

# Software Model Checking During Design and Implementation \*

Fang Yu

Advisor: Tevfik Bultan  
Department of Computer Science  
University of California, Santa Barbara

November 29, 2006

## Abstract

I present a brief survey of verification techniques used in current model/program/design checkers. My future research will focus on integrating these techniques to achieve scalable software verification.

## 1 Introduction

Model checking is a promising technique for establishing correctness of software systems. The research in software model checking has shifted from verification of hand-built models at the design stage to automated verification of the code at the implementation stage. I present a brief survey of current tools and techniques used in design and implementation level verification of software systems. My future research goal is to compare and integrate these verification techniques to achieve scalable software verification.

This survey consists of three parts: 1) model checking, 2) program verification, and 3) software modelling and analysis. I first discuss several key model checking techniques developed in the past two decades. These include abstraction [CGL92], symbolic model checking [BCL90,BCMDH90], symmetry reduction [ES94], partial order reduction [CGMP99], bounded model checking [AKMM03, CBRZ01], induction [SSS00] and interpolation [McMillan03]. These techniques significantly increased the power of model checking, and most of them are widely adopted in current model checkers. In this survey, I focus on three model checkers that use different combinations of these techniques: SPIN (an explicit state model checker [Holzmann97]), SMV (a symbolic, BDD-based model checker [McMillan93]), and ALV (a symbolic, infinite-state, constraint-based model checker [YBB05]).

While model checkers such as SPIN, SMV and ALV verify temporal logic properties of specifications written in their own input languages, program checkers analyze specific properties of real programs written in modern programming languages. I first discuss the abstract-check-refine paradigm [HJMS02] and thread modular reasoning [HJMQ03] to facilitate program verification,

---

\*This is the reading list for Fang's Major Area Exam.

and then I investigate the following program checkers that use model checking techniques to find errors in programs: 1) Java: ESC/Java [FRLN02], Java Pathfinder [BHPV00], Bandera [CD00], and 2) C/C++: Blast [BMMR01], VeriSoft [Godefroid97], CMC [MPCED02], CBMC [CKL04], Bebop [BR01]. These program checkers address different program correctness issues based on their algorithms, target languages and verification properties. I discuss the core algorithms used in these checkers.

The third part of this survey pays attention on software modelling and analysis. Software modelling is extremely important for developing dependable software systems. It is necessary to have tools and techniques for building design models for a software system. These design models should allow the managers and the software developers (possibly building different components of the same system) to communicate their ideas about the software system before and during implementation. Moreover, these design models should allow automated analysis in order to remove design flaws before they become part of the implementation. I will briefly review the following software modelling languages: Java Modelling Language (JML) [LBR99], Unified Modelling Language (UML) [OMG99], Message Sequence Charts (MSCs) [MSC96], Object Constraint Language (OCL) [OCL99], Alloy [Jackson01]. These modelling languages are used to formally specify the software design before code generation/implementation. Previous research addressing correctness in these design languages include a) Bogor [RRDH04] for verification of JML, b) model checking UML state machines [CH00, LP99], c) Alur's work [AY99] on MSCs, d) USE [GBR03] for UML/OCL simulation, and e) Alloy Analyzer [DCJ06].

Below, I will provide a brief summary, as well as the list of papers for each of the three topics mentioned above.

## 2 Model Checking

A fundamental problem in software testing is that one cannot extrapolate between test cases. That is if one chunk of software works, that fact says nothing about the other; they are discrete and separate. Instead of investigating sufficient test cases to cover all scenarios, model checking poses an attractive way to explore all feasible executions to guarantee the system correctness. However, the number of possible states are exponential in the size of the software system, which is known as the *state explosion problem*.

