Verification of Faulty Message Passing Systems with Continuous State Space in PVS

Concetta Pilotto
Computer Science Department
California Institute of Technology
pilotto@cs.caltech.edu

Jerome White
Computer Science Department
California Institute of Technology
jerome@cs.caltech.edu

Abstract

We present a library of PVS meta-theories that verifies a class of distributed systems in which agent communication is through message-passing. The theoretic work, outlined in [4], consists of iterative schemes for solving systems of linear equations, such as message-passing extensions of the Gauss and Gauss-Seidel methods. We briefly review that work and discuss the challenges in formally verifying it.

1 Introduction

We present a framework for verifying a class of distributed message passing systems with faulty communication. In this class processes (*agents*) communicate by exchanging messages that can be lost, delayed or reordered. In earlier work, we introduced and analyzed the theoretic basis for this protocol by presenting a class of iterative message-passing schemes for solving systems of linear equations [4]. This class of problems has practical applications in areas such as robot pattern formation protocols [15, 5]. For example, decentralized schemes where groups of robots form a fence around a dangerous area can be expressed as message-passing algorithms where agents solve a system of linear equations. Our previous work formalized this class of systems and proved their convergence to the solution, even in the presence of faulty communication. This paper is an addendum to that work, providing a formal model of the problem in PVS [16], and presenting its verification using the tool.

A challenge that arises in verification of distributed systems in general, and this problem in particular, is that systems exhibit a dense state space operating over continuous time. These systems are highly nondeterministic and contain uncountable data-types. Such properties make its verification using implementation based tools, such as model checkers, difficult, and in some cases impossible. As is outlined in this paper, a theorem prover is an appropriate tool to address these issues.

Our model consists of a library of PVS meta-theories [17] built on top of I/O automata [11, 2] with extensions for timed and hybrid systems [10, 14, 13]. The library offers a solution to the verification of distributed linear system solvers in general. By exposing various assumptions, it is applicable to instances of this problem in particular. Our work follows a very large body of literature where theorem provers have been used for modeling [7, 8, 1, 3], and verification [9, 12, 6]

This paper is organized as follows. Section 2, summarizes the class of distributed systems we focus on. Section 3 describes the architecture of our framework, showing PVS code fragments where necessary. Section 4 discusses our verification procedure and presents an application of our framework to an existing pattern formation protocol. We conclude with Section 5.

2 Faulty Message Passing Distributed System

In this section, we discuss a class of distributed systems where agents interact by sending messages. Messages may be lost, delayed or arrive out of order. This class consists of message-passing iterative schemes for solving systems of linear equations of the form $Ax = b$, where A is a real-valued matrix of size $N \times N$, and x, b are real-valued vectors of length N, with $N > 0$. The goal of these systems is to iteratively compute the vector x starting from an initial guess vector $x0$. The message passing version of the Gauss, Jacobi and Gauss-Seidel algorithms are examples of iterative schemes belonging to this class.

We model this as a distributed system of N agents that communicate via a faulty broadcast channel. Each agent is responsible for solving a specific variable using a specific equation of the system of linear equations. For example, agent i is responsible for solving the variable $x[i]$ using the i-th equation. We consider schemes where each agent repeatedly broadcasts a message containing the current value of its variable, $x[i]$. We assume agents broadcast their value infinitely often, where the number of messages sent within a finite time interval is assumed to be finite. Upon receiving a message, an agent computes a value for its variable using its equation. It assumes that its variable is the only unknown of the equation and sets the values of the other variables to the last message received from the corresponding agents. For example, agent i sets the value $x[i]$ as follows:

$$x[i] := b[i] - \sum_{j \neq i} A[i, j]x[j] \tag{1}$$

where $x[j]$ stores the last message received from agent j. We assume that the matrix A (D1) is invertible, (D2) has diagonal entries are all equal to 1, (D3) is weakly diagonally dominant ($\forall i : \sum_{j \neq i} A[i, j] \leq A[i, i]$), and (D4) is strictly diagonally dominant in at least one row ($\exists k : \sum_{j \neq k} A[k, j] < A[k, k]$).

