Towards Verified Distributed Software Through Refinement of Formal Archetypes

K. Mani Chandy[1], Brian Go[1], Sayan Mitra[2], Jerome White[1]

[1] California Institute of Technology
(mani,bgo,jerome)@cs.caltech.edu
[2] University of Illinois at Urbana Champaign
(mitras)@crhc.uiuc.edu

Abstract. This paper discusses experiments with a "model-based" approach for developing verified distributed systems in which program development is carried out by stepwise refinement: we encode, specifications and algorithm archetypes in the PVS theorem prover, carry out stepwise refinement and concomitant proofs, and obtain collections of verified algorithms encoded in PVS. Finally we transform algorithms from PVS to programs in Java. We consider a class of systems in which state spaces may be continuous and state transitions may be continuous or discrete. Coordinated multi-vehicle systems are examples of this class. Temporal properties of this class of problems are specified in terms of convergence: the system state gets arbitrarily close to a limit as time tends to infinity. Our meta-theorems for verifying convergence are extensions from control theory to a temporal logic of continuous time and state spaces.

1 Introduction

This paper discusses experiments with a "model-based" approach for developing verified distributed systems in which program development is carried out by stepwise refinement: we encode theorems, specifications and algorithm archetypes as meta-theories in the PVS theorem prover [19], carry out stepwise refinement and concomitant proofs using the theorem prover, and obtain collections of verified algorithms encoded in PVS. Lastly, we transform algorithms from PVS to programs in Java.

Verifying the correctness of distributed algorithms is not easy because of race conditions and nondeterminism. It is particularly challenging for systems in which messages may be lost or delayed, as well as systems with real-time constraints. Distributed systems with continuous state spaces are often even harder to verify because state spaces are uncountable. Moreover, programs that mechanically transform high-level specifications to actual executable code (see, for example, [21]) do not always generate programs with satisfactory performance.

In this paper we describe an ongoing project which attempts to overcome these difficulties for a class of distributed systems in which state spaces may be continuous and state transitions may be continuous or discrete. Distributed control systems such as multi-vehicle systems and multi-robot systems are examples of this class. A key temporal properties of this class of problems is convergence: the system state gets arbitrarily close to a desired value as time tends to infinity. The corresponding property of distributed algorithms with discrete state spaces is termination: a desired state is reached in a finite number of steps. Our meta-theorems for verifying convergence are extensions of theorems from control theory to a temporal logic of continuous time and continuous state spaces [18]. We use the same theory for developing discrete-state and continuous-state algorithms.

In stepwise verification [2, 4, 9] programmers develop collections of meta-theorems and verified algorithm *archetypes*, and refine the algorithm archetypes to obtain specific algorithms. Our contribution is to provide an example of this approach using a theorem prover, PVS, for verification of a class of distributed algorithms including algorithms for some control-theoretic problems. Archetypes of distributed systems algorithms are represented as timed or untimed I/O automata [14] and encoded in PVS using the translation scheme presented in [17, 16].

Although in our experiments the programs that implement each agent algorithm are very small, mechanical verification took considerable effort. The challenge was not in verifying the correctness of the few lines of code that implement a single agent, but in proving properties of compositions of agents. In our experiments the transformation from PVS to Java notation could have been done mechanically and efficiently, because the transformations are as straightforward as showing that simple functions, such as *max*, implemented in Java are equivalent to the functions defined in PVS. In this set of experiments, however, we carried out the transformations by hand.

A secondary goal of our project is to develop pedagogical devices for distributed computing and distributed control systems. We expect that material structured around a small collection of theorems, algorithm archetypes, and formal proofs will facilitate student understanding of distributed systems. Our work suggests that a small formal foundation can be used to develop programs for the problems discussed in most courses in this area.

The paper is organized as follows: we begin by introducing the problem domain. The stages of distributed system design using our verifica-

tion approach are presented. We demonstrate applications to convergent, game-theoretic, and multi-agent distributed system abstractions. We conclude with a reflective discussion.

2 Preliminaries

2.1 Distributed System Model

We describe a distributed system with N agents where the state of each agent is a value of some type, $\mathcal{S}$. The state of the system is an array S of type $\mathcal{S}$ indexed by $\{0, \ldots, N-1\}$. The jth element in the array, the state of agent j, is denoted by $S[j]$. We denote the state of the system at time t, for $t \geq 0$, by $S^{(t)}$. In a continuous state transition system, t is the set of real numbers, while in a discrete state transition system t is the set of natural numbers.

