Generic Tools for Verifying Concurrent Systems*

Rance Cleaveland
Steven T. Sims
Department of Computer Science
North Carolina State University
226 Withers Hall
Raleigh, NC 27695-8206 USA
rance@csc.ncsu.edu

Abstract

Despite the enormous strides made in automatic verification technology over the past decade and a half, tools such as model checkers remain relatively underused in the development of software. One reason for this is that the bewildering array of specification and verification formalisms complicates the development and adoption by users of relevant tool support. This paper proposes a remedy to this state of affairs in the case of finite-state concurrent systems by describing an approach to developing customizable yet efficient *verification tools.*

1 Introduction

The field of automatic verification of finite-state systems has experienced tremendous advances over the past decade and a half, as efficient verification algorithms have been developed and associated tools built and applied to case studies of substantial complexity [13, 18]. Within the hardware community, commercial interest in these tools has even begun, as companies such as Intel, National Semiconductor and Chrysalis Symbolic Design have incorporated the use of automatic verification tools in their design processes. Despite these developments, however, verification technology remains largely unused in the software community in general, even in areas such as process control and communications protocols, where finite-state models form the basis of system implementations, thereby rendering them candidates for automatic analysis.

One may identify several cultural and technical reasons for this lack of uptake within the software community: lack of training, uncertainty about how to deploy formal analysis in the software process, skepticism about the benefits versus the costs of formal analysis, etc. While it is beyond the scope of this paper to discuss all these concerns, we do note that even users who might be interested in formal approaches to the analysis of finite-state concurrent systems are confronted with the following issues.

1. Which design notation should be used for representing software artifacts? The literature contains a number of proposals, including Esterel [1], Statecharts [24], SDL [9], LOTOS [29], and CSP [25], to name only a few of the best-known ones.

2. How should requirements for designs be formulated? Again, the literature contains numerous suggestions, including finite-state machines, Computation Tree Logic [12] and Linear Temporal Logic [27], to name a few.

This bewildering array of choices has two negative consequences. The first is that no specification formalism has yet achieved a "critical mass" of users. The second is that tool support (necessary for any serious use of formal analysis) remains fairly primitive from a user's perspective; the lack of a "market" for any single formalism has dissuaded tool builders from expending the resources needed to build sophisticated and usable tools. The lack of appropriate tool support has in turn retarded the uptake of automatic verification among software designers.

In this paper we propose a framework for developing *generic* and *customizable* verification tools and investigate its use as a basis for efficient automated analysis of finite-state systems. The framework is intended to ease dramatically the task of developing usable tools for (operationally based) verification formalisms, thereby removing, at least in principle, one obstacle to the increased adoption of verification technology in practice.

*Research supported by NSF grant CCR-9120995, NSF Young Investigator Award CCR-9257963, NSF grant CCR-9402807, NSF grant CCR-9505562, and AFOSR grant F49620-95-1-0508.

2 Fundamental Concepts in the Verification of Finite-State Systems

This section sketches the concepts we deem fundamental for the analysis and verification of finite-state systems. The first two involve approaches to establishing that finite-state systems satisfy their specifications. In general, one may identify two schools of thought regarding the verification of systems: *logic-based* approaches and *refinement-based* techniques. The former typically involve the use of a temporal logic for describing desired system properties; one then uses a *model checker* to determine whether or not the properties hold of a putative implementation. The latter uses abstract, "high-level" systems as specifications; one then proves an implementation correct by showing that it "refines" such a specification (i.e. is related to it be an appropriate behavioral equivalence or preorder). Both approaches have their uses, and a number of temporal logics and behavioral relations have been proposed for verification purposes [13, 18].

