Strategic Directions in Concurrency Research

RANCE CLEAVELAND

Department of Computer Science, North Carolina State University, Raleigh, NC ⟨rance@csc.ncsu.edu⟩

SCOTT A. SMOLKA ET AL.[1]

Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY ⟨sas@cs.sunysb.edu⟩

1. INTRODUCTION

Concurrency is concerned with the fundamental aspects of systems of multiple, simultaneously active computing agents that interact with one another. This notion is intended to cover a wide range of system architectures, from tightly coupled, mostly synchronous *parallel* systems, to loosely coupled, largely asynchronous *distributed* systems. We refer to systems exhibiting concurrency as *concurrent systems*, and we call computer programs written for concurrent systems *concurrent programs*.

Concurrency has emerged as a separate area of research in response to the intrinsic difficulties experienced by designers of concurrent systems in diverse fields within computer science. These difficulties stem in general from two main sources. The first is that the presence of multiple threads of control can lead to subtle and often unanticipated interactions between agents—in particular, issues such as interference, race conditions, deadlock, and livelock are peculiar to concurrent programming. The second arises from the fact that many concurrent systems, such as operating systems and distributed databases, are *reactive*, meaning that they are designed to engage in an ongoing series of interactions with their environments. Unlike traditional sequential programs, reactive systems should not terminate, and consequently traditional notions of correctness that rely on relating inputs to expected outputs upon termination no longer apply. Developing mathematical frameworks that accurately and tractably account for these phenomena has been a major goal of researchers in concurrency.

[1] Working Group members include Rajeev Alur (University of California, Berkeley), Jos Baeten (C.M. Technical University, Eindhoven), Jan A. Bergstra (University of Amsterdam), Eike Best (University of Hildesheim), Rance Cleaveland (North Carolina State University), Rocco De Nicola (University of Firenze), Helen Gill (Mitre Corp.), Roberto Gorrieri (University of Bologna), Mohamed G. Gouda (University of Texas, Austin), Jan Friso Groote (CWI), Tom A. Henzinger (University of California, Berkeley), C.A.R. Hoare (Oxford University), Maj. David Luginbuhl (AFOSR), Albert Meyer (MIT), Dale Miller (University of Pennsylvania), Jayadev Misra (University of Texas, Austin), Faron Moller (University of Uppsala), Ugo Montanari (University of Pisa), Amir Pnueli (Weizmann University), Sanjiva Prasad (IIT Delhi), Vaughan R. Pratt (Stanford University), Joseph Sifakis (VERIMAG Grenoble), Scott A. Smolka—Chair (State University of New York, Stony Brook), Bernhard Steffen (University of Passau), Bent Thomsen (ICL), Frits Vaandrager (University of Nijmegen), Moshe Vardi (Rice University), and Pierre Wolper (University of Liège).

The editorial activities of Rance Cleaveland have been supported in part by ONR grant N00014-92-J-1582, NSF grants CCR-9257963 and CCR-9402807, and AFOSR grant F49620-95-1-0508; and those of Scott Smolka by NSF grant CCR-9505562 and AFOSR grants F49620-93-1-0250 and F49620-95-1-0508.

This report focuses on research into models and logics for concurrency and their application in specifying, verifying, and implementing concurrent systems. This general area has become known as *concurrency theory*, and its roots may be traced back to the 1960s [Dijkstra 1968; Petri 1962]. Our aim is to survey the rich collection of mature theories and models for concurrency that exist; to review the powerful specification, design, and verification methods and tools that have been developed; and to highlight ongoing active areas of research and suggest fruitful directions for future investigation.

By focusing on concurrency theory and its use in verification, we necessarily omit consideration of other concurrency-related topics such as concurrency control in database systems, concurrent program debugging, operating systems, distributed system architecture, and real-time systems. The interested reader is referred to other working group reports, which address many of these topics.

The remainder of this report develops along the following lines. The next section discusses the current state of concurrency research; the section following then presents strategic directions for future investigation. Finally, some concluding remarks are given.

2. CONCURRENCY RESEARCH—THE STATE OF THE ART

This section gives an overview of the current state of research in concurrency. It reviews work on mathematical models of concurrency, discusses results obtained in system specification and verification, and surveys the practical impact of concurrency theory.

2.1 Models for Concurrency

As the introduction suggests, concurrent systems differ in essential ways from sequential ones. In order to characterize the fundamental aspects of these systems, researchers have developed a variety of mathematical theories of concurrency over the past two decades. It should first be noted, however, that there is a basic agreement between the different schools of thought: unlike the semantic theories developed for sequential computation, models of concurrent systems cannot be viewed as mappings from inputs to outputs, owing to the reactive nature of such systems as well as their capacity for nondeterminism. Instead, it is necessary to take into account the fact that systems interact with other systems and hence may exhibit patterns of stimulus/response relationships that vary over time. Consequently, rather than adopting the notion of input/output mapping as primitive, theories of concurrency are generally based on the assumption that certain aspects of system behavior are *atomic*. Systems then execute by engaging in patterns of these "indivisible" events.

The point of departure for different theories of concurrency lies in what constitute "patterns" of events. More specifically, models that have been developed may be classified on the basis of the stances they adopt with respect to three basic dichotomies:

—intensionality versus extensionality,
—interleaving versus true concurrency, and
—branching time versus linear time.

The first dichotomy also arises in the semantics of sequentiality, but the last two are peculiar to concurrency. We elaborate on each in the following.

Intensionality versus extensionality. Intensional models focus on describing what systems do, whereas extensional models are based on what an outside observer sees. Consequently, intensional theories are sometimes referred to as *operational*, and extensional ones are called *denotational*, the denotation of a process being the set of observations it can engender.

Intensional theories model systems in terms of states and transitions between states. In *labeled transition systems* [Keller 1976] transitions are decorated with atomic actions representing interactions with the environment; these have been studied extensively in the context of *process algebras* such as ACP [Baeten and Weijland 1990], CCS [Milner 1989], and CSP [Hoare 1985]. *I/O automata* [Lynch and Tuttle 1989] also follow this approach, but they also allow distinctions to be made between different kinds of actions (e.g., input and output). *Petri nets* [Reisig 1985] extend the state/transition paradigm by allowing states to be "distributed" among different locations. UNITY [Chandy and Misra 1988] adopts an imperative style, with transitions corresponding to the (atomic) execution of conditional assignment statements.

In contrast, extensional models first define a notion of observation and then represent systems in terms of the observations that may be made of them. One of the most basic observations about a system is a *trace*, or sequence of atomic actions executed by a system [Hoare 1985]. Elaborations of this model include *acceptance trees* [Hennessy 1988] and *failure sets* [Brookes et al. 1984], both of which decorate basic trace information with information about what stimuli systems may respond to after performing a sequence. *Synchronization trees* [Milner 1980] encode system behavior as a tree with edges labeled by actions. *Mazurkiewicz traces* [Mazurkiewicz 1987] include an independence relation between actions to capture possible concurrency. *Kahn nets* [Kahn 1974] define the behavior of dataflow systems, whose components are given by I/O equations, using fixed-point theory. Other models, such as *pomsets* [Pratt 1986], encode concurrency information using partial orders on atomic events possibly enriched with a conflict relation (*event structures* [Winskel 1989]) to capture information about choices made during execution.

Interleaving versus true concurrency. Interleaving models "reduce" concurrency to nondeterminism by treating the parallel execution of actions as the choice between their sequentializations. This view may be termed a "uniprocessor" approach to concurrency. To avoid anomalies having to do with process starvation, *fairness constraints* are sometimes imposed to ensure that individual processes "make progress" if they are capable of doing so [Francez 1986]. Theories such as ACP, CCS, and CSP employ interleaving, as do I/O-automata and UNITY; both of the latter also include mechanisms for enforcing fairness constraints. The trace-based and synchronization-tree extensional models also model concurrency via interleaving.

In contrast, truly concurrent models treat concurrency as a primitive notion; the behavior of systems is represented in terms of the causal relations among the events performed at different "locations" in a system. Petri nets represent concurrency in this manner, as do Mazurkiewicz traces, Kahn nets, pomsets, and event structures.