Abstraction is one of the most important techniques for attacking the state explosion problem. The idea is to check if a property holds on an abstract system, and then based on this result infer if the same property holds on the concrete system. An abstraction can be formed by defining a mapping between concrete states and abstract states such that the abstract transition system allows more behaviors than the concrete system but has fewer states. Verification task is performed on the abstract transition system, which has fewer states than the concrete transition system, reducing the cost of verification. In [CGL92], Clarke et al. show how to construct abstract transition systems so that any property verified on the abstract system holds on the

concrete one. Their approach works for the properties expressed in the temporal logic  $\forall CTL^*$ . Since the abstract transition system has more behaviors than the concrete one, a counterexample generated for the abstract transition system might not be a feasible counter-example for the concrete transition system. In [CGJLV00], Clarke et al. propose a complete methodology for  $\forall CTL^*$  by using counterexamples to refine abstract models. The main idea is to iteratively refine an abstract transition system in order to remove infeasible counter-examples. I will talk about this approach in more detail while discussing the Blast program checker below.

Another way of dealing with the state explosion problem is to reduce the number of concrete states that need to be searched during the state space exploration. There are two main techniques that use this approach 1) the symmetry reduction technique by Emerson and Sistla [ES94] and 2) the partial order reduction technique Clarke et al. [CGMP99]. Both of these techniques induce an equivalence relation on states of the system and perform analysis without exploring the states which have an equivalent state that has already been explored. I.e., during the state space exploration if we come to a state that has an equivalent state that has already been explored, then the current state is not explored.

*Explicit state model checking* systematically and exhaustively searches error states in a state graph. A basic algorithm is that the checker starts from an initial state and recursively generates successive system states by executing the nondeterministic events of the system. States are stored in a hash table to ensure each state is explored at most once. The process continues either until all reachable states are explored or the checker runs out of resources. SPIN [Holzman97], developed at Bell Labs, is a widely used explicit state model checker. On the other hand, in *symbolic model checking* [BCL90, BCMDH90], system states are encoded using clever data structures such as Binary Decision Diagrams (BDDs) that provide canonical and compact representation of large state spaces and allow their efficient manipulation. SMV (Symbolic Model Verifier) [McMillan93], developed at CMU, is a BDD-based model checker for checking properties expressed in temporal logic CTL.

Though symbolic model checking with BDDs has been successfully used for formally verifying finite state systems such as sequential circuits and protocols, the bottleneck of this method is that BDDs may grow exponentially, particularly for those systems with a large number of variables. To address this problem, Clarke et al. [CBRZ01] proposed bounded model checking (BMC) technique. In BMC, the model checking problem is reduced to checking satisfiability of a Boolean logic formula. This is achieved by concatenating the formula for the initial condition with a formula that corresponds to unrolling of the transition relation a fixed number of times (determined by the given bound). A SAT solver is used to determine the satisfiability of the resulting formula. One major advantage of BMC is the efficiency of current SAT solvers. SAT solvers today can handle formulas with hundreds of thousands of variables and millions of clauses using a variety of heuristics. This enables verification of large systems using BMC. Bounded model checkers are known to perform well in bug finding but poor for proving correctness

[AKMM03]. In order to prove correctness, the bound used in BMC has to be as large as the diameter of the transition system. Finding this diameter of a transition system is proven to be NP-complete, and in most cases, even if the diameter is known, this bound is prohibitively high to reach. Hence, SAT-based model checking is considered to be complementary to BDD-based model checking. NuSMV 2 [CCGGPRST02] incorporates BMC techniques in SMV, and users can use both techniques to verify their systems.

On the other hand, some researchers investigated unbounded model checking with SAT-based techniques. Shreean et al. [SSS00] incorporated an inductive verification method in BMC. An inductive proof consists of 1) a base case and 2) an induction step. In BMC, to prove that a property  $P$  holds in all states, an inductive proof is constructed by showing that 1)  $P$  holds in all the initial states, and 2)  $P$  is preserved by the transition relation. The inductive method is sound and once an inductive proof is constructed, it may handle large models. However, there is no guarantee that whether or when an inductive proof can be constructed. McMillan [McMillan03] presented a fully SAT-based technique for unbounded symbolic model checking based on Craig interpolants. An interpolant  $P$  is derived in linear time from a given refutation, i.e., a proof that there is no counter example of  $k$  steps or fewer.  $P$  is used as an over approximation image of the initial constraints so that  $P$  is true in every state reachable from the initial state in one step and no state satisfying  $P$  can reach a final state in  $k-1$  steps. McMillan proved that by increasing  $k$ , eventually either a true counterexample is discovered or the property is proven to be true. Termination is guaranteed in finite state systems.