We are interested in studying termination or convergence, of protocols with respect to the values of certain state variables. Let $\mathcal{T}$ be the restriction of the type $\mathcal{S}$ to the valuations of those variables that we care about in the particular application. For agent j, we denote the restriction of its state $S[j]$ to these variables of interest by $X[j]$. In a mobile agent system, for example, $X[j]$ may represent the location of agent j while $S[j]$ includes other state variables such as agent j's velocity and its memory of past actions. We denote the value of $X[j]$ at time t by $X^{(t)}[j]$. This is the restriction of agent j's state $S^{(t)}[j]$ to the variables of interest. As it is clear from the context which set of variable values we mean, with some abuse of terminology, we refer to both $S[j]$ and $X[j]$ as the state of agent j.

The class of problems we consider are those in which the system terminates in or converges to, a state which is a function of the initial state, $X^{(0)}$. Let g be a function that maps arrays of type $\mathcal{T}$ to arrays of the same type, $\mathcal{T}$. The state to which the system must converge is $X^{(*)}$ where

$$X^{(*)} = g(X^{(0)})$$

In a mobile agent problem $X^{(0)}$ could be the array of an initial agent locations and $X^{(*)}$ the desired final state. In this case, for instance, $g(X^{(0)})$ may be the configuration in which all agents form an equispaced straight line connecting the initial locations of agents 0 and $N-1$. This class of problems includes problems such as computing the shortest paths between all pairs of agents, but does not include problems such as mutual exclusion and global snapshot detection.

2.2 Representation of Distributed Systems in PVS

We encode the above model of distributed systems as timed or untimed automata in PVS. This encoding is standard and is based on the theories developed in previous work [3, 16]—the exception being our encoding of fairness criterion and fair-executions of automata, which is based on our recent work [18]. Here, we review the fairness condition briefly. For the simplicity of exposition we present the definitions in terms of basic set theory instead of the typed higher order logic of PVS.

An automaton is defined by (1) a set of states, (2) a nonempty subset of states called the initial states, (3) a nonempty, possibly uncountable, set of actions, and (4) a fairness criterion. Associated with each action a is a function f_a from states to states. Execution of action a when the system is in a state S takes the system to a state $f_a(S)$. It is possible that execution of an action a in a state S leaves the state unchanged, that is, $f_a(S) = S$. An execution is an initial state $S^{(0)}$ followed by a finite or infinite sequence of action-state pairs: $(a_n, S^{(n)})$ where:

$$f_{a_n}(S^{(n-1)}) = S^{(n)}$$

Although the set of actions may be uncountable, it is represented in PVS as a finite set of functions that maps the current state, and a set of auxiliary parameter values, to a new state. For example, the uncountable set of actions that moves an agent along a line by an amount Δx, for all real values of Δx, is represented by a function *move* that maps the current location x of an agent to a new location defined by:

$$move(x, \Delta x) = x + \Delta x$$

Specifications of Fairness. We denote a fairness condition $\mathcal{F}$ by a finite collection $\{F_i\}_{i=1}^{n}, n \in \mathbb{N}$, where each F_i is a set of transitions. An infinte execution α is said to be $\mathcal{F}$-fair if every $F_i \in \mathcal{F}$ is represented (by *some* action $a \in F_i$) infinitely often in α.

Consider, for example, the specification of the fairness requirement that a system (whose transitions are agent interactions) is never permanently partitioned into non-interacting (nonempty) subsets of agents. For every partition of the agents into two nonempty subsets we define a set F_j of actions, $F_j \in \mathcal{F}$, where the actions in F_j operate on the states of agents in both subsets. Since F_j is represented infinitely often, at least one action that involves agents in both subsets is executed infinitely often.

2.3 Theorems for Proving Convergence

In our recent work [18] we have developed a PVS theory for proving convergence and termination of a general class of automata. Here we give two examples of theorems which are preliminary to proving convergence properties.

A concept used in temporal logic is that of stable predicates. Stability of the predicate P over the set of states is the implication that if P holds in any state S, then it holds in all states reachable from S. We use a PVS theorem that the stability of P, denoted by stable.P can be deduced from:

$$(\forall S, S' : \ \mathit{transition}(S, S') \wedge P(S) \ \Rightarrow \ P(S'))$$

where the relation *transition* specifies the set of all possible transitions (pre-state/post-state pairs) in the automaton.