Branching time versus linear time. The difference between branching-time and linear-time models lies in their treatment of the choices that systems face during their execution. Linear-time models describe concurrent systems in terms of the sets of their possible (partial) runs, whereas in branching-time models the points at which different computations diverge from one another is recorded. Thus traces and pomsets are linear-time models and synchronization trees and event structures are branching-time models.

Discussion. Different decisions about the three dichotomies are appropriate for different purposes. Extensional models provide a useful basis for explaining system behavior, although intensional ones are often more amenable to automated analysis, as they typically give rise to (finite-)state machines. An interleaving semantics is useful for specifying systems, whereas a truly concurrent semantics might be the basis for describing possible implementations where, for example, performance issues are of interest. A branching-time semantics is useful for modeling future behavior of a system, whereas linear time suffices for describing execution histories.

Although traditional theories have concerned themselves with modeling choice and concurrency, more recent work has focused on extending them to treat other aspects of system behavior, including real-time, probability, and security. This point is addressed in more detail in Section 3.1.

2.2 Verification Formalisms

In dealing with concurrent programs care must be taken when defining the notion of "correctness." Traditional (deterministic) sequential programs may be viewed as (partial) functions from inputs to outputs; in such a setting specifications may be given as a pair consisting of a precondition describing the "allowed" inputs and a postcondition describing the desired results for these inputs. This notion of specification remains appropriate for a concurrent program representing a "parallelized" version of some sequential program; in this case, concurrency (parallelism) has been introduced solely for performance purposes. For reactive, nonterminating, and potentially nondeterministic concurrent systems, however, this approach is too limited. This section presents alternative notions that have been developed and discusses related approaches for reasoning about systems.

2.2.1 *Specification.* Two general schools of thought have emerged regarding appropriate mechanisms for specifying concurrent systems. One emphasizes the use of logical formulas to encode properties of interest; another uses "high-level" systems as specifications for lower-level ones. What follows elaborates on each of these.

Logics for concurrency. In a seminal paper, Lamport [1977] argued that the requirements that designers wish to impose on reactive systems fall into two categories. *Safety properties* state that "something bad never happens"; a system meeting such a property must not engage in the proscribed activity. *Liveness properties*, on the other hand, state that "something good eventually happens"; to satisfy such a property, a system must engage in some desired activity. Although informal, this classification has proved enormously useful, and formalizing it has been a main motivation for much of the work done on specification and verification of concurrent systems. Some of this work has aimed at semantic characterizations of these properties [Alpern and Schneider 1985]. Others have developed logics that allow the precise formulation of safety and liveness properties. The most widely studied are *temporal logics*, which were first introduced into the computer-science community by Pnueli [1977] and support the formulation of properties of system behavior over time. The remainder of this section reviews some of the research into temporal logic.

The dichotomies presented earlier in modeling concurrency also reveal themselves in the development of temporal logics. In particular, two schools of temporal logic have emerged [Emerson and Halpern 1986]:

—*Linear-time* logics permit properties to be stated about the execution sequences a system exhibits.

—*Branching-time* logics allow users to write formulas that include some sensitivity to the choices available to a system during its execution.

Numerous variants of linear-time and branching-time temporal logic have been proposed, as researchers have investigated operators that ease the formulation of properties in different settings [Manna and Pnueli 1991]. The expressiveness of these formalisms has also been compared and contrasted [Emerson and Halpern 1986], and a (in some sense) canonically expressive temporal logic, the modal mu-calculus [Kozen 1983], has been developed.

The other two dichotomies—intensional versus extensional and interleaving versus true concurrency—remain relatively unexplored. Traditional temporal logics have generally adopted an extensional view of system behavior and an interleaving model of concurrency, although recent work has explored logics for true concurrency [Thiagarajan 1994].

Finally, other logics have also been developed for reasoning about concurrent systems, including various dynamic logics and logics of knowledge. The former permit the inclusion of programs inside formulas [Peleg 1987], and the latter allow users to express the understanding that individual agents have of other agents' states at a given point in time [Halpern and Moses 1990; Halpern and Zuck 1992].

Behavioral relations. Another popular approach to specifying concurrent systems involves the use of *behavioral equivalences and preorders* to relate specifications and implementations. In this framework, which was introduced by Milner [1980] and has been extensively explored by researchers in process algebra [Baetan and Weijland 1990; Hoare 1985; Milner 1989], specifications and implementations are given in the same notation; the former describes the desired high-level behavior, and the latter provides lower-level details indicating how this behavior is to be achieved. In equivalence-based methodologies, proving an implementation correct amounts to establishing that it behaves "the same as" its specification; in those based on preorders, one instead shows that the implementation provides "at least" the behavior dictated by the specification.

In support of this approach, a number of different equivalences and preorders have been proposed based on what aspects of system behavior should be observable. Relations may be classified on the basis of the degree to which they abstract away from internal details (viz. the intensional/extensional dichotomy) of system descriptions; the amount of sensitivity they display to choices systems make during their execution (viz. the linear/branching-time dichotomy); and the stance they adopt with respect to interleaving versus true concurrency. For example, bisimulation equivalence [Milner 1980] is a branching-time, interleaving-based, intensional equivalence, whereas observational equivalence [Milner 1980] is a branching-time, interleaving-based extensional equivalence. Other noteworthy relations include the failures/testing relations (linear-time, interleaving, extensional) [Brookes et al. 1984; De Nicola and Hennessy 1984] and pomset equivalence (linear-time, truly concurrent, extensional) [Pratt 1986]. Interested readers are referred to van Glabbeek [1990] for a detailed study of the relationship among different equivalences.

These relations may also be used to bridge the gap between the intensional and extensional models of concurrency described previously. That is, in order to define a relation over intensional models, one first selects the extensional, or "observable," information that processes may exhibit and then uses this as the basis for relating models.

Comparison. The chief distinction between the two specification approaches arises in the amount of information they require users to specify. Logic-based approaches support very loose specifications, as one is allowed to identify single properties that systems should have. System-based approaches require fairly complete specifications of required observable behavior, although in this respect preorders generally allow looser specifications than equivalences. On the other hand, behavioral relations provide support for stepwise refinement of systems as well as compositional approaches to analyzing system behavior, which temporal logics in general do not, since the specification and system notations differ.

Connections between the two approaches have also been explored. In particular, a temporal logic induces an equivalence on systems as follows: two systems are equivalent if and only if they satisfy the same formulas. Using this framework, relationships between different linear- and branching-time logics and equivalences have been established [Hennessy and Milner 1985; Browne et al. 1988].

2.2.2 *Verification Methodologies and Algorithms.* *Verification* is the process of establishing that a system satisfies a specification given for it. The previous subsections described different approaches to specifying concurrent systems; this one gives an overview of techniques for establishing that systems meet such specifications.

Proof systems. One verification methodology involves using axioms and inference rules to prove that systems satisfy specifications. Researchers have investigated proof systems for several different specification formalisms, including variants of temporal logic and a variety of behavioral relations, and we briefly review these here.

Early proof systems aimed at generalizing frameworks for sequential programs to cope with concurrency [Owicki and Gries 1976]. In particular, specifications were given as pairs of conditions circumscribing the allowed inputs and desired outputs, and consequently this approach was not appropriate for reactive systems. However, it was in the context of this work that *interference* emerged as a crucial complicating factor in verifying concurrent programs. To cope with nonterminating systems, other researchers focused their attention on developing proof methodologies for establishing that systems satisfied different specifications given as *invariants* [Lamport 1980].

Traditional temporal-logic-based proof systems rely on proving implications between temporal formulas [Pnueli 1977]. To prove a system correct, one first translates it into an "equivalent" temporal formula and proves that this formula entails the specification. The virtue of this approach is that one can use (existing) axiomatizations of temporal logic to conduct proofs; the disadvantage is the translation requirement, which usually obscures the structure of the system. More recent proof systems such as UNITY [Chandy and Misra 1988] adopt proof rules for specific system notations and temporal logics; other work has been devoted to the development of domain-specific rules for establishing certain classes of properties [Manna and Pnueli 1991, 1995].

