Aug. 1996.
20. Kamp, J. A. W., "Tense Logic and the Theory of Linear Order", Ph.D. thesis, University of California, Los Angeles, 19868.
21. R. P. Kurshan, *Computer Aided Verification*, Princeton Univ. Press, 1994.
22. R.P. Kurshan and K. McMillan. A Structural Induction Theorem for Processes. In *Proceedings of the Eight Annual ACM Symposium on Principles of Distributed Computing*, pages 239-247, 1989.
23. B. Lubachevsky. An Approach to Automating the Verification of Compact Parallel Coordination Programs I.*Acta Informatica 21*, 1984.
24. K. McMillan, *Symbolic Model Checking*, Ph.D. Dissertation, CMU, 1992.
25. K. McMillan, Verification of Infinite State Systems by Compositional Model Checking, CHARME'99.
26. M Pandey and R. E. Bryant, "Exploiting Symmetry When Verifying Transitor-Level Circuits by Symbolic Trajectory Evaluation", CAV 1997: 244-255.
27. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the eighteenth Symposium on Foundations of Computer Science*. 1977.
28. A. Pnueli, "Verification Engineering: A Future Profession" (A. M. Turing Award Lecture), Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC 1990), San Diego, August, 1997; `http://www.wisdom.weizmann.ac.il/~amir/turing97.ps.gz`
29. F. Pong and M. Dubois. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Transactions on Parallel and Distributed Systems*, August 1995.
30. J-P. Queille and J. Sifakis, "Specification and Verification of Concurrent Systems in CESAR", *International Symposium on Programming*, Springer LNCS no. 137, pp 337-351, 1982.
31. A. P. Sistla, Parameterized Verification of Linear Networks Using Automata as Invariants, CAV, 1997, 412-423.
32. I. Vernier. Specification and Verification of Parameterized Parallel Programs. In *Proceedings of the 8th International Symposium on Computer and Information Sciences*, Istanbul, Turkey, pages 622-625,1993.
33. P. Wolper and V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In J. Sifakis(ed) *Automatic Verification Metods for Finite State Systems*, Springer-Verlag, LNCS 407, 1989.