A concept used to prove progress properties is that "P leads to Q," denoted by $P \leadsto Q$, where P and Q are predicates on states. "P leads to Q" means that if P holds at any point in any execution, then Q holds at that point or at a later point in the execution. We can use the following theorem from [18]: $P \leadsto Q$ is implied by

$$(\text{stable}.P \wedge \exists F \in \mathcal{F} : \forall a \in F, S : P(S) \Rightarrow Q(f_a(S)))$$

Intuitively, in each infinite execution, F is represented by some action infinitely often. If all actions in F take the system from a state S in which P holds to a state $f_a(S)$ in which Q holds, then if P holds at some point in an execution, Q must hold at some later point in that execution.

For this work we build up on the existing theory to obtain sufficient conditions for proving termination and convergence of certain special classes of distributed systems. Next we describe the relationship between these classes of distributed systems with our formal archetypes.

2.4 Stages of Design

The central idea in organizing designs of collections of programs is repeated use of theorems. Thus, programs that have common specification and proof structures give rise to algorithm archetypes. We illustrate this idea with program transformations and algorithm refinement (see Figure 1).

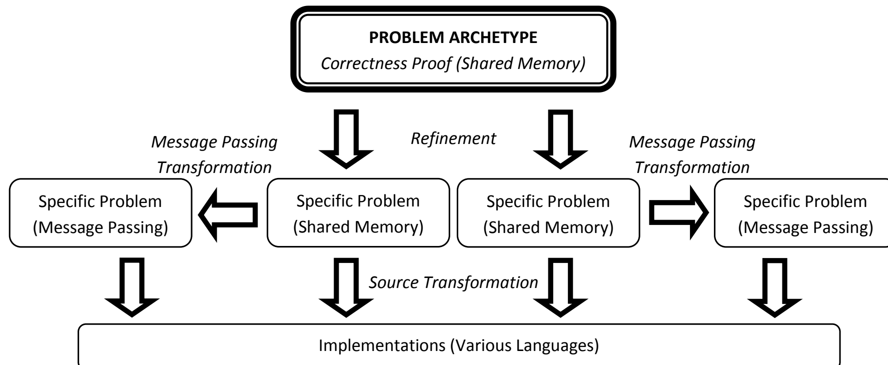

Fig. 1. The process of taking an abstract problem archetype to an actual implementation. We first prove properties about abstract archetypes using a mechanical theorem prover. We then refine the abstraction for specific problem domains. Finally, the refined pieces are implemented into correct executable code.

Program Transformation. A theorem that we found to be particularly useful in these experiments is that certain classes of verified programs for shared-memory systems can be mechanically transformed into correct programs for message-passing systems even if messages may be lost or delayed [10]. The transformation is straightforward: an action in a shared-memory algorithm, in which an agent j reads the state of an agent k, is transformed into an action in a message-passing algorithm, in which j reads the value of the state in the last message that j received from k. The state in a message from k is the state of k when the message was sent which may not be k's current state. Nevertheless, the theorem says that the message-passing algorithm converges provided sub-level sets of the Lyapunov functions have the proper structure. Thus, for some problems, after we mechanically verify a shared-memory algorithm, we also get a verified message-passing implementation for free.

Refinement Steps. An example of refinement, that we actually use in Section 4 to derive algorithms for mobile agent systems, is as follows. We start with theorems for a class of games called dynamic games in which agents change states in continuous or discrete steps to maximize their rewards. Next, we prove additional theorems for the subclass of linear dynamic games in which the best-response functions of agents are linear. Then we use the above program transformations to obtain results for linear dynamic games in which agents communicate by message passing. At

the next refinement step we show that a specific mobile agent system is an instance of a linear dynamic game. The final step would be transforming agent algorithms from PVS to mobile robots in Player/Stage [22].

3 Example 1: Consensus Problems

Our first example deals with consensus problems. Here all agents are required to have the same X-value in the final state:

$$\forall i, j : X^{(*)}[i] = X^{(*)}[j].$$

Let $f(X^{(0)})$ be the desired final X-value of each agent, then

$$\forall j : X^{(*)}[j] = f(X^{(0)}).$$

3.1 Archetype for Associative, Commutative, and Idempotent Operators

We develop theorems and algorithms for different properties of f. We consider here functions f that have the following property: there exists an associative, commutative, idempotent operator $\circ$ such that:

$$f(X) = \begin{cases} X[0] & \text{if } N = 1, \\ f(X[0, \ldots, N-2]) \circ X[N-1] & \text{otherwise.} \end{cases}$$

For example, if $f(X)$ is the maximum over all j of $X[j]$, then $u \circ v$ could be defined as the maximum of u and v. Other examples include minimum, greatest common divisor and least common multiple.