So what is fundamental to these approaches? In the case of model checking, we and others [22, 3] posit that the **modal mu-calculus** [26] constitutes an expressive and efficient basis. This logic provides simple modalities and propositional constructs together with mechanisms for defining properties recursively. Efficient model-checking algorithms have been developed for fragments of this logic [2, 19, 21], and other temporal logics have efficient translations into these fragments [3, 22]. Regarding refinement-based approaches, we argue for the fundamentality of **(bi)simulation**. Efficient algorithms exist for determining whether systems are related by bisimulation equivalence (simulation preorder) [30, 23, 11], and other relations may be computed efficiently by combining decision procedures for (bi)simulation with appropriate transformations on the underlying finite-state systems [10, 16]. In addition, general theories of bisimulation-based "diagnostic information" that explain why a system fails to refine another have been developed [11].

The final fundamental notion involves the definition of design notations for representing finite-state systems. In order to be usable as a basis for formal analysis, such notations must, in addition to having useful constructs, be equipped with a formal semantics that unambiguously defines an association between "programs" in the language and finite-state machines representing their behavior. To give such a semantics, we advocate the use of operational semantic in general, and **structural operational semantics (SOS)** [31] in particular, as a rigorous yet conceptually clear presentation style. SOS presentations consist of collections of inference rules that specify the single-step transitions of systems in terms of the execution steps of their components. Languages such as CCS [28], LOTOS [29] and CSP [25] have such a semantics, and it has become the preferred style for defining the meaning of constructs in process algebra [5]. An additional virtue of operational semantics, and SOS in particular, involves its connection with *simulation*: an operational account of a language implicitly defines how to simulate "programs" in the language.

3 The Concurrency Workbench And Analytical Genericity

The previous section presented a proposed foundation for the automatic verification of finite-state systems. In order for this theory to be of practical as well as theoretical value, one must show that it can be used as a basis for the development of usable verification tools. This section and the one following explore this issue by describing our experience with two associated automatic verification tools: the Concurrency Workbench of North Carolina [17] and the Process Algebra Compiler of North Carolina [15].

The Concurrency Workbench was originally conceived as a "laboratory" for experimenting with different techniques for verifying finite-state systems represented in CCS [16]. The tool incorporated implementations of bisimulation, prebisimulation and mu-calculus model-checking algorithms and provided support for easily customizing these algorithms to calculate a variety of different behavioral relations and for introducing new temporal constructs. The original public release of the system suffered from several performance bottlenecks, and consequently while it was easy to customize it could be frustratingly inefficient. The tool was nevertheless used successfully in the analysis of several case studies [7].

The Concurrency Workbench of North Carolina (CWB-NC) [17] represents a completely reimplemented version of the original CWB. Our goal in this effort has been to show that the inefficiencies of the CWB were due not to its genericity (as some have suggested) but rather to lower-level implementation issues that can be addressed in a design-language- and analysis-independent manner. Consequently, the CWB-NC retains the (pre)bisimulation/mu-calculus orientation of the original CWB, but it contains more efficient implementations of the low-level routines. It also cleanly separates routines that are design-language-specific (parsers, unparsers, transition calculation) from those that are independent of the design notation (bisimulation, model checking, simulator) in order to facilitate modifications to the language that is supported. Figure 1 contains a representation of the architecture of the CWB-NC. The CWB-NC has been publicly available since September of 1996 from URL `www4.ncsu.edu/~rance/WWW/cwb-nc.html` and has been used in the analysis of several reasonably sophisticated case studies [14, 20]. While a detailed comparison has not been conducted, preliminary evidence suggests that the CWB-NC is 2–3 orders of magnitude faster than an earlier version of the CWB (specifically, version 6.0).