Agents within our model use two variables, x and z, to represent the solution of their equation. Given agent i, the value of $x[i]$ is the current solution to Eq. (1), while the value of $z[i]$ is the agents current estimate of the solution. Initially, both $x[i]$ and $z[i]$ are set to $x0[i]$. When an agent receives a message, it updates the solution $x[i]$ using Eq. (1). The variable $z[i]$ is updated when the agent "moves," at which point it evolves $z[i]$ according to some predefined dynamics towards $x[i]$.

Using two variables to store the solution vector gives us a better means to represent continuous dynamics. In real-world applications, it is unrealistic to assume instantaneous updates of the solution vector x; thus, we maintain the current estimate vector z. In robotics, for example, systems of equations are used in pattern formation protocols, where x can be thought of as agent positions. When a robot receives a message, the move from its current location to its newly computed one is not instantaneous. Instead, the robot moves according to some time-related dynamics. In robotics application, z models the current robot positions while x represents their destination locations.

In earlier work, we have provided sufficient conditions for proving correctness of this class of distributed systems [4]. Specifically, these schemes are correct if x and z converge to the solution of the system $A^{-1}b$ as time approaches infinity. More formally,

Theorem 1 ([4]). *If A satisfies D1–4, then the system converges to $A^{-1}b$.*

In the proof, it is shown that the maximum error of the system at each point of the computation eventually decreases by a factor of α, with $\alpha \in [0, 1)$. Convergence is then proven using induction over the agent set. As a base case, we show that eventually the maximum error of the agents satisfying D4 is reduced by α. Then, assuming that this is the case for agents at distance k from some agent satisfying D4, we show that the property holds for all agents at distance $k+1$. This induction proof can be represented as a forest of trees of size N, rooted at agents satisfying D4. Iterating the proof, it is possible to show that starting from some initial maximum error of the system, E_0, the error eventually decreases to αE_0, then to $\alpha^2 E_0$, then $\alpha^3 E_0$, ..., converging to 0 as time goes to infinity.

3 PVS Verification Framework

In this section we describe the tool for verifying this class of distributed systems. Our framework [17], summarized in Figure 1, has been built on the top of the PVS formalization of I/O automata [11, 2] and

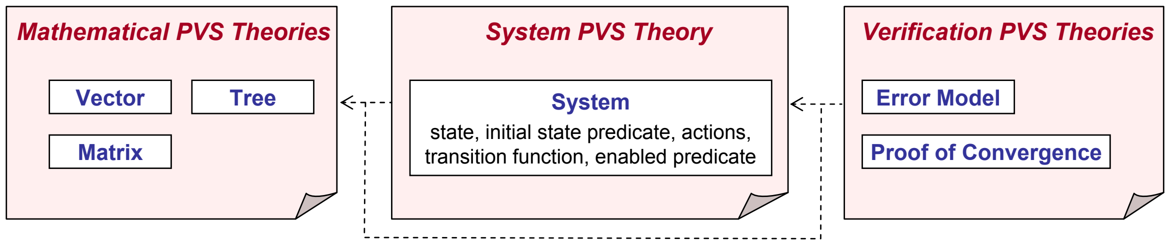
Mathematical PVS Theories
Vector
Tree
Matrix
System PVS Theory
System
state, initial state predicate, actions, transition function, enabled predicate
Verification PVS Theories
Error Model
Proof of Convergence

Figure 1: The structure of our PVS framework. A set of mathematical theories describe generic data-types (Section 3.1). These are used by the theory that models a distributed message passing system (Section 3.1). Finally, we verify that the system converges to the solution using properties of the mathematical structures (Section 3.2).

its extensions to timed and hybrid I/O automata [10, 14, 13]. Our library uses the parts of the PVS NASA library [1]. First, we present the formalization of the system, then discuss the proof of correctness.

3.1 Modeling a Distributed Message-Passing System

The system is modeled using the automaton meta-theory [14] that we have specialized for matrix algebra. The automaton consists of system state (Figure 2), system action (Figure 3(b)), enabling condition for those actions (Figure 4(b)), and transition functions (Figure 4(a)). The enabling condition is a means of defining which subset of actions are valid in a given state. Our action set handles system communication and timing, along with subsequent agent state changes. In this section, we describe the system's overall operation from the standpoint of our action set.