In general, model checking of infinite state systems is undecidable. ALV (Action Language Verifier) [YBB05] is a conservative infinite state model checker, which incorporates BDD and polyhedra/automata representations to verify systems with integer variables. It adopts symbolic model checking to verify CTL properties, but since it allows unbounded integers, the fixpoint computations are not guaranteed to converge. Various heuristics, such as widening, are implemented to compute an approximation of the fixpoint. Using these approximations ALV can verify or falsify a property. However, in some cases, if the approximations are not precise enough, ALV may not be able to give a conclusive answer.

## 2.1 Reading List

- AKMM03 N. Amla, R. Kurshan, K. McMillan and R. K. Medel, Experimental Analysis of Different Techniques for Bounded Model Checking. In Proc. of TACAS'03, LNCS, Warsaw, Poland, 2003.
- BCL90 J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In VLSI 91, Edinburgh, Scotland, 1990.
- BCMDH90 J.R. Burch, E.M. Clarke, K.L. McMillan, D.L.Dill, L.J. Hwang. Symbolic Model Checking:  $10^{20}$  States and Beyond, IEEE LICS, 1990.

- CGJLV00 E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement. In Twelfth Conference on Computer Aided Verification (CAV'00). SpringerVerlag, July 2000.
- CBRZ01 E. Clarke, A. Biere, R. Raimi, and Y. Zhu, Bounded Model Checking Using Satisfiability Solving. In Formal Methods in System Design, July 2001.
- CCGGPRST02 A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella, NuSMV 2: An Open Source Tool for Symbolic Model Checking. In Proceeding of International Conference on Computer-Aided Verification (CAV 2002). Copenhagen, Denmark, July 27-31, 2002
- CGL92 E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, January 1992.
- CGMP99 E. Clarke, O. Grumberg, M. Minea, D. Peled. State-Space Reduction using Partial-Ordering Techniques, STTT 2(3), 1999, pp.279-287.
- ES94 E. A. Emerson and A. P. Sistla, Symmetry and model checking, In Proc. of the International Conference on Computer Aided Verification (CAV93), LNCS 697, pp. 463-478, Elounda, Greece, 1993.
- Holzmann97 G. J. Holzmann, The model Checker SPIN, IEEE Transaction on Software Engineering, Vol 23, No. 5, May, 1997, pp. 279-295.
- KLMPY98 R. P. Kurshan, V. Levin, M. Minea, D. Peled and H. Yenigun, Static Partial Order Reduction, In Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS'98), LNCS 1384, pp. 345-357, Lisbon, 1998.
- McMillan93 K. L. McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic, 1993.
- McMillan03 K. L. McMillan, Interpolation and SAT-Based Model Checking. In Proc. of Computer Aided Verification (CAV'03), LNCS 2725, pp. 1-13, 2003.
- SSS00 M. Sheeran, S. Singh and G. Stålmarck, Checking Safety Properties Using Induction and a SAT-solver. In Proc. of FMCAD 2000. LNCS 1954:108, 2000.
- YBB05 Tuba Yavuz-Kahveci, Constantinos Bartzis, and Tevfik Bultan. Action Language Verifier, Extended. In Proc. of the 17th International Conference on Computer Aided Verification (CAV05), LNCS 3576, pp. 413-427, 2005.

### 3 Program Verification

Program verification tools integrate model checking and static analysis techniques to automatically verify programs written in modern programming languages, such as Java and C/C++. Two main questions in program verification are 1) what is an appropriate model for a program that is simultaneously a) abstract enough to permit efficient checking and b) precise enough to preclude false positives as well as yield real traces, and 2) how can we derive such models automatically?