Behavioral-relation-based proof systems have typically been *algebraic* (hence the term *process algebra*) [Milner 1989; Baeten and Weijland 1990]; in order to prove that two systems are equivalent, one uses equational reasoning. Sound and complete proof systems have been devised for a number of different equivalences, preorders, and description languages.

Algorithms. Finite-state systems turn out to be amenable to *automatic* verification, since their observable behavior can be finitely represented. These systems arise in practice in areas such as hardware design and communication protocols; this fact has spurred interest in the development of verification algorithms for temporal logic and relation-based specifications.

The task of determining automatically whether a system satisfies a temporal formula is usually referred to as *model checking*. One may identify two basic approaches to model checking. The first, which came out of the branching-time logic community, relies on an analysis of the structure of the formula to determine which system states satisfy the formula [Clarke et al. 1986; Cleaveland and Steffen 1993]. The second, which arose from the linear-time community, is automaton-based [Vardi and Wolper 1986]; one constructs an automaton from the negation of the formula in question and then determines whether the "product" of this automaton and the automaton representing the system is empty (if so, the system is correct). The two approaches turn out to be related, and indeed automaton-based approaches have been developed for branching-time logics and "structure-based" ones have been devised for linear-time logics [Bernholtz et al. 1994; Bhat et al. 1995]. The time complexities of the best algorithms in each case are proportional to the product of the number of system states and the size of the formula (branching-time) or an exponential of the size of the formula (linear-time).

Algorithms have also been devised for determining whether systems are related by semantic relations. Traditional approaches for calculating equivalences combine partition-based algorithms for bisimulation equivalence [Paige and Tarjan 1987; Kanellakis and Smolka 1990] with automaton transformations that modify the automata corresponding to the systems being checked [Cleaveland et al. 1993]. Methods for computing preorders follow a similar scheme, with the "base" relation being a variant of the simulation preorder.

The key impediment to the practical application of these algorithms is the *state-explosion problem*. In general, the number of states a system has will be exponential in the number of concurrent processes; thus, as the number of processes grows, enumerating all possible system states rapidly becomes infeasible. Much of the recent work on model and relation checking has been devoted to techniques for ameliorating the effects of state explosion. Some of the techniques that have been developed include:

—*symbolic representations* of state spaces via, for example, *binary decision diagrams* [Burch et al. 1992], support the compact encoding of system states;

—*on-the-fly algorithms* rely on a demand-driven generation of states to avoid the construction of irrelevant system configurations [Andersen 1994; Bhat et al 1995]; and

—*redundancy elimination* strives to reduce the number of redundant system states that algorithms analyze, and includes partial-order reduction techniques [Holzmann et al. 1996], semantic minimization [Roy and de Simone 1990], and symmetry-based approaches [Clarke et al. 1993; Emerson and Sistla 1993].

Finally, researchers have also developed techniques for handling certain kinds of infinite-state systems, including context-free processes [Baeten et al. 1993] and those based on dense real time [Alur and Dill 1994].

2.2.3 *Tools.* The past decade has also witnessed the development of tools that use concurrency theory as a basis for automating the design and analysis of concurrent systems. What follows is essentially a categorization of tools based on the degree of interaction demanded from the user during the verification process. The interested reader is also referred to the report of the formal methods working group ⟨/a116-clarke/⟩ for additional tool information.

Interactive tools. These typically employ a theory for reasoning about concurrent systems in conjunction with a deduction engine to allow users to conduct proofs that systems enjoy certain properties. The virtue of such tools is that they permit the analysis of systems—such as those that are parameterized or manipulate data—that lie beyond the scope of fully automated tools; their disadvantage is that they can require substantial user interaction. Sample tools include PVS [Owre et al. 1992] and Coq/μCRL [Bezem and Groote 1993], both of which enrich type theories for manipulating values with notions of concurrency and which have been used to prove the correctness of parameterized systems and distributed algorithms and protocols; STeP [Alur and Henzinger 1996a], which provides automated support for establishing that concurrent systems satisfy temporal formulas as well as a decision procedure for temporal logic and a model checker; and PAM [Lin 1991], which oversees the construction of algebraic proofs of equivalence between terms in a process algebra that the user specifies.

Automatic tools. These provide implementations of one or more of the verification algorithms discussed above. The chief virtue of these tools is that they are automatic; the disadvantage is that the classes of systems that may be analyzed are restricted to those having some appropriate finitary representation. Sample tools include SPIN [Alur and Henzinger 1996a] and COSPAN [Alur and Henzinger 1996a], which support model checking in linear-time formalisms; the Concurrency Workbench [Cleaveland et al. 1993], FDR [Roscoe 1994], and AUTO [Roy and de Simone 1990], which compute various semantic relations and (in the case of the Workbench) implement branching-time model checkers; and Xesar [Queille and Sifakis 1982] and SMV [Alur and Henzinger 1996a], which support branching-time model checking. UV [Kaltenbach 1996] is a model checker for UNITY that combines automatic checking of linear-time temporal properties with interactive features that allow a user to supply hints for speeding up the checking procedure. PEP [Margaria and Steffen 1996] allows the automatic analysis of Petri-net-based systems. Other tools have been developed for verifying real-time and hybrid systems; these include HyTech [Henzinger et al. 1995], Kronos [Daws and Yovine 1995] and UppAal [Margaria and Steffen 1996]. Some tools, such as Statemate [Harel et al. 1990] and the Concurrency Factory [Alur and Henzinger 1996a], also provide support for the generation of code from system models.

Metatools. The multitude of concurrency models and verification formalisms has also spurred interest in the development of metatools that support the customization and collaborative use of existing tools. Examples include PAM [Lin 1991], which allows users to change the process algebra and equational axiomatization being used; the Process Algebra Compiler [Cleaveland et al. 1995], which can be used to generate new front ends for the Concurrency Workbench; and MetaFRAME [Alur and Henzinger 1996a], which provides a framework for integrating tools so that they may be used in conjunction with one another.

2.2.4 *Applications.* The remainder of Section 2 reviews some of the applications in the area of programming languages and system verification to which concurrency theory has been put.

Concurrent programming languages. Traditionally, the inclusion of concurrency into programming languages has been done in a somewhat ad hoc manner; the general paradigm has been to augment sequential languages such as C with facilities for using operating-system-supplied primitives such as process creation and semaphores. This meant that programs written in these languages were not portable across different operating systems; in addition, the low-level nature of system calls led to programs that were difficult to maintain.

Programming language researchers have over the past decade begun to investigate the design of programming constructs based on models of concurrency described above as a means of remedying this problem. Examples include OCCAM [Inmos International 1988], which arose out of the work done on CSP [Hoare 1985]; synchronous programming languages such as ESTEREL [Berry and Ganthier 1992] and LUSTRE [Halbwachs 1993], which evolved from dataflow models [Kahn 1974; Kahn and MacQueen 1977]; and PICT [Pierce and Turner 1995], CML [Reppy 1992], and Facile [Thomsen et al. 1996], which are functional concurrent programming languages based on process algebras (CCS [Milner 1989], π-calculus [Milner et al. 1992]), and their higher-order extensions (Higher Order π-calculus [Sangiorgi 1992] and CHOCS [Thomsen 1995]). Other languages, such as Linda [Carriero and Gelernter 1989], have been given rigorous semantics using techniques borrowed from concurrency theory [Jagannathan and Weeks 1994]. See Section 3.4 for a discussion of the use of concurrency models in programming language design as a strategic research direction.

System verification. Significant case studies to which concurrency theory has been brought successfully to bear are too numerous to list completely. Those that follow are taken from a forthcoming special issue of the journal *Science of Computer Programming* devoted to industrial applications; the interested reader is also referred to the formal methods report for others.

—The XTP and DREX communications protocols are proved correct, as is Bull's Flowbus architecture.