We have added the appropriate linear algebra types for the matrices and vectors to the automaton theory:

```
Vector: TYPE = [Index -> real]
Matrix: TYPE = [Index, Index -> real]
```

A vector is modeled as a function from type `Index` to real numbers, and a matrix as a function that maps an `Index` pair to a real number. We have defined these types functionally to facilitate manipulation in PVS. Users supply our library with instances of these types, which we refer to throughout with the variables `A`, of type `Matrix`, and `b` and `x0`, of type `Vector`. The type `Index` is an interval of natural numbers bounded by `N`, with `N` being the number of agents in the system

```
Index: TYPE = below(N)
```

That is, `Index` $= \{0,\ldots,\mathtt{N}-1\}$. The type `Vector` and `Index` belong to the PVS NASA library [1], while the type `Matrix` is our extension to this library.

3.1.1 System state

The state of the system is made up of the state of the agents, along with the state of the communication channel. An agent is responsible for its current and target values, along with the set of messages in the communication channel for which it is the recipient. We define the system state in PVS with the record type `S`, outlined in Figure 2. The fields `target`, and `lastmsg` describe the state of the agents, while `buffer` describes the state of the channel. The field `now` corresponds to the clock of the system, storing the current time. The field `next` is a vector containing, for each agent, the future time that that agent is allowed to execute a `send` action. The `target` field stores the target value of each agent. The `lastmsg`

```
S: TYPE = [# target:  Vector,                      % vector of agents
             lastmsg: Matrix,                      % matrix of values
             buffer: [Index, Index -> Pset],       % state of the channel
             now:     nonneg_real,                 % system clock
             next:   [Index -> nonneg_real] #]     % agent send deadlines
```

Figure 2: System state. Refer to Figure 3(a) for the definition of `Pset`.

```
Msg  :TYPE = [# loc: real,
                 id: Index #]
Pkt  :TYPE = [# msg: Msg,
                ddl: posreal #]
Pset:TYPE = set[Pkt]
b    :posreal
d    :posreal
```

(a) Channel types

```
ACS: DATATYPE BEGIN
 send(p:Pkt, i:Index, d1:posreal): send?
 receive(p:Pkt, i:Index): receive?
 move(i:Index, delta_t:posreal): move?
 msgloss(p:Pkt, i:Index): msgloss?
 nu_traj(delta_t:posreal): nu_traj?
END ACS
```

(b) Actions of the system

Figure 3: Components of the system automaton

field is a matrix in which the diagonal entries hold the current value of each agent; the non-diagonal entries store the last message that agent i has received from agent j. The `target` and the diagonal entries of `lastmsg` correspond to the variables x and z, respectively, from the mathematical model outlined in Section 2. The field `buffer` models the communication between agents; each pair of agents (i, j) has a directed communication channel, with i being the sender of the messages in the channel and j being the receiver. The entry `buffer(i)(j)` of type `Pset` contains the set of packets sent by i and not yet received by j. The type `Pset` is defined in Figure 3(a).

The initial condition of the system is described using the predicate *starts*?. It holds in the state `s` if the global clock of this state is set to 0, and the `target` set to the initial guess, `x0`:

```
start?(s: S): bool =
 now(s) = 0 AND (FORALL(i: Index): next(s)(i) <= d) AND
 target(s) = x0 AND (FORALL(i, j: Index): lastmsg(s)(i,j) = x0(i))
```

The condition on `next` vector is necessary because in our model agents send messages infinitely often. This condition ensures that all agents will execute a `send` action in bounded time. Parameter `d`, defined formally in Section 3.1.2, is the upper bound on an agents sending rate. Note that the communication channels are not necessarily empty initially.

3.1.2 Communication channel

The communication layer is a broadcast channel allowing for lost, delayed, or out-of-order messages. Our model assumes that (C1) messages are either lost or delivered in finite but unknown time and (C2) for each pair of communicating agents the receiver receives infinitely many messages. The first assumption models a communication medium with bounded, but unknown, delay. The second assumption ensures that there are no permanent partitions between communicating agents.