3.2 Archetype for Shared Memory and Message Passing Systems

We first develop an archetype for all consensus algorithms in which f has the desired properties. In doing so, we make the system assumption that each agent can read the states of other agents but cannot modify them, and each agent can both read and modify its own state.

Consider a system in which the state $S[j]$ of agent j equals $X[j]$, and where an action is defined by an ordered pair (j, k) where j and k are agent indices. The action (j, k) represents agent j reading the value of agent k and setting its own state to $X[j] \circ X[k]$ while other agents take

no action. The action (j, k) takes the system from a state X to a state X' where:

$$X'[i] = \begin{cases} X[i] & \text{if } i \neq j, \\ X[j] \circ X[k] & \text{otherwise.} \end{cases}$$

The fairness requirement is weak: it states that for all nonempty subsets of agents A, $\bar{A}$, where $\bar{A}$ is the complement of A, there exists $j \in A$ and $k \in \bar{A}$, such that action (j, k) is executed infinitely often. We prove and mechanically verify (see [1]) the following theorem:

Theorem 1 *A system with the above actions and fairness requirements reaches the state $X^{(*)}$ in a finite number of actions.*

The proof of the algorithm archetype for shared-memory systems has the properties required to allow us to use the previously mentioned transformation theorem to show that the following message-passing version is also correct.

1. Each agent j sends $X[j]$ infinitely often to the message-communication layer.
2. When an agent j receives a message m, it sets its state to $X'[j]$ where:

$$X'[j] = X[j] \circ m$$

3.3 Instantiations of the Archetype: Consensus Protocols

Next, we develop algorithms, encoded in PVS, for different instantiations of the $\circ$ operator, such as *max*, *gcd* and convex hull. Because we have already verified that the transitions are safe and progress, with respect to consensus, our only remaining responsibility is to show that the instantiated operator respects our assumptions. That is, to show that a given function is associative, commutative, and idempotent. In the case of *min* and *max*, the PVS proofs are trivial. While the proofs are more complex for other operators, the effort required is much less than that required to prove safety and progress of the algorithms they instantiate.

3.4 Programs Implementing the PVS Algorithms

Finally, we develop Java implementations for each of the PVS algorithms. Our implementations consist of three base classes: one representing operators, another for agents, and a third for communication. The operator class is an abstract one, refined to obtain specific functions of interest,

such as *max* and *gcd*. Our communication layer is a harness to compose agents. It is similar to, and could be replaced by, more standardized middlewares, such as Java Message Service (JMS).

Based on our PVS model, our Java implementation has three requirements:

1. Each agent sends its state in a message infinitely often. In Java this can be done by each agent "sleeping" for some bounded time, and then sending its state when it wakes up.
2. Send and receive operations must be implemented appropriately for the middleware (using JMS, for example).
3. The operation of updating the agent's state (in our case this takes the form $X'[j] = X[j] \circ X[k]$) must be implemented.

As mentioned earlier, the systems that we are verifying and later implementing were fairly simple. Thus, going from PVS to Java is straightforward and our requirements could, for the most part, be verified by hand. Moreover, because most of the requirements are middleware specific, we assume that they have been independently demonstrated to be correct. Thus, our primary concern in implementation was that the mechanism for updating agent states satisfied the PVS requirements.

4 Example 2: Game Theoretic Algorithms

In this section we study a second class of algorithm archetypes that ultimately yield distributed algorithms for mobile-agent systems and scientific computation. We actually developed the archetypes from the bottom-up by starting with a specific problem in mobile-agent systems. Later on, we discovered the abstractions and theorems that helped us solve other problems. In the interest of a uniform narrative, however, we present this section top-down: how programs could be developed by applying sequences of refinement steps to the common archetype.

4.1 Archetype for Dynamic Games

We begin, as we did in Section 3, by considering shared-state systems in which each agent can read, but not modify, the states of other agents [8]. Let $S[j]$ be the state of agent j. We use the game-theoretic notation, $S[-j]$ to denote the array of states of all agents other than j. For example, for a 3-agent system, $S[-1] = [S[0], S[2]]$. Associated with each agent j is its *best response* function, β_j, which is a function from $S[-j]$ to $S[j]$. Each

agent has an objective function that the agent attempts to maximize, and the best response is the state of an agent j that maximizes its objective given that the states of the other agents remain at $S[-j]$. For example, in a mobile agent system where the state of each agent is its location in space, and the goal of agent j is to move to the midpoint of agents $[j-1]$ and $[j+1]$: β_j could be defined to be $(S[j-1]+S[j+1])/2$.