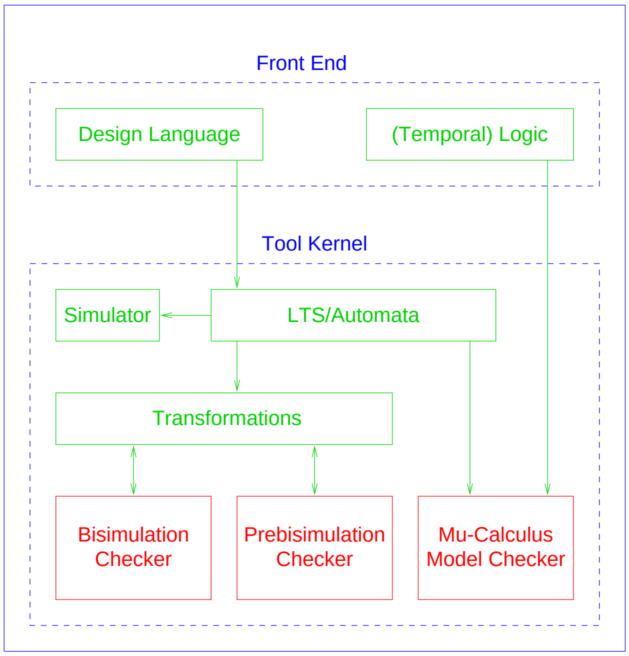

Figure 1: The Architecture of the CWB-NC.

4 The Process Algebra Compiler and Language Genericity

Our experience with the CWB and CWB-NC suggests that (pre)bisimulations and the modal mu-calculus form an efficient yet easily customizable basis for system verification. However, changing the design language supported by the CWB-NC requires substantial and delicate recoding in order for performance to be acceptable. In order to alleviate the difficulty of this task, the Process Algebra Compiler (PAC) project between NCSU and INRIA-Sophia Antipolis was undertaken with support from the US National Science Foundation and INRIA [15]. The PAC aims to produce efficient front ends for verification tools from high-level descriptions of the syntax and semantics of the design language the front-end is intended to support. The Process Algebra Compiler of North Carolina (PAC-NC) constitutes the specialization of the PAC for the CWB-NC.

The PAC-NC takes as input files defining the abstract and concrete syntax of a design notation and its operational semantics as SOS rules and generates SML code (the implementation language of the CWB-NC) implementing parsers, unparsers and relevant semantic routines (primarily a transition calculator). A user may then insert these routines into the CWB-NC in order to change the design notation supported by the tool. Figure 2 graphically depicts this process. It should be noted that all versions of the PAC, including the PAC-NC, use the same PAC front end; they differ only in the code they produce, since different verification tools expect routines in different languages and with different functionalities.

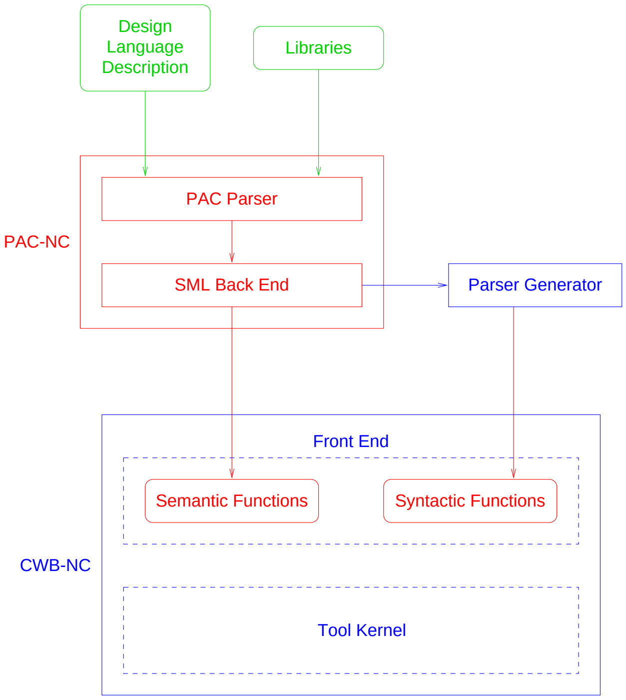

Figure 2: The PAC-NC Architecture.