To model such faulty communication, we consider not only packets and channels, but timing and ordering properties as well. Figure 3(a) outlines the PVS data types used for this purpose. Messages

```
trans(a: ACS, s: S): S =
CASES a OF
 nu_traj(delta_t): % ...
 move(i,delta_t): % ...
 send(p,i,d1): % ...
 receive(p,i): % ...
 msgloss(p,i): % ...
ENDCASES
```

(a) Transition function of the system describing the behavior of each action of the system.

```
enabled(a: ACS, s: S): bool =
CASES a OF
 nu_traj(delta_t): % ...
 move(i,delta_t): % ...
 send(p,i,d1): % ...
 receive(p,i): % ...
 msgloss(p,i): % ...
ENDCASES
```

(b) Enabling predicate of the system describing the enabling condition of each action of the system.

Figure 4: Components of the system automaton. The body of the transition function and enabling predicate are defined in Section 3.1.3

sent between agents (`Msg`) are represented as a record, consisting of the agent's location and identifier. Messages, along with their delivery deadline, are contained within packets (`Pkt`). Sets of packets (`Pset`) make up a dedicated, directed, channel between two agents. Because a set lacks ordering, it makes it an appropriate type for a communication channel that allows for out-of-order messages. Timing within the channel is handled by the constants `b` and `d`. The former is an upper bound on packet deadlines—at most `b` units of time—and models maximum message delay. In our model, each packet can be received at any time in the interval $[\mathtt{now}, \mathtt{now}+\mathtt{b}]$. The constant `d` is an upper bound on the interval between consecutive `send` actions. Combined with `next`, it ensures that the `send` action is executed infinitely often. The future time that an agent is allowed to execute a `send` action is set to a value in the interval $[\mathtt{now}, \mathtt{now}+\mathtt{d}]$.

3.1.3 System actions

Actions within our system consist of agent movement and message transmission, channel manipulation, and system clock maintenance. Their corresponding PVS definitions are outlined in Figure 3(b) and 4. The `send`, `receive`, and `move` actions are executed by agents, while the `msgloss` action is used by the communication channel to simulate packet loss. Finally, the `nu_traj` action updates the system time. This section describes the behavior of each action, implemented by the function `trans` (see Figure 4(a)), and when they are enabled, implemented by the predicate `enabled` (see Figure 4(b)).

New trajectory. The `nu_traj` action advances the time variable of the system, `now`, by `delta_t` units, where `delta_t` is the input parameter of the action;

```
nu_traj(delta_t: posreal): s WITH [now := now(s) + delta_t]
```

It is enabled when the new value of the global clock does not violate a packet deadline:

```
nu_traj(delta_t): FORALL(p: Pkt): ddl(p) >= now(s) + delta_t
```

Agent move. The `move` action models the movement of an agent from its current value to one based on its locally computed solution to the equation of the system. The parameters of the action are the agent that moves and the time interval. In our implementation, agent i sets $z[i]$ (stored in `lastmsg(i,i)`) to $x[i]$ (stored in `target(i)`) and advances the global clock of `delta_t` units. In PVS

```
move(i: Index, delta_t: posreal): s WITH
  [ lastmsg := lastmsg(s) WITH [ (i,i) := target(s)(i) ],
        now := now(s) + delta_t]
```

This action can be executed only if packet deadlines are not violated by the new time of the system:

```
move(i, delta_t): FORALL(p: Pkt): ddl(p) >= now(s) + delta_t
```

Agent send. When executing a `send` action, agent `i` broadcasts its packet `p` to all agents in the system and schedules its next `send` action:

```
send(p: Pkt, i: Index, d1: posreal): s WITH [
  buffer := LAMBDA (k, j: Index):
   IF ((k = i) AND (j /= i)) THEN union(p, buffer(s)(k,j))
   ELSE buffer(s)(k,j)
   ENDIF,
  next := next(s) WITH [(i) := next(s)(i) + d1 ]]
```

In updating the buffer, the agent is adding its packet, `p`, to all of its outgoing channels. Notice that an agent does not send a message to itself. The `send` action is executed only if the time when the agent is allowed to send is equal to the global time of the system. Furthermore, the sent packet must contain the identifier of the agent, its current target location, and correct packet deadline. The detailed PVS code follows

```
send(p, i, d1): next(s)(i) =
  now(s) AND d1 <= d AND id(msg(p)) = i AND
  loc(msg(p)) = target(s)(i) AND ddl(p) = now(s) + b
```