To address these questions, Henzinger et al. [HJMS02] adopted abstract-check-refine paradigm [CGJLV00]. They start with a coarse abstraction which over approximates system behaviors, and iteratively check and refine the abstract model based on the infeasible counter examples. An intuitive coarse abstraction is the Control Flow Automata (CFA), which is a non-deterministic finite automata used to trace control locations of procedures without tracing values of variables. They refine CFA via predicate abstraction, i.e., they choose the branch conditions as predicates and trace how their values change. Instead of tracing the exact values of variables, they trace the truth of Boolean formulas over a finite set of predicates  $P$  over the variables in programs. Since the system is over approximated, the counterexample may be infeasible in the real program. Once its infeasibility is determined by running the program, the abstraction is refined. This method is known as *lazy abstraction*. The refinement is processed by expanding  $P$  with the set of predicates in branch conditions that make the path infeasible. In [HJMK04], Henzinger et al. further improve the refinement by discovering Craig interpolants. They have realized these ideas in the program checker Blast [BMMR01], which has successfully verified the C programs with more than 130,000 lines of codes.

Another issue in program verification is how to handle interleaving behaviors of multi-threaded programs. Henzinger et al. [HJMQ03] extend thread modular reasoning to abstract multi-threaded programs. The main idea of thread modularity is that the state space of *one* thread is explored at a time, making assumptions about how the environment can interfere. The system contains a main thread (system) and a context (environment) which is an abstraction of all the other threads. Then they verify that 1) this composed system is safe ("assume") and 2) the context is a sound abstraction ("guarantee"). Instead of tracking the control location of each thread separately, they count the number of threads at each control location via the context automata.

Below, we list and briefly review some current program checkers for Java and C/C++.

- Extended Static Checking for Java (ESC/Java) [FLLNSS02] is a static analyzer for Java programs developed by Compaq research lab. Programmers specify design decisions in an annotation language, and ESC automatically checks the consistency between the annotations and the actual code using automated theorem proving techniques. During verification, loops are unrolled a fixed number of times, and methods are analyzed individually

by replacing each method call in a method body with its pre and post conditions.

- Java PathFinder (JPF) [VHPL02] provides a verification and testing environment for Java. JPF is an explicit state model checker which has its own Java Virtual Machine to execute bytecodes, as well as a search component to guide the execution. JPF incorporates many techniques to facilitate exhaustive state space search. These techniques include 1) predicate abstraction to reduce the state space, 2) static analysis for supporting partial order and symmetry reductions, and 3) runtime analysis to pinpoint problematic segments. A hash table recording visited states in an integer vector is used for state comparison, while a stack of complex data structures is used for backtracking.
- Bandera is another tool for verification of Java programs. It translates Java bytecodes to either Promela (SPIN's input language) or to the input language of SMV.
- Verisoft is an explicit state model checker but does not store states. It systematically executes the actual C code. Unlike JPF, it limits the search depth rather than limiting the number of visited states. Partial order reduction technique is implemented in Verisoft to alleviate state space explosion.
- CMC [MPCED02] is an explicit state model checker working on C/C++ implementations directly. CMC models a system as a collection of processes (each runs unmodified C/C++ code) and explores the reachable state space by alternative scheduling decisions and nondeterministic events.
- CBMC [CKL04] uses bounded model checking techniques to verify C and Verilog programs. In CBMC, Clarke et al. apply variable renaming to create a single assignment program and construct a Boolean logic formula by bounding the depth of the loops and recursion.
- In Slam project, Ball et al. [BMMR01] use predicate abstraction to translate C programs into Boolean programs, and use their symbolic BDD-based model checker, Bebop [BR00], to verify the generated Boolean programs. Similar to Blast, infeasible counterexamples are used to refine previous abstractions at each iteration [BCDR04].