—Several hardware protocols are formalized and verified, including the Futurebus cache coherence protocol, the PCI localbus, and the I^2C bus.

—The timing properties of earthquake-resistant active structural control systems are shown to satisfy given bounds.

—A failure recovery protocol for the Heathrow air-traffic information system is described and verified using modal process logic.

—The timing constraints between interlockings guarding the safety of British railyards are analyzed and debugged.

—A commonly used distributed leader election algorithm for unidirectional ring networks is shown to be incorrect, and a corrected version is verified.

—Formal approaches to eliminating undesired feature interactions in telephony services are discussed and applied.

—Grid protocols being used in multiprocessor environments are proved correct.

3. STRATEGIC DIRECTIONS

We present a number of strategic directions for future concurrency research, each of which is accompanied by a "grand challenge." The challenges are intended to serve as a yardstick for progress made along the various research directions.

3.1 Beyond Correctness

In the sequential world, programs are typically modeled as (partial) functions mapping inputs to outputs, and verification formalisms generally require the use of preconditions (to constrain inputs) and postconditions (to constrain outputs). A correct sequential program should also terminate for all inputs. See Jones [1992] for comprehensive coverage of proof techniques for sequential programs. Researchers have also focused on certain "nonfunctional" aspects of programs, most notably efficiency (time and space). See the ACM SDCR Theory of Computation working group report in this issue and at http://geisel.csl.uiuc.edu/~loui/complete.html.

For concurrent systems, particularly those of the reactive variety, many nonfunctional requirements are of interest to system builders. These include the following.

Timeliness. Concurrent systems are often subject to timing constraints; for example, if the amount of water in a water pump exceeds a certain level, then the pump should shut off within *t* seconds. Embedded systems, which interact with their environment through sensors and actuators, are a typical example of a class of concurrent systems for which real-time behavior is a major concern and performance requirements must be met.

Fault tolerance. Concurrent systems must often provide reliable service despite the occurrence of various types of failures. Such fault tolerance is invariably achieved through redundancy of system components.

Probability. Although not a requirement per se, equipping specifications with probability information can be useful for specifying system fault characteristics, for example, the rate at which a faulty communications channel drops messages.

Continuous behavior. A hybrid system consists of a nontrivial mixture of discrete and continuous components, such as a digital controller that controls the continuous movement of control rods in a nuclear reactor.

Security. As systems become more distributed and more accessible, it is increasingly important to make them resistant to passive or active misuse by intruders.

Mobility. Mobile processes are processes that can exchange communication links, thereby introducing the possibility of dynamically reconfigurable network architectures. Higher-order processes, which are closely related to mobile processes, permit process passing.

Note that these different types of requirements are sometimes contradictory. Reliability, for instance, requires the use of additional resources, which may degrade system performance and thus endanger timeliness (see Kanellakis and Shvartsman [1992] for a treatment of the reliability versus efficiency tradeoff in parallel programs).

Recognizing their importance, concurrency researchers have been turning their attention more and more to the nonfunctional requirements of concurrent programs. For example, substantial progress has been made on models and logics for real-time [De Bakker et al. 1992], probability [Hansson 1994, Larsen and Skou 1992, and van Glabbeek et al. 1995], mobility [Milner et al. 1992, Sangiorgi 1992, Thomsen 1995], and hybrid systems [Pnueli and Sifakis 1995].

Challenge. Solidify our understanding of phenomena such as real-time and mobility and at the same time develop new formalisms for those phenomena that remain largely unexplored (e.g., security, although see Focardi and Gorrieri [1995] and Roscoe [1995]). Achieving these goals is likely to necessitate interaction with researchers from other computer-science disciplines (security experts, system engineers, etc.), and from noncomputer-science disciplines such as electrical and mechanical engineering and control theory.

We should also strive to develop *semantic partnerships* between formalisms that would facilitate the construction of a model capturing the requirements most relevant to a given problem, for example, a model that embodies timing and probability information in the context of mobile systems (see also Section 3.2, which treats, in greater detail, the challenge of taxonomizing and unifying semantic models of concurrency).

Finally, these formalisms must be brought out of the "test-tube" environment and applied in an integrated way to the specification, verification, and design of complex real-life systems. A concrete challenge here is the formal analysis of the emerging micro-electromechanical systems (MEMS) technology, which presents a host of modeling challenges. The interested reader should see Gabriel [1995].

Another challenging application is air-traffic control, whose safety-critical nature and high degree of concurrency make it an ideal test-case for concurrency research. The utility of a formal approach is already evident in the work of Heimdahl and Leveson [1996], where a formal requirements specification of the commercial traffic collision avoidance system (TCAS II) was produced and tested. Greater challenges lie in the future, particularly if today's centralized solutions that require airplanes to use specific approach patterns to runways are abandoned in favor of decentralized "free-flight" solutions. Although free-flight solutions can improve efficiency and solve congestion problems, the complexity of their design demands use of formal approaches. Principal to this design are problems of coordination and conflict resolution among multiple agents [Sastry et al. 1995].

3.2 A Unifying Semantic Framework of Concurrency

> The progress of science involves a constant interplay between diversification and unification. Diversification extends the boundaries of science to cover new and wider ranges of phenomena; successful unification reveals that a range of experimentally validated theories are no more than particular cases of some more general principle. The cycle continues when the general principle suggests further specialisations for experimental investigation.
>
> From C.A.R. Hoare's position statement (http://www.acm.org/surveys/1996/HoareUnifying/)

During the past twenty years, concurrency theory has produced a mature but loose collection of models, theorems, algorithms, and tools. The collection is mature because it is based on a solid mathematical foundation (see Section 2.1) and has made possible documented successes in system design and analysis (see Section 2.2.4). The collection is loose because a theoretical result or a practical application is typically carried out within a particular formalism, whose idiosyncrasies may support the result or application in undeclared ways.

Overall, myriad formalisms have led to healthy diversity rather than fragmentation of the discipline: existing concurrency theories have proved suitable for a wide range of application domains, and comparative concurrency theory[2] has identified deep mathematical relationships among individual theories.

Yet potential users, such as designers of communication protocols and embedded systems, have been reluctant in applying concurrency-theoretic methods. This reluctance is reinforced by the perceived need of having to buy into one particular formalism and tool from what must seem, to the bystander, a bewildering array of choices.

Challenge. Develop a systematic and coherent way of presenting concurrency theory to its potential users. This should be achieved by a simple uniform framework that permits an application-oriented taxonomy of the major models of concurrency and a structured organization of the core results.

A uniform framework for concurrency will aid not only potential users of concurrency-theory-based tools, but also students and researchers. In particular, such a framework could provide a basis for concurrency education in the computer science curriculum (see Section 3.5). It could also aid the development of new methods and tools for the design and analysis of heterogeneous systems, consisting of synchronous and asynchronous, discrete and continuous, hardware and software components. For example, a sufficiently broad framework could support the implementation of asynchronous protocols using synchronous circuits.

The literature already contains many proposals for highly abstract, general-purpose models of concurrency,[3] and the uniform framework we seek could very well evolve from this body of work. To make this discussion more concrete and to illustrate the rich diversity among the proposed models, we briefly describe three of these efforts.

A *reactive module* [Alur and Henzinger 1996b] is a kind of state transition system that has been proposed as a uniform framework for synchronous and asynchronous computation in the context of hardware-software codesign. It supports an abstraction operator that

[2] Winskel [1987], van Glabbeek [1990], Olderog [1991], and Abramsky [1996].

[3] Groote and Vaandrager [1992], Best et al. [1992], Milner [1993], Glabbeek [1993], Aceto et al. [1994], Winskel and Nielsen [1995], Mifsud et al. [1995], Montanari and Rossi [1996], Abramsky [1996], and Alur and Henzinger [1996b].

can be used to collapse an arbitrary number of consecutive computation steps into a single step, thereby providing the ability to turn an asynchronous system into a synchronous one (temporal scalability). It supports a hiding operator that changes external variables into internal ones, which can be used to change a synchronous system into an asynchronous one (spatial scalability). Finally, the model is equipped with a natural notion of composition that permits compositional reasoning, that is, the ability to reason about a composite system in terms of the system components.