Agent receive. When agent `i` receives packet `p`, it updates the `lastmsg` variable, computes a new value for its `target`, and removes the packet from the channel:

```
receive(p: Pkt, i: Index):
 LET m: Msg = msg(p), j: Index = id(m), l: real = loc(m),
     Ci: vector = update(row(lastmsg(s), i), j, l) IN s WITH
  [ buffer := buffer(s) WITH
    [ (j,i)   := remove(p, buffer(s)(j,i)) ],
      lastmsg := lastmsg(s) WITH [ (i,j) := l ],
      target  := target(s)  WITH [   (i) := gauss(Ci,i) ]]
```

The `gauss` function implements Eq. (1). The action can be executed if the `p` is in the channel from `msg(p)` to `i`, and its deadline does not violate the global time of the system,

```
receive(p,i): buffer(s)(id(msg(p)), i)(p) AND ddl(p) >= now(s)
```

Message loss. Message loss is modeled by removing a given packet from a directed channel:

```
msgloss(p: Pkt, i: Index): LET m: Msg = msg(p), j: Index = id(m) IN s
  WITH [ buffer := buffer(s)
    WITH [ (j,i) := remove(p, buffer(s)(j,i)) ]]
```

It is enabled only if the packet belongs to this channel:

```
msgloss(p,i): buffer(s)(id(msg(p)), i)(p)
```

3.2 Verification Meta-Libraries

This subsection focuses on proving system correctness. We discuss how the error of the system is represented, and how it is used to prove convergence in PVS. As outlined in [4], and briefly discussed in Section 2, we prove that the error of each agent converges to 0 as time tends to infinity.

3.2.1 Error Model

The error of an agent is defined as the distance between its current value and its desired value. We define the vector of desired values, `xstar`, axiomatically, using the standard definition:

```
xstar_def_ax: AXIOM FORALL(i: Index): xstar(i) = gauss(xstar,i)
```

During system execution, the value of an agent is represented in three places: within its state, within the set of packets in transit on its outgoing channels, and within the received message field of other agents. Although these are all values of the same agent, agent dynamics, message delay and reordering do not guarantee their equality. In the proof of correctness, it is enough to consider only the maximum error of these values. We define the error of the system axiomatically

```
me_all_error: AXIOM FORALL(i: Index): mes(s,i) <= me(s)
me_ex_error:  AXIOM EXISTS(i: Index): mes(s,i)  = me(s)
```

where `mes` is the maximum error of agent `i` within in the system, and `me` is the error of the system, defined as maximum of all agents errors within the system.

3.2.2 Proof of Correctness

As discussed in Section 2, our proof of correctness relies on certain assumptions about the given matrix `A` (see D1–4). Invertibility, along with diagonal and strict-diagonal dominance, were modeled, respectively, as

```
inv?(m): bool = EXISTS(n: Matrix):
                      prod(m,n) = prod(n,m) AND diag?(prod(m,n))
dd?(m):  bool = FORALL(r: Index): sum(row(abs(m),r),r) <= abs(m(r,r))
sdd?(m): bool = EXISTS(r: Index): sum(row(abs(m),r),r) <  abs(m(r,r))
```

where `m` is of type `Matrix`. Using these definitions, we can describe the assumptions on `A` made by the model. We use the PVS assumption facility to access these properties within the meta-theory and obligate users of our library to discharge them;

```
ASSUMING
  inverse_exist: ASSUMPTION inv?(A)                           % D1
  diag_entry:    ASSUMPTION FORALL(i: Index): A(i,i) = 1      % D2
  diag_dominant: ASSUMPTION dd?(A)                            % D3
  strictly_diag_dominant: ASSUMPTION sdd?(A)                  % D4
ENDASSUMING.
```

Reasoning about system convergence requires the analysis of the system along an arbitrary execution. Our responsibility is to show that (E1) the error of the system does not increase, and that (E2) it eventually decreases by a lower-bounded amount. Using the diagonally dominant assumption on A, we can prove E1:

```
not_incr_error: LEMMA enabled(a,s) IMPLIES me(s) >= me(trans(a,s))
```