The state transitions of the system are as follows. At most one agent changes its state in a state transition. Consider a transition in which the states of all agents other than j remain unchanged; agent j may set its state to any convex combination of its current value $S[j]$ and its best response $\beta_j(S[-j])$, i.e., agent j picks any real number p in the interval $[0, 1]$, and sets its state to $p.\beta_j(S[-j]) + (1-p).S[j]$. The model approximates systems in which multiple agents move concurrently by interleaving arbitrarily small steps (small values of p) of different agents. Since a state transition could have $p = 0$, which results in no change in state, we impose the fairness property that each agent executes steps infinitely often with $p \geq \epsilon$ for some positive constant ϵ.

We prove theorems about dynamic games for arbitrary best-response functions, and later use the theorems in refinement steps. One theorem deals with Nash equilibria of these games. Consider the surface defined by the best response $S[j] = \beta_j(S[-j])$ of agent j for all values of $S[-j]$. It is easy to see that if there exists a point that lies on the intersection of these surfaces for all j, then that point is a Nash equilibrium and a fixed point of the system dynamics.

Though there exists equilibria, the system may not converge to an equilibrium point from an initial state. Next we consider a subclass of dynamic games in which existence of a Nash equilibrium guarantees its feasibility. We consider games in which the best response functions are linear and the state of each agent is a real number. In this case, the equilibrium conditions for each agent j

$$S[j] = \beta_j(S[-j])$$

is a linear equation, and the system of equations for all agents j is of the form:

$$A.S = b$$

where A is an $N \times N$ matrix, and b is an N-vector. Further we restrict attention to matrices A in which the diagonal elements are nonzero and normalized to unity. Then, the best response of an agent j in system state S is: $b_j - \sum_{k \neq j} A[j,k].S[k]$. There is a unique equilibrium if, and only if, A

is invertible. The system converges to the equilibrium $A^{-1}b$ if A is weakly diagonally dominant:

$$\forall j : |A[j,j]| \leq \sum_k |A[j,k]|$$
$$\exists j : |A[j,j]| < \sum_k |A[j,k]|$$

and the Laplacian-generation graph corresponding to A is strongly connected. Furthermore, the proof of convergence has the properties that allow us to deduce that the mechanical transformation of this shared-state algorithm to a message-passing algorithm also converges [10].

4.2 Archetype for Message-Passing Linear Dynamic Games

In the message passing version, each agent j sends a message containing its state $S[j]$ to the communication layer repeatedly. Each agent j maintains a vector R_j of the most recent messages it has received from other agents. Initially the elements of this vector are set to $\perp$ identifying that the value is undefined. When an agent j receives a message m from an agent k it sets $R_j[k]$ to m. Agent j computes its best response as $\beta_j(R_j[-j])$, rather than $\beta_j(S[-j])$. (The best response for agent j is undefined while R_j is undefined.) In the message passing algorithm, an agent j computes its best response, based *not on the true states of other agents*, but on the states of other agents in messages that agent j received.

4.3 Instantiation of Archetype: Pattern Formations Protocol

We now consider the following problem: given N mobile agents, where agents 0 and $N-1$ are stationary in some Cartesian space, the behavior of all other agents as follows. Agent j has local variables $C_j[j-1]$ and $C_j[j+1]$, which contain the last messages it received from agents $j-1$ and $j+1$, respectively. If both values are defined, agent j moves, in a straight line, towards the midpoint of $C_j[j-1]$ and $C_j[j+1]$ using arbitrary dynamics (e.g., velocity, acceleration). When messages arrive and the midpoint changes, the agent changes course, moving in a straight line from its current position to the new midpoint. Messages may get lost and messages may be delayed for arbitrary finite time, but messages between any pair of neighboring agents is delivered infinitely often.

Our initial concern was whether or not agents converge to a state in which they are all in an equally-spaced straight line. We proved that the answer to this question is "yes," and we verified this theorem in

PVS [10]. A very simple refinement step, however, uses the theorems of the previous problem class by restricting matrix A as follows. The only non-zero non-diagonal elements of A are $A[j-1,j] = 0.5$ for $j < N$ and $A[j,j+1] = 0.5$ for $j \geq 0$. Verifying that this matrix is weakly diagonally dominant is straightforward.