Causal computing [Montanari and Rossi 1996] is a general framework for concurrent computation based on the "true concurrency" concurrency model. In the causal approach, concurrent computations are defined as those equivalence classes of sequential computations that are obtained by executing concurrent events in any order. A *causal program* consists of a set of rewriting rules, events are rule applications, and simultaneous rewritings correspond to concurrent events. Causal computing has been used as a model of computation for process algebras, constraint and logic programming, term and graph rewriting, and mobile and coordination systems; please see the position statement of Ugo Montanari ⟨a51-montanari⟩, see Table of Contents for Volume 28(4es), this issue for a complete address and a closer look at causal computing.

A *Chu space* [Gupta and Pratt 1993] is an event-state-symmetric generalization of the notion of event structure consisting of a set of events, a set of states, and a binary relation of *occurrence* between them. The interpretation of process algebra over event structures extends straightforwardly to Chu spaces, whereas the language of process algebra expands to include the operations of linear logic, compatibly interpreted. Chu spaces may be equivalently formulated in terms of unfolded Petri nets, propositional theories, or history-preserving process graphs [van Glabbeek and Plotkin 1995]. Chu spaces are of independent mathematical interest, forming a bicomplete self-dual closed category first studied by Barr and Chu [Barr 1979] that has found applications elsewhere in proof theory, game theory, and Stone duality [Pratt 1995]. Chu spaces are the subject of Vaughan Pratt's position statement, accessible at ⟨/a54-pratt/⟩ (see Table of Contents, Vol. 28(4es) for complete address).

3.3 Design and Verification Methodologies

There are two important observations about the current state in the design and verification of concurrent systems.

—There are many proposed techniques, and guidelines for using these techniques, that differ in quality and applicability. For example, an approach to design and verification based on a true concurrency model may yield greater computational efficacy than one based on interleaving, since the former admits the application of partial-order reduction techniques. Similarly, certain correctness properties (e.g., of the form "from any state is it possible to get to a state satisfying proposition p") are expressible in branching-time temporal logic but not in linear-time temporal logic [Clarke et al. 1995].

—The design and also the verification of many concurrent systems follow a similar paradigm. For example, a number of the applications reported in Section 2.2.4 consisted of first attaining a thorough understanding of the application at hand, then coding the application in the abstract specification language of a model checking or bisimulation checking tool, and finally running the tool on the problem. In some cases, this process was repeated several times due to an initial misunderstanding of some part of the specification, a lack of a successful strategy for dealing with state-space explosion, and so on.

Challenge. Transform the existing array of design and verification techniques into sound and tested methodologies. The resulting methodologies should extend the range of existing techniques to applications orders of magnitude larger in size and complexity. Similar to concerns raised in Section 3.2 regarding a uniform semantic framework for concurrency, we should also seek ways to combine methodologies to better suit the demands of a given application, and, relatedly, develop an application-oriented taxonomy of methodologies.

To produce a next generation of truly usable methodologies, the following issues must be addressed.

Algorithmic support. Further advances (i.e., beyond those listed in Section 2.2.2) are needed to better cope with the state-space explosion problem inherent in concurrent system design and verification. Compositional methods, in which the analysis of a system is decomposed into an analysis of its components, and refinement methods, in which a system is analyzed at varying levels of abstraction, may play a key role. The issue of algorithmic support is addressed more fully in the position statement of Pierre Wolper: ⟨a127-wolper⟩, see the Table of Contents, Vol. 28(4es), this issue.

Tool support. The problems confronting today's tools, such as lack of portability and scalability, need to be addressed. Furthermore, tools should be better integrated into the software engineering lifecycle. Traditionally, software engineering devotes much attention to organizational and procedural issues in software development and relatively little to methods for system analysis; in this respect, it resembles a management discipline rather than an engineering one. Tools based on concurrency theory offer a particularly appropriate starting point for putting the engineering into software engineering.

Technology transfer. The capacity of today's design and verification technology will improve only if subjected to protracted exposure to real-life industrial and government (i.e., defense) applications. The transfer of this technology to these arenas is thus a key issue, and will be facilitated by increased tool support (including improved user interfaces) and education. The role of tool support in technology transfer is the subject of the new Springer-Verlag journal *Software Tools for Technology Transfer*, to begin publication in September 1997. Education should include training in both tool usage and in the concurrency-theoretic concepts underlying the tools (see Section 3.5). Furthermore, one can expect a domino effect: as companies and agencies achieve positive results in the use of the latest design and verification methodologies, more users are likely to follow suit.

3.4 Programming Languages for Concurrency

In Section 2.2.4, we briefly discussed the impact models of concurrency have had on the design of concurrent programming languages. Strategically, we believe that the continued use of concurrency theory—especially as this theory continues to evolve—in the design and implementation of programming languages is an important research direction. The main rationale behind this belief is simple and powerful: the development of programming languages with strong foundations in concurrency theory would blur, and in some cases completely eliminate, the distinction between a system model and a system implementation. As such, the task of designing, coding, and verifying concurrent systems would be significantly eased.

Designing programming languages for concurrency is a nontrivial task. Adding a notion of concurrency to an existing language may break some of the tacit assumptions of the language's design, such as determinism and no (interprocess) interference on variables. Thus, designing a language with an explicit concurrency model from scratch is likely to produce a language with a cleaner and better-understood semantics. Moreover, rooting a programming language in a sufficiently expressive concurrency model could lead to the construction of reusable abstractions for sophisticated synchronization mechanisms, in much the same way as objects are used in sequential systems today.

A concurrency model underlying the design of a parallel programming language might also increase our ability to efficiently port a program across a wide spectrum of system architectures, without unduly restricting the parallelism in the ported version of the program. Concurrency models by their very nature tend to be abstract, meaning that they are largely architecture-independent. In particular, they typically assume no a priori limit on the number of processors available for running processes (but see Gerber and Lee [1994] for an exception), and hence capture a notion of maximal parallelism.

To better address the portability issue, we should more closely examine the interplay between true concurrency models and interleaving-based models (Section 2.1). The former come into play when mapping individual processes to individual processors in the target architecture; the latter are relevant when multiple processes must be mapped to the same processor. A better understanding of this interplay is one of the potential benefits of a unifying framework for concurrency (Section 3.2).

Concurrency theory has also begun to influence the design of type systems for functional concurrent programming. In Nielson and Nielson [1993, 1994] function types of the form $f : t_1 \xrightarrow{b} t_2$ are allowed, where b is a process algebra behavior expression. The type is to be read: function f takes a value of type t_1 and produces a value of type t_2 and in doing so has behavior b. Under some conditions these behaviors will be inferable and "principal" in a sense similar to that of SML [Milner et al. 1990]. A similar approach has been proposed in Nierstrasz [1990] for concurrent object-oriented languages based on process algebra.

Type systems of this nature seem like a particularly promising technique for integrating a concurrency model into a programming language. For instance, we could devise new type systems that guarantee safety and liveness properties in the same way that traditional type systems guarantee safety in calling functions and procedures [Abramsky et al. 1996].

Challenge: Design usable, safe, and secure languages incorporating a well understood concurrency model. The need for such languages is particularly urgent now that mobile agents or applets have started to roam the Internet. Currently, mobile agents are written in languages such as Java, Telescript, and Safe Tcl/Tk, all providing rudimentary support for concurrency in the form of threads and semaphores. There is, however, a growing need to extend the concurrency constructs of these languages or design new languages with more advanced synchronization mechanisms, including synchronous channels, multicast groups and constraint variables. Concurrency theory—in particular, higher-order and mobile theories[4]—can play an important role in achieving these goals. The resulting languages would help system designers analyze sophisticated software systems built using mobile agents.