To prove E2, as discussed in Section 2, we built a representation of the forest structure in PVS. The tree is represented by functions `ancs`, `root_t?` and `parent` using the PVS list structure defined in the prelude.

```
ancs: FUNCTION[Matrix, Index -> ListIndex]
root_t?(m: Matrix, i: Index): bool = sdd?(m,i) AND null?(ancs(m,i))
parent(m: Matrix, i: Index): Index =
   IF root_t?(m,i) THEN i ELSE car(ancs(m,i)) ENDIF
```

where `ancs` is a function which, given a matrix and agent identifier, returns the complete path of the agent to a rooted node in the tree. The path is defined as a list of agent identifiers having type `ListIndex`. Note that rooted agents satisfy D4.

Next, we define a factor α by which the system will eventually decrease. Given the rooted forest, for each node of the forest we recursively define the quantity `p_value`. We prove that this value is positive and (strictly) upper bounded by 1. The factor α is the maximum of these quantities. In PVS,

```
alpha_all: AXIOM FORALL(i: Index): p_value(i) <= alpha
alpha_ex:  AXIOM EXISTS(j: Index): alpha = p_value(j)
```

Using induction on the forest, we prove that the maximum error of the system eventually decrease by α. Assuming that the error of the system is upper-bounded by W, the base case is to prove that the error of the roots of the tree eventually decreases by α. From there, we prove that eventually the error of the node is upper bounded by $W \cdot \alpha$, assuming that the error of all ancestors of a node is upper-bounded by the same quantity. To handle this within the theorem prover, we defined our own induction scheme:

```
p: VAR [Index -> bool]
induct_ancs: THEOREM
  (null?(ancs(m,i)) IMPLIES p(i)) AND
  (cons?(ancs(m,i)) AND p(car(ancs(m,i))) IMPLIES p(i))
    IMPLIES FORALL(j: I | member(j, ancs(m,i)) OR j = i): p(j)
```

This theorem ensures that if we prove that some property holds at the root of the tree, and, given a node, we prove that it holds at the node, given that it holds for its parent, then we can safely derive that the property holds for all nodes along a path of the tree. In this case, that property is the factor of α decrease.

4 Framework Discussion

In this section, we offer commentary on our experience using PVS and present an application of our framework.

4.1 PVS Implementation

Our library consists of over 30 lemmas, almost 2000 proof steps. We took advantage of PVS pre- and user-defined types for modeling agent and channel states. Implementation of the system infrastructure consumed about 30% of our effort. Vectors, matrices and trees were used extensively throughout our library. The PVS NASA libraries provided some relief, but modeling diagonally dominant matrices and proving lemmas on products of matrices and vectors forced us to extend them. Although NASA does provide a representation of trees, their recursive implementation made proving properties we required very difficult. Unlike the NASA implementation, where they visit the tree starting from the leaves, we were more interested in both proving properties on the tree starting from the root and inducting over the structure as well. Further, the NASA library has limited support for weighted directed graphs, something critical for our model. For these reasons, we preferred to give an implicit definition of trees and ensure the needed properties axiomatically.

While proving the convergence property of the system, we gave implicit definitions to many nonlinear functions for convenience within the proofs. For example, we did not define the maximum error of the system recursively. We preferred, instead, to use two axioms and define the maximum as the upper bound on the error of the agents such that the value is reached by some agent (see `me_all_error` and `me_ex_error` in Section 3.2.1). This choice reduced the number of proofs in the library, as we did not derive these facts from their recursive definitions. Proving these facts in PVS is possible, but beyond the scope of our work.

We managed the proof of Theorem 1 by breaking it into smaller lemmas. This allowed us to tackle small proofs where the goal was to show that eventually a specific property held. For example: proving that the error of the target position of an agent eventually decreases by the constant factor α; then proving that its error in the outgoing channels eventually decreases by this factor; and finally showing that its error stored in the state of the remaining agents eventually decreases by this factor. We showed these lemmas using the two assumptions on the communication medium (see C1–2). These assumptions simplified our proofs since we did not consider traces of the automaton in the proof of convergence. Using this collection of sub-lemmas, we were able to prove, directly, that after $d+b$ time units the maximum error of the agent decreases by the factor α.