### 3.1 Reading List

BCDR04 Thomas Ball, Byron Cook, Satyaki Das, Sriram K. Rajamani. Refining Approximations in Software Predicate Abstraction, TACAS 2004.

BMMR01 T. Ball, R. Majumdar, T. Millstein, S.K. Rajamani, Automatic Predicate Abstraction of C programs, Proc. PLDI 2001.

BR00 Thomas Ball, Sriram K. Rajamani, Bebop: A Symbolic Model Checker for Boolean Programs, SPIN 2000 Workshop on Model Checking of Software, LNCS 1885, August/September 2000, pp. 113-130.

- FRLNSS02 Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002), pages 234-245, 2002.
- CD00 J.C. Cobett, M.B. Dwyer, et al. Bandera: Extracting finite state models from Java source code, Proc. 22nd Int. Conf. on Software Engineering(ICSE00), pp.439-448.
- CKL04 E. Clarke, D. Kroening, and F. Lerda, A Tool for Checking ANSI-C Programs, Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), LNCS 2988, pp. 168-176.
- Godefroid97 P. Godefroid, VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. Proc. of the 9th Conference on Computer Aided Verification, LNCS 1254, pp. 476-479, June 1997.
- HJMK04 Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from Proofs. Proceedings of the 29th Annual Symposium on Principles of Programming Languages, ACM Press, pp. 232-244, 2004.
- HJMQ03 Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular Abstraction Refinement. In Proceedings of the 15th International Conference on Computer-Aided Verification (CAV), LNCS 2725, Springer-Verlag, pages 262-274, 2003.
- HJMS02 Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy Abstraction. Proceedings of the 29th Annual Symposium on Principles of Programming Languages(POPL), ACM Press, pp. 58-70, 2002.
- MPCED02 Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, David L. Dill. CMC: A pragmatic approach to model checking real code. Proc. Operating System Design and Implementation (OSDI) 2002.
- PSSD00 David Y. W. Park, Ulrich Stern, Jens U. Sakkebaek, and David L. Dill. Java model checking. In Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00), pages 253-256, Sep. 2000
- VHBPL03 W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda, Model Checking Programs, Automated Software Engineering Journal Volume 10, Number 2, April 2003

## 4 Software Modelling and Analysis:

In this section, we focus on some design languages and efforts to verify their correctness. A robust software design plays an essential role in developing dependable software systems. Hence, it is necessary to develop techniques for assisting engineers to formally specify their designs before implementation.

The Unified Modelling Language (UML) [OMG99] is a widely adopted standard notation for specifying object-oriented software systems. UML contains several visual formalisms such as class diagrams, sequence diagrams and statechart diagrams that can be used to specify both structural (class diagrams) and dynamic (sequence and statechart diagrams) views of a software system. Earlier research in verification of UML models focussed on translation of UML models to the input languages of existing verification tools. Clarke et al. [CH00] translate statechart diagrams to the input language of SMV; and Lilus et al. [LP99] translate statechart diagrams to Promela (the input language of SPIN). Instead of direct translation, VeriAgent [MCGOFK03] proposes a protocol interface based on first order logics to integrate UML and formal verification tools.

Message Sequence Chart (MSC) [Z.120] is a scenario-based specification for describing design requirements. MSCs correspond to formalization of sequence diagrams used in UML. Briefly, an MSC specifies the desired message exchange sequences among a set of distributed components. An MSC defines a partial order among the message send and receive events. One traditional way to verify a given MSC is generating a set of communicating state machines that correspond to the message exchange sequences specified by the MSC and then to verify these communicating state machines. The time complexity of analyzing communicating state machines is known as PSPACE hard. Alur et al. [AY99] further proved that by constructing a suitable automaton for the linearization of partial order specified by the MSC, the model checking problem is coNP-complete.