3.5 Concurrency Education

It is unlikely that concurrency-based design and verification methodologies will gain widespread acceptance until the user community is educated in these methodologies and the concepts underlying them. Concurrency education

[4] Thomsen [1995], Milner et al. [1992], Sangiorgi [1992], and Nielson and Nielson [1994].

should start in the undergraduate curriculum. Moreover, as aptly pointed out in Rosenberg [1995], concurrency, whether we recognize it or not, pervades many areas of computer science (e.g., hardware, operating systems, languages, compilers, and algorithms) and an undergraduate well versed in concurrency basics will be better prepared for these courses.

Challenge. Introduce a course on concurrency basics into the official ACM Undergraduate Computer Science Curriculum by the year 2000.

An undergraduate course on concurrency basics (a course entitled "Concurrency Theory" may sound too intimidating) should include coverage of the following essential concepts.

—*What is concurrency?* provide basic definition and show how this concept pervades the undergraduate computer science curriculum.

—*State transition systems:* structural (graph isomorphism) versus behavioral (bisimilarity, language equivalence) equivalence.

—*Systems specification:* CCS [Milner 1989], CSP [Hoare 1985], Petri nets [Reisig 1985], statecharts [Harel 1987].

—*From syntax to state transition systems:* structural operational semantics [Plotkin 1981], compositionality, refinement.

—*Requirements specification:* finite executions and invariant assertions; infinite executions, fairness, and temporal assertions [Manna and Pnueli 1991].

—*Requirements verification:* model checking, proof checking, behavioral relation checking.

—*Case studies:* choose sampling of applications from the undergraduate computer science curriculum, for example, circuit design and avoidance of race conditions (hardware), mutual exclusion (operating systems), concurrency control (database systems), parallel prefix computation (algorithms).

4. CONCLUSIONS

We have outlined the current state of the art in concurrency research and proposed strategic directions and challenges for further investigation. We hope that we have convinced the reader that the field of concurrency is thriving. Mature theories and models exist; specification, design, and verification methods and tools are under active development; many successful applications of substantial complexity have been carried out and the potential for further and more sophisticated application is enormous.

The future of computing is interactive and networked. Consequently the role of concurrency theory, which aims at understanding the nature of interaction, will continue to grow.

An active mailing list for the field can be subscribed to by sending email to concurrency@cwi.nl.

ACKNOWLEDGMENTS

This report has benefited from helpful comments from Chris Hankin, Michael Loui, Jack Stankovic, and Peter Wegner.

REFERENCES

ABRAMSKY, S. 1996. Retracing some paths in process algebra. In *Proceedings of CONCUR '96—Seventh International Conference on Concurrency Theory, Lecture Notes in Computer Science,* Vol. 1119, (Pisa, Italy, Aug.) U. Montanari and V. Sassone, Eds., Springer-Verlag, Berlin, New York, 1–17.

ABRAMSKY, S., GAY, S., AND NAGARAJAN, R. 1996. Specification structures and propositions-as-types for concurrency. In *Logics for Concurrency: Structure vs. Automata,* F. Moller and G. Birtwistle, Eds. *Lecture Notes in Computer Science,* Vol. 1043, (Banff, Canada, Aug) Springer-Verlag. Berlin, New York,

ACETO, L., BLOOM, B., AND VAANDRAGER, F. W. 1994. Turning SOS rules into equations. *Inf. Comput. 111,* 1 (May), 1–52.

ALPERN, B. AND SCHNEIDER, F. B. 1985. Defining liveness. *Inf. Process. Lett. 21,* 181–185.

ALUR, R. AND DILL, D. 1994. The theory of timed automata. *TCS 126,* 2.

ALUR, R. AND HENZINGER, T. A. 1996a. *Computer Aided Verification (CAV '96), Lecture Notes in*

Computer Science, Vol. 1102, (New Brunswick, NJ, July) Springer-Verlag. Berlin, New York.

ALUR, R. AND HENZINGER, T. A. 1996b. Reactive modules. In *Proceedings of the Eleventh IEEE Symposium on Logic in Computer Science,* IEEE Computer Society, Washington, DC, 207–218.

ANDERSEN, H. R. 1994. Model checking and Boolean graphs. *Theor. Comput. Sci. 126,* 1.

BAETEN, J. C. M. AND WEIJLAND, W. P. 1990. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, New York.

BAETEN, J. C. M., BERGSTRA, J. A., AND KLOP, J. W. 1993. Decidability of bisimulation equivalence for processes generating context-free languages. *J. ACM 40,* 653–682.

BARR, M. 1979. **-Autonomous Categories, Lecture Notes in Mathematics,* Vol. 752, Springer-Verlag. Berlin, New York.

BERNHOLTZ, O., VARDI, M. Y., AND WOLPER, P. 1994. An automata-theoretic approach to branching-time model checking. In *Computer Aided Verification (CAV '94), Lecture Notes in Computer Science,* D. L. Dill, Ed., Vol. 818, (Stanford, CA, June) Springer-Verlag, Berlin, New York. 142–155.

BERRY, G. AND GONTHIER, G. 1992. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming 19*, 2 (Nov.), 87–152.

BEST, E., DEVILLERS, R., AND HALL, J. G. 1992. The Petri box calculus: A new causal algebra with multilabel communication. In *Advances in Petri Nets 1992,* G. Rozenberg, Ed., *Lecture Notes in Computer Science,* Vol. 609, Berlin, New York. Springer-Verlag, 21–69.

BEZEM, M. AND GROOTE, J. F. 1993. A formal verification of the alternating bit protocol in the calculus of constructions. Logic Group Preprint Series 88, Dept. of Philosophy, Utrecht University, March.

BHAT, G., CLEAVELAND, R., AND GRUMBERG, O. 1995. Efficient on-the-fly model checking for CTL*. In *Tenth Annual Symposium on Logic in Computer Science (LICS '95)* (San Diego, July), Computer Science Press, New York, 388–397.

BROOKES, S. D., HOARE, C. A. R., AND ROSCOE, A. W. 1984. A theory of communicating sequential processes. *J. ACM 31,* 3 (July), 560–599.

BROWNE, M. C., CLARKE, E. M., AND GRUMBERG, O. 1988. Characterizing finite Kripke structures in propositional temporal logic. *Theor. Comput. Sci. 59,* 1,2, 115–131.

BURCH, J. R., CLARKE, E. M., MCMILLAN, K. L., DILL, D. L., AND HWANG, L. J. 1992. Symbolic model checking 10^{20} states and beyond. *Inf. Comput. 98,* 2 (June), 142–170.

CARRIERO, N. AND GELERNTER, D. 1989. Linda in context. *Commun. ACM, 32,* 4 (April), 444–458.

CHANDY, K. M. AND MISRA, J. 1988. *Parallel Program Design. A Foundation.* Addison-Wesley, Reading, MA.

CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8,* 2, 244–263. *Lecture Notes in Computer Sci.* Vol. 697.

CLARKE, E. M., FILKORN, T., AND JHA, S. 1993. Exploiting symmetry in model checking. In *Computer Aided Verification* (Elounda, Greece, June), C. Courcoubetis, Ed., Springer-Verlag, Berlin, New York, 450–462.

CLARKE, E. M., GRUMBERG, O., HIRAISHI, H., JHA, S., LONG, D. E., MCMILLAN, K. L., AND NESS, L. A. 1995. Verification of the Future+ cache coherence protocol. *Formal Methods Syst. Des. 6,* 217–232.

CLEAVELAND, R. AND STEFFEN, B. U. 1993. A linear-time model checking algorithm for the alternation-free modal mu-calculus. *Formal Methods Syst. Des. 2,* 121–147.

CLEAVELAND, R., MADELAINE, E., AND SIMS, S. 1995. A front-end generator for verification tools. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '95),* (Aarhus, Denmark, May), E. Brinksma, R. Cleaveland, K. G. Larsen, and B. Steffen, Eds., *Lecture Notes in Computer Science,* Vol. 1019, Springer-Verlag, Berlin, New York, 153–173.

CLEAVELAND, R., PARROW, J., AND STEFFEN, B. U. 1993. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Trans. Program. Lang. Syst. 1,* 15, 36–72.