Efficiency Issues. The CWB-NC makes extremely heavy use of the semantic routines for a design language; to construct an automaton from a design language "program", for instance, the transitions function must be called for each state. Consequently, in order for the PAC-generated front ends to be usable, great care must be taken to ensure the efficiency of the automatically-generated semantic routines. To achieve this, the PAC-NC combines a general pattern-matching-oriented approach with two low-level optimizations in the semantic routines it generates. We briefly describe these here; the interested reader is referred to [15] for a more detailed account.

To build a function for a semantic routine from an SOS specification, the PAC-NC first analyzes the rules on the basis of the design language constructs they are applicable to. It then generates a function that, given a "program" in the design language, determines the applicable rules, recursively calculates the semantic information for appropriate subprograms, and then uses the rules to combine the results of the recursive calls appropriately. To make this process as efficient as possible, the produced routine also does the following.

Call caching. The results of certain previous recursive calls are stored in a table in order to avoid duplication of effort. (Which call results are cached in this manner is presently left up to the user, although this information could also be determined by analyzing the SOS rules appropriately.)

Tree flattening. Parse trees are represented in a compact fashion in order to facilitate hashing and equality-checking on trees.

Tables 1 and 2 contain the time and space results of some experiments with different PAC-generated front ends for the CWB-NC. The experiments were conducted on a 180 MHz Pentium Pro machine with 64 MB of memory. The timing table records the time needed to construct an automaton from a given program, while the space table indicates the maximum heap size needed. The programs used include the following.

802-2: A simplified version of the IEEE 802-2 token ring protocol.

Mailer: A version of the electronic mail protocol used by the Computer Science Department of the University of Edinburgh in the late 1980s [6].

ATM: An account of version 3.0 of the User/Network Interface in the ATM communications protocol [10].

Emitter: A sender in a communications protocol.

Railway (LOTOS): A railway signaling scheme.

Railway (PCCS): The same railway signaling scheme in the PCCS process algebra [14].

In general, caching and tree flattening lead to significant improvements in timing behavior. Somewhat surprisingly, they also induce improved memory performance on occasion. This seeming anomaly results from sharing in the parse tree representations that caching in particular supports. It should also be noted that the benefits of tree flattening grow as the syntactic complexity of designs increases. Thus, the improvement induced by tree flattening in the ATM example and the LOTOS examples is much bigger than in the other, less syntactically elaborate examples. Finally, caution should be used in interpreting the space-usage results, owing to the well-known difficulties in the space profiling of garbage-collected languages such as SML.

Table 1: Timings for PAC-generated Front Ends.
PM = "pattern matching"
+C = "pattern matching and caching"
+CF= "patern matching, caching and tree flattening

TIME (CPU seconds)					
System	Lang.	States	PM	+C	+CF
802-2	CCS	331	3.41	1.30	1.55
Mailer	CCS	1616	9.53	5.02	5.56
ATM	CCS	59614	723.54	189.44	83.56
Emitter	LOTOS	5571	361.35	58.69	55.06
Railway	LOTOS	19724	1244.61	363.05	171.74
Railway	PCCS	11905	2081.37	385.53	319.23

Table 2: Space Usage for PAC-generated Front Ends.
PM = "pattern matching"
+C = "pattern matching and caching"
+CF= "patern matching, caching and tree flattening

SPACE (max process size in MB)					
System	Lang.	States	PM	+C	+CF
802-2	CCS	331	8.848	8.000	9.040
Mailer	CCS	1616	10.000	10.768	11.476
ATM	CCS	59614	54.846	42.144	54.756
Emitter	LOTOS	5571	15.256	14.932	15.128
Railway	LOTOS	19724	38.368	34.016	38.880
Railway	PCCS	11905	21.656	38.492	43.180

It should be noted that the original hand-written CWB semantic routines employed the same pattern-oriented approach to the calculation of semantic information, although neither call caching nor tree flattening were used.