4.2 Applications

In earlier work, we discussed a specific pattern formation protocol where the goal of the system was for agents to form a straight line [5]. The system consisted of N robots, with robot 0 and $N-1$ fixed throughout the execution. Agent i $(0 < i < N-1)$ only communicated with its immediate neighbors, robots $i-1$ and $i+1$. Based on this communication, i computed its new position as the average of the left and right received values.

This protocol can be modeled as a system of linear equations, and, as such, can be verified using our library. We instantiate the matrix A with a tri-diagonal matrix having the value 1 along the main diagonal, and -0.5 on the secondary diagonals (with the exception of rows 0 and $N-1$, which have 0 on the secondary diagonals). Given this structure, we are obligated to discharge the assumptions of the library outlined in Section 3.2.2: that the input matrix A satisfies D1–4.

5 Conclusions

We have presented a verification framework for distributed message-passing systems that takes into account a faulty communication medium. This framework consists of a library built within the PVS theorem prover that allows for the verification of distributed algorithms for solving systems of linear equations. We have also detailed the implementation of the framework, outlining our use of a specialized automaton theory and induction scheme. We have discussed challenges in the implementation and presented an application where the framework is used to verify a specific robot pattern formation protocol. Future work includes extending these results to a richer class of systems, such as non linear convex systems of equations, and providing their corresponding verification using a theorem prover.

6 Acknowledgments

The authors would like to thank the referees for their very helpful and constructive comments. This work was supported in part by the Multidisciplinary Research Initiative (MURI) from the Air Force Office of Scientific Research.

References

[1] Nasa langley pvs libraries. http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html, 2010.

[2] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proceedings of the 1st International Workshop on User Interfaces for Theorem Provers (UITP '98)*, July 1998.

[3] L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in isabelle/hol. In *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '07)*, volume 4732 of *Lecture Notes in Computer Science*, pages 38–53, Berlin, Heidelberg', September 2007. Springer-Verlag.

[4] K. M. Chandy, B. Go, S. Mitra, C. Pilotto, and J. White. Verification of distributed systems with local-global predicates. *To Appear: Formal Aspect of Computing*.

[5] K.M. Chandy, S. Mitra, and C. Pilotto. Convergence verification: From shared memory to partially synchronous systems. In *Proceedings of 5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS '08)*, volume 5215 of *Lecture Notes in Computer Science*, pages 217–231. Springer Verlag, September 2008.

[6] P. Frey, R. Radhakrishnan, H.W. Carter, P.A. Wilsey, and P. Alexander. A formal specification and verification framework for time warp-based parallel simulation. *IEEE Transition on Software Engineering*, 28(1):58–78, 2002.

[7] H. Gottliebsen. Transcendental functions and continuity checking in pvs. In *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '00)*, volume 1869 of *Lecture Notes in Computer Science*, pages 197–214, Berlin, Heidelberg, August 2000. Springer-Verlag.

[8] J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[9] P.B. Jackson. Total-correctness refinement for sequential reactive systems. In *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '00)*, volume 1869 of *Lecture Notes in Computer Science*, pages 320–337, Berlin, Heidelberg, August 2000. Springer-Verlag.

[10] D.K. Kaynar, N.A. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata (Synthesis Lectures in Computer Science)*. Morgan & Claypool Publishers, 2006.

[11] N.A. Lynch and M.R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[12] S. Maharaj and J. Bicarregui. On the verification of VDM specification and refinement with PVS. In *Proceedings of the 12th International Conference on Automated Software Engineering (ASE '97)*, page 280, Washington, DC, USA, November 1997. IEEE Computer Society.

[13] S. Mitra. *A Verification Framework for Hybrid Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2007.

[14] S. Mitra and M. Archer. PVS strategies for proving abstraction properties of automata. *Electronic Notes in Theoretical Computer Science*, 125(2):45–65, 2005.

[15] R. Olfati-Saber, J.A. Fax, and R.M. Murray. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95(1):215–233, January 2007.

[16] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Proceedings of 11th International Conference on Automated Deduction (CADE '92)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[17] C. Pilotto and J. White. Infospheres project. http://www.infospheres.caltech.edu/nfm, January 2010.