DAWS, C. AND YOVINE, S. 1995. Two examples of verification of mutirate automata using KRONOS. In *Sixteenth Annual IEEE Real-Time Systems Symposium* (Pisa, Italy, Dec.), Computer Society Press, 66–75.

DE BAKKER, J. W., HUIZING, C., DE ROEVER,W.-P., AND ROZENBERG, G. 1992. *Proceedings of REX Workshop on Real-Time: Theory in Practice.* Lecture notes in Computer Science, vol. 600, (Mook, the Netherlands, June 1991) Springer-Verlag, Berlin, New York.

DE NICOLA, R. AND HENNESSY, M. 1984. Testing equivalences for processes. *Theor. Comput. Sci. 34,* 83–133.

DIJKSTRA, E. W. 1968. Cooperating sequential processes. In *Programming Languages,* Academic Press, London, 43–112. Originally appeared as Tech. Rep. EWD-123, Technical University of Eindhoven, the Netherlands, 1965.

EMERSON, E. A. AND HALPERN, J. Y. 1986. 'Sometime' and 'not never' revisited: On branching versus linear time temporal logic. *J. ACM 33* 1, 151–178.

EMERSON, E. A. AND SISTLA, A. P. 1993. Symmetry and model checking. In *Computer-Aided Verification* (*CAV '93*), C. Courcoubetis, Ed., Springer-Verlag, Berlin, New York, 463–478.

FOCARDI, R. AND GORRIERI, R. 1995. A classification of security properties for process algebras. *J. Comput. Sec. 3,* 1, 5–33.

FRANCEZ, N. 1986. *Fairness.* Springer-Verlag, Berlin, New York.

GABRIEL, K. J. 1995. Engineering microscopic machines. *Sci. Am.* (Sept.).

GERBER, R. AND LEE, I. 1994. A resource-based prioritized bisimulation for real-time systems. *Inf. Comput. 113,* 1 (Aug.), 102–142.

VAN GLABBEEK, R. J. 1990. Comparative concurrency semantics and refinement of actions. Ph.D. Thesis, Free University of Amsterdam, 1990. Available through URL http://theory.stanford.edu/~rvg/thesis.html.

VAN GLABBEEK, R. J. 1993. Full abstraction in structural operational semantics. In *Proceedings of the Third AMAST Conference,* (Twente, The Netherlands, June), M. Nivat, C. Rattray, T. Rus, and G. Scollo, Eds. Workshops in Computing, Springer-Verlag, Berlin, New York, 77–84.

VAN GLABBEEK, R. J. AND PLOTKIN, G. 1995. Configuration structures. In *Logic in Computer Science,* IEEE Computer Society, Washington, DC, 199–209.

VAN GLABBEEK, R. J., SMOLKA, S. A., AND STEFFEN, B. 1995. Reactive, generative, and stratified models of probabilistic processes. *Inf. Comput. 121,* 1 (Aug.), 59–80.

GROOTE, J. F. AND VAANDRAGER, F. W. 1992. Structured operational semantics and bisimulation as a congruence. *Inf. Comput. 100,* 2 (Oct.), 202–260.

GROSSMAN, R. L., NERODE, A., RAVN, A. P., AND RISCHEL, H., Eds. 1993. *Hybrid Systems, vol. 736,* (Lyngby, Denmark, Oct. 1992) Lecture Notes in Computer Science, Springer-Verlag. Berlin, New York.

GUPTA, V. AND PRATT, V. R. 1993. Gates accept concurrent behavior. In *Proceedings of the Thirty-Fourth Annual IEEE Symposium on Foundations of Computer Science,* (Nov.), 62–71.

HALBWACHS, N. 1993. *Synchronous Programming of Reactive Systems.* Kluwer Academic, Boston, MA.

HALPERN, J. AND MOSES, Y. 1990. Knowledge and common knowledge in a distributed environment. *J. ACM 37,* 3 (July), 549–587.

HALPERN, J. AND ZUCK, L. 1992. A little knowledge goes a long way: Knowledge-based derivations and correctness proofs for a family of protocols. *J. ACM 39,* 3 (July), 449–478.

HANSSON, H. A. 1994. *Time and Probability in Formal Design of Distributed Systems, Real-Time Safety Critical Systems,* Vol. 1, Elsevier, Amsterdam.

HAREL, D. 1987. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program. 8,* 231–274.

HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., SHTULL-TRAURING, A., AND TRAKTENBROT, M. 1990. Statemate: A working environment for the development of complex reactive systems. *IEEE Trans. Softw. Eng. 16,* 4 (April), 403–414.

HEIMDAHL, M. AND LEVESON, N. 1996. Completeness and consistency in hierarchical statebased requirements. *IEEE Trans. Softw. Eng. SE-22,* 6, 363–377.

HENNESSY, M. 1988. *Algebraic Theory of Processes.* MIT Press, Cambridge, MA.

HENNESSY, M. AND MILNER, R. 1985. Algebraic laws for nondeterminism and concurrency. *J. ACM 32,* 1, 137–161.

HENZINGER, T. A., HO, P.-H., AND WONG-TOI, H. 1995. HyTech: The next generation. In *Sixteenth Annual IEEE Real-Time Systems Symposium,* (Pisa, Italy, Dec.), Computer Society Press, 56–65.

HOARE, C. A. R. 1985. *Communicating Sequential Processes.* Prentice-Hall, London.

HOLZMANN, G., PELED, D., AND PRATT, V. R. 1996. *Partial-Order Methods in Verification (POMIV '96).* DIMACS Series in Discrete Mathematics and Computer Science. American Mathematical Society, Providence, RI.

INMOS INTERNATIONAL 1988. OCCAM-*2 Reference Manual,* Prentice-Hall International, Englewood Cliffs, NJ.

JAGANNATHAN, S. AND WEEKS, S. 1994. Analyzing stores and references in a parallel symbolic language. In *ACM Conference on LISP and Functional Programming,* 294–305.

JONES, C. B. 1992. The search for tractable ways of reasoning about programs. Tech. Rep. TR UMCS-92-4-4, Department of Computer Science, University of Manchester, 1992. Available through URL http://www.cs-.man.ac.uk/csonly/cstechrep/Abstracts/UMCS-92-4-4.html.

KAHN, G. 1974. The semantics of a simple language for parallel programming. In *Information Processing 74*, J. L. Rosenfeld, Ed., North-Holland, Amsterdam.

KAHN, G. AND MACQUEEN, D. B. 1977. Coroutines and networks of parallel processes. In *Information Processing 77*, North-Holland, Amsterdam, 993–998.

KALTENBACH, M. 1996. Interactive verification exploiting program design knowledge: A model checker for UNITY. PhD thesis, University of Texas, Austin. Available through

URL http://www.cs.utexas.edu/users/markus/diss.html.

Kanellakis, P. C. and Shvartsman, A. A. 1992. Efficient parallel algorithms can be made robust. *Distrib. Comput. 5,* 4, 201–217.

Kanellakis, P. C. and Smolka, S. A. 1990. CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput. 86,* 1 (May), 43–68.

Keller, R. 1976. Formal verification of parallel programs. *Commun. ACM 19,* 7 (July), 371–384.

Kozen, D. 1983. Results on the propositional mu-calculus. *Theor. Comput. Sci. 27,* 333–354.

Lamport, L. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng. SE-3,* 2, 125–143.

Lamport, L. 1980. The "Hoare logic" of concurrent programs. *Acta Inf. 14,* 21–37.

Larsen, K. G. and Skou, A. 1992. Bisimulation through probabilistic testing. *Inf. Comput. 94,* 1, (Sept.), 1–28.

Lin, H. 1991. PAM: A process algebra manipulator. In *Computer Aided Verification (CAV '91),* (Aalborg, Denmark, July), *Lecture Notes in Computer Science,* K. G. Larsen and A. Skou, Eds., Vol. 575, Springer-Verlag, Berlin, New York, 136–146.