5 Conclusions and Directions for Future Work

This paper proposes a generic framework for the automatic verification of finite-state systems and shows how efficient tool support may be given for it. The framework consists of three basic concepts: (pre)bisimulations as a basis for refinement, the modal mu-calculus as a basis for model checking, and structural operational semantics as a basis for defining the semantics of design notations. The Concurrency Workbench of North Carolina and the Process Algebra Compiler of North Carolina exploit this framework to provide efficient yet easily customizable tool support based on these notions.

In the future we would like to investigate techniques for improving the space utilization of PAC-generated front ends. Recent work [3] also points to an abstract basis for model checking that circumvents the need for defining translations in the mu-calculus, and we would like to investigate the development of a model-checker generator based on these ideas. It could also be fruitful to look into the provision of generic support for symbolic approaches, such as those oriented around Binary Decision Diagrams [8]; steps in this direction may be found in [4]. Finally, it would also be interesting to investigate techniques for generically analyzing other kinds of systems, including those that pass values, exhibit real-time behavior, and have probabilistic aspects to their functioning.

References

[1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[2] G. Bhat and R. Cleaveland. Efficient local model checking for fragments of the modal μ-calculus. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 107–126, Passau, Germany, March 1996. Springer-Verlag.

[3] G. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In *Eleventh Annual Symposium on Logic in Computer Science (LICS '96)*, pages 304–312, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[4] B. Bloom and A. Dsouza. Generating BDD models for process algebra terms. In P. Wolper, editor, *Computer Aided Verification (CAV '95)*, volume 939 of *Lecture Notes in Computer Science*, Liège, Belgium, June 1995. Springer-Verlag.

[5] B. Bloom, S. Istrail, and A. Meyer. Bisimulation can't be traced. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages (POPL '88)*, pages 229–239, San Diego, January 1988. IEEE Computer Society Press.

[6] G. Brebner. Private communication.

[7] G. Bruns. *Distributed Systems Analysis with CCS*. Prentice-Hall, London, 1997.

[8] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[9] CCITT. CCITT recommendation Z.100: Specification and description language SDL. Technical report, ITU General Secretariat, 1988.

[10] U. Celikkan. *Semantic Preorders in the Automated Verification of Concurrent Systems*. PhD thesis, North Carolina State University, 1995.

[11] U. Celikkan and R. Cleaveland. Generating diagnostic information for behavioral preorders. *Distributed Computing*, 9:61–75, 1995.

[12] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[13] E.M. Clarke and J.M. Wing. Formal methods : state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[14] R. Cleaveland, G. Luettgen, V. Natarajan, and S. Sims. Modeling and verifying distributed systems using priorities: A case study. *Software Concepts and Tools*, 17:50–62, 1996.

[15] R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. In E. Brinksma, R. Cleaveland, K.G. Larsen, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 153–173, Aarhus, Denmark, May 1995. Springer-Verlag.

[16] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[17] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[18] R. Cleaveland and S. Smolka. Strategic directions in concurrency research. *ACM Computing Surveys*, 28(4):607–625, December 1996.

[19] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2:121–147, 1993.

[20] W. Elseaidy, R. Cleaveland, and J. Baugh. Modeling and verifying active structural control systems. *Science of Computer Programming*, To appear.

[21] E.A. Emerson, C. Jutla, and A.P. Sistla. On model-checking for fragments of μ-calculus. In C. Courcoubetis, editor, *Computer Aided Verification (CAV '93)*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396, Elounda, Greece, June/July 1993. Springer-Verlag.

[22] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Symposium on Logic in Computer Science (LICS '86)*, pages 267–278, Cambridge, Massachusetts, June 1986. IEEE Computer Society Press.

[23] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219–236, 1989/1990.

[24] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[25] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[26] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[27] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, Berlin, 1992.

[28] R. Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.

[29] International Standards Organization. LOTOS – a formal description technique based on the temporal ordering of observational behavior. Technical Report ISO/TC 97/SC 21 N 1573, ISO, February 1987.

[30] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

[31] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.