4.4 Programs Implementing PVS Algorithms

The next stage in refinement is to implement the PVS algorithm first on a mobile robot simulator, Player/Stage, and then on ER-1 robots. We have simulations and are beginning the robot implementation.

5 Reflective Discussion

The experiments we are continuing to carry out at Caltech and Illinois (UIUC) help in evaluating the costs and benefits of formal methods and theorem proving tools. A potentially fertile area of application for approaches such as this is distributed control systems. In this domain, agents communicate by messages that can be lost or delayed. The area is appropriate because: (1) some algorithms in this domain don't appear to be either obviously correct or obviously incorrect, (2) this area uses results from domains such as control, game and probability theory that are not studied extensively by the theorem-proving community, and (3) the domain appears to have theorems that are relevant for a variety of applications; especially as inexpensive sensor actuators and analytic engines are integrated into message-passing sense and respond systems.

It is well-known that mechanically verifying large systems is laborious and requires expertise—our experience was no exception. In our case, we also had to ascend a learning curve in using PVS. The benefits of the effort, however, come from reuse. Although our algorithms are small, we are able to mechanically verify key distributed system building blocks that larger systems could take advantage of.

There are different types of errors in programs including typing mistakes, errors in programs specifications, and errors in design; our experiments restricted attention to design errors. The specifications of the algorithms we studied were very small, so our experiments are not relevant for understanding specification error. Our theorems and algorithms were encoded in PVS—we did not prove the correctness of their transformations to Java.

For situations where the PVS specification and the corresponding Java implementations are much more complicated, we hope to use existing automated program verification techniques. Tools such as extended static checkers, JML [7], model checkers, randomized testing [11], formal methods [2] and high-level languages would prove beneficial. Work in program and data structure checking [20, 12, 5, 6, 15] would complement our work as well. We anticipate such a collective effort to meet the software verification grand challenge [13], an effort that at some level will require archetype refinement and verification.

References

1. Infospheres project. `http://www.infospheres.caltech.edu/vstte08`, October 2008.
2. J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.
3. Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000.
4. R. J. R. Back and K. Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3):324–346, May 1996.
5. Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM.
6. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, 2007.
7. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools Technology Transfer*, 7(3):212–232, 2005.
8. K. M. Chandy. Reasoning about continuous systems. *Science of Computer Programming*, 14(2-3):117–132, 1990.
9. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.
10. K. Mani Chandy, Sayan Mitra, and Concetta Pilotto. Convergence verification: From shared memory to partially synchronous systems. In *In proceedings of Formal Modeling and Analysis of Timed Systems (FORMATS '08)*, volume 5215 of *LNCS*, pages 217–231. Springer Verlag, 2008.
11. Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 621–631, Washington, DC, USA, 2007. IEEE Computer Society.
12. Kobi Inkumsah and Tao Xie. Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 425–428, New York, NY, USA, 2007. ACM.

13. C. Jones, P. O'Hearn, and J. Woodcock. Verified software: a grand challenge. *Computer*, 39(4):93–95, April 2006.
14. Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata (Synthesis Lectures in Computer Science)*. Morgan & Claypool Publishers, 2006.
15. Viktor Kuncak and Martin C. Rinard. An overview of the Jahob analysis system: project goals and current status. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, April 2006.
16. Hongping Lim, Dilsun Kaynar, Nancy Lynch, and Sayan Mitra. Translating timed I/O automata specifications for theorem proving in PVS. In *Proceedings of Formal Modelling and Analysis of Timed Systems (FORMATS '05)*, volume 3829 of *LNCS*, Uppsala, Sweden, September 2005. Springer.
17. Sayan Mitra. *A Verification Framework for Hybrid Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2007.
18. Sayan Mitra and K. Mani Chandy. A formalized theory for verifying stability and convergence of automata in PVS. In *In proceedings of Theorem Proving in Higher Order Logics (TPHOLS '08)*, volume 5170 of *LNCS*, pages 230–245. Springer, 2008.
19. S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.
20. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.
21. Joshua A. Tauber. *Verifiable Compilation of I/O Automata Without Global Synchronization*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2005.
22. Richard T. Vaughan, Brian Gerkey, and Andrew Howard. On device abstractions for portable, resuable robot code. In *In Proceedings of the IEEE/RSJ International Conference on Intelligent Robot Systems (IROS 2003)*, pages 2121–2427, October 2003.