The Object Constraint Language (OCL) [OCL99] is an expression language to describe constraints on object-oriented models. OCL is used primarily for describing class and state invariants, as well as describing pre/post conditions for operations. OCL is used in conjunction with visual diagrams used in UML models (such as class, sequence and statechart diagrams) to add precision to these visual models and allow expression of features that cannot be expressed using such diagrams. Gogolla et al. *simulate* UML models and check OCL constraints in UML-based Specification Environment (USE) [GBR03]. This type of analysis can only guarantee correctness of the system states that are explored during the simulation. Surprisingly, there are few checkers to verify UML/OCL models.

Alloy [Jackson99, Jackson 01] is a concise software modelling language, which offers a compatible declaration syntax with graphical object models and a set-based formula syntax to express complex constraints. Compared to UML/OCL, Alloy offers a compact representation of system designs and an automatic verification mechanism. Alloy Analyzer [JSS00, Jackson00] is used to check properties of Alloy models. It first translates Alloy models to a first order logic formula. Then, this first order logic formula is translated to a quantifier-free Boolean logic formula by bounding the scope (i.e., by bounding the number of objects). Alloy analyzer uses a SAT solver to check satisfiability of the resulting Boolean formula, which leads to identification of errors within the given scope.

## 4.1 Reading List

- AY99 R. Alur and M. Yannakakis. Model checking of message sequence charts. In CONCUR'99: Concurrency Theory, Tenth International Conference, LNCS 1664. pp. 114-129.
- CH00 E. Clarke and W. Heinle. Modular translation of statecharts to SMV. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, Aug. 2000.
- GBR03 Martin Gogolla, JoLrn Bohling, and Mark Richters. Validation of UML and OCL Models by Automatic Snapshot Generation. In Proc. 6th Int. Conf. Unified Modeling Language (UML'2003). Springer, Berlin, LNCS 2863, 2003.
- Jackson99 Daniel Jackson, A Comparison of Object Modelling Notations: Alloy, UML and Z. MIT Lab for Computer Science. August 11,1999.
- Jackson00 Daniel Jackson. Automating First-Order Relational Logic. Proc. ACM SIGSOFT Conf. Foundations of Software Engineering, November 2000.
- Jackson01 Daniel Jackson, Alloy: A Lightweight Object Modeling Notation. Proc. in the ACM Transactions on Software Engineering and Methodology, Vol. 11, Issue 2, pages 256-290; April 2002.
- Jackson06 Daniel Jackson. Dependable Software by Design. Scientific American. June 2006.
- JSS00 Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the alloy constraint analyzer. In Proc. 22nd International Conference on Software Engineering, Limerick, June 2000.
- LBR99 Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), Behavioral Specifications of Businesses and Systems, chapter 12, pages 175-188. Copyright Kluwer, 1999.
- LP99 J. Lilius and I. P. Paltor. Vuml: a tool for verifying uml models. Technical report, Abo Akademi University, 1999.
- MCOGFK03 E. Mota, E. Clarke, W. Oliveira, A. Groce, M. Falcao, and J. Kanda. VeriAgent: an Approach to Integrating UML and Formal Verification Tools." In Sixth Brazilian Workshop on Formal Methods (WMF 2003), pp. 111-129, Brazil, October 2003.
- OCL99 OMG. Object Constraint Language Specification. In OMG Unified Modeling Language Specification, Version 1.3, June 1999.
- OMG99 OMG, editor. OMG Unified Modeling Language Specification, Version 1.3, June 1999. Object Management Group, Inc., Framingham, Mass.

- RRDH04 Robby, Edwin Rodriguez, Matthew B. Dwyer, John Hatcliff. Checking JML Specifications Using An Extensible Software Model Checking Framework, August 2004. In the International Journal of Software Tools for Technology Transfer (STTT).
- RDH06 Robby, Matthew B. Dwyer, John Hatcliff. Bogor: A Flexible Framework for Creating Software Model Checkers, In the Proceedings of Testing: Academic and Industrial Conference - Practice And Research Techniques, June 2006.
- MSC96 Z.120. ITU-TS recommendation Z.120: Message Sequence Chart (MSC), 1996.