Lynch, N. A. and Tuttle, M. R. 1989. An introduction to input/output automata. *CWI Quarterly 2,* 3 (Sept.), 219–246.

Manna, Z. and Pnueli, A. 1991. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Berlin, New York, Springer-Verlag.

Manna, Z. and Pnueli, A. 1995. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, Berlin, New York.

Margaria, T. and Steffen, B. 1996. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)* (Passau, Germany, March), Lecture Notes in Computer Science, Vol. 1055, Springer-Verlag, Berlin, New York.

Mazurkiewicz, A. 1987. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course* (Bad Honnef, Sept.), W. Brauer, W. Reisig, and G. Rozenberg, Eds., Lecture Notes in Computer Science, Vol. 255, Springer-Verlag, Berlin, New York, 279–324.

Mifsud, A., Milner, R., and Power, J. 1995. Control structures. In *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science,* (San Diego, CA, June) IEEE Computer Society Press, Los Alamitos, CA, 188–198.

Milner, R. 1980. A calculus of communicating systems. *Lecture Notes in Computer Science,* Vol. 92, Springer-Verlag, Berlin, New York.

Milner, R. 1989. *Communication and Concurrency.* International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ.

Milner, R. 1993. Action calculi, or syntactic action structures. In *Proceedings of the Nineteenth MFCS,* Lecture Notes in Computer Science, Vol. 711, Springer-Verlag, Berlin, New York, 105–121.

Milner, R., Tofte, M., and Harper, R. 1990. *The Definition of Standard ML.* MIT Press, Cambridge, MA.

Montanari, U. and Rossi, F. 1996. Graph rewriting and constraint solving for modelling distributed systems with synchronization. In *Proceedings of COORDINATION '96,* (Cesana, Italy, April) Lecture Notes in Computer Science, Vol 1061. P. Ciancarini and C. Hankin, Eds. Springer-Verlag, Berlin, New York.

Nielson, F. and Nielson, H. R. 1993. From CML to process algebras. In *Proceedings of CONCUR '93—Fourth International Conference on Concurrency Theory. Lecture Notes in Computer Science,* Vol. 715, Springer-Verlag, Berlin, New York.

Nielson, H. R. and Nielson, F. 1994. Higher-order concurrent programs with finite communication topology. In *Proceedings of the Twenty-First Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* (Portland, OR, Jan.) ACM Press, New York, 84–97.

Nierstrasz, O. 1990. Viewing objects as patterns of communicating agents. In *Proceedings of ECOOP/OOPSLA '90; European Conference on Object-Oriented Programming/Object-Oriented Programming Systems, Languages and Applications, SIGPLAN Not. 25*, 10, 38–43.

Olderog, E.-R. 1991. *Nets, Terms and Formulas: Three Views of Concurrent Processes and their Relationship.* Cambridge Tracts in Theoretical Computer Science 23. Cambridge University Press, New York.

Owicki, S. and Gries, D. 1976. An axiomatic proof technique for parallel programs. *Acta Inf. 6,* 319–340.

Owre, S., Rushby, J., and Shankar, N. 1992. PVS: A prototype verification system. In *Eleventh International Conference on Automated Deduction (CADE)* (Saratoga Springs, NY, June), D. Kapur, Ed., Lecture Notes in Artificial Intelligence, Vol. 607, Springer-Verlag, Berlin, New York, 748–752.

Paige, R. and Tarjan, R. E. 1987. Three partition refinement algorithms. *SIAM J. Comput. 16* 6 (Dec.), 973–989.

Peleg, D. 1987. Concurrent dynamic logic. *J. ACM 34,* 2 (April), 450–479.

Petri, C. A. 1962. Kommunikation mit Automaten. Schriften des IIm 2, Institut für Instrumentelle Mathematik, Bonn, 1962. English translation available as *Communication* *with Automata,* Tech. Rep. RADC-TR-65-377, Vol. 1, Suppl. 1, Applied Data Research, Princeton, NJ, 1966.

PIERCE, B. C. AND TURNER, D. N. 1995. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming,* (Sendai, Japan, April), Lecture Notes in Computer Science, Vol. 907, Springer-Verlag, Berlin, New York.

PLOTKIN, G. D. 1981. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University.

PNUELI, A. 1977. The temporal logic of programs. In *Eighteenth IEEE Symposium on Foundations of Computer Science,* Computer Society Press, Los Alamitos, CA, 46–57.

PNUELI, A. AND SIFAKIS, J. 1995. *Special Issue on Hybrid Systems of Theoretical Computer Science 138*, 1 (Feb), Elsevier Science Publishers.

PRATT, V. R. 1986. Modeling concurrency with partial orders. *Int. J. Parallel Program. 15,* 1, 33–71.

PRATT, V. R. 1995. The Stone gamut: A coordinatization of mathematics. In *Logic in Computer Science,* IEEE Computer Society, Los Alamitos, CA, 444–454.

QUEILLE, J. P. AND SIFAKIS, J. 1982. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming* (Berlin), Lecture Notes in Computer Science, Vol. 137, Springer-Verlag, Berlin, New York.

REISIG, W. 1985. *Petri Nets—An Introduction.* EATCS Monographs on Theoretical Computer Science, Vol. 4. Springer-Verlag, Berlin, New York.

REPPY, J. H. 1992. High-order concurrency. Ph.D. Thesis, Cornell University, 1992. Also Cornell University Computer Science Dept. Tech. Report 92–1285.

ROSCOE, A. W. 1994. Model checking CSP. In *A Classical Mind: Essays in Honor of C. A. R. Hoare.* Prentice-Hall International, Englewood Cliffs, NJ.

ROSCOE, A. W. 1995. CSP and determinism in security modelling. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy,* IEEE Computer Society Press, Los Alamitos, CA, 114–127.

ROSENBERG, A. L. 1995. Thoughts on parallelism and concurrency in computing curricula. *ACM Comput. Surv. 27,* 2 (June), 280–283.

ROY, V. AND DE SIMONE, R. 1990. Auto/Autograph. In *Computer Aided Verification (CAV '90)* (Piscataway, NJ, June), E. M. Clarke and R. P. Kurshan, Eds. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science,* Vol. 3, American Mathematical Society, Providence, RI, 235–250.

SANGIORGI, D. 1992. Expressing mobility in process algebras: First-order and higher-order paradigms. Ph.D. Thesis, University of Edinburgh, Edinburgh, Scotland.

SASTRY, S., MEYER, G., TOMLIN, C., LYGEROS, J., GODBOLE, D., AND PAPPAS, G. 1995. Hybrid control in air traffic management systems. In *Proceedings of the Thirty-Fourth IEEE Conference on Decision and Control,* 1478–1483.

THIAGARAJAN, P. S. 1994. A trace based extension of linear time temporal logic. In *Ninth Annual Symposium on Logic in Computer Science (LICS '94)* (Versailles, France, July), Computer Society Press, Washington, D. C., 438–447.

THOMSEN, B. 1995. A theory of higher order communicating systems. *Inf. Comput. 116,* 1 (Jan.), 38–57.

THOMSEN, B., LETH, L., AND KUO, T.-M. 1996. A Facile tutorial. In *Proceedings of CONCUR '96—Seventh International Conference on Concurrency Theory,* (Pisa, Italy), Lecture Notes in Computer Science, Vol. 1119, Springer-Verlag, Berlin, New York, 278–298.

VARDI, M. AND WOLPER, P. 1986. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS '86)* (Cambridge, MA, June), Computer Society Press, 332–344.

WINSKEL, G. 1987. Petri nets, algebras, morphisms, and compositionality. *Inf. Comput. 72,* 197–238.

WINSKEL, G. 1989. An introduction to event structures. In *REX School and Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency* (Noordwijkerhout, The Netherlands, May/June), Lecture Notes in Computer Science, Vol. 354, J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds., Springer-Verlag, Berlin, New York, 364–397.

WINSKEL, G. AND NIELSEN, M. 1995. Models for concurrency. In *Handbook of Logic in Computer Science,* Vol. 4, S. Abramsky, D. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, New York, 1–148.