Verifying a Self-Stabilizing Mutual Exclusion Algorithm

S. Qadeer
University of California at Berkeley
Department of EECS, Berkeley CA 94720 U.S.A.
Phone: (510) 642-1490 Email: shaz@eecs.berkeley.edu

N. Shankar
SRI International
Computer Science Laboratory, Menlo Park CA 94025 U.S.A.
Phone: (650) 859-5272 Email: shankar@csl.sri.com

Abstract

We present a detailed description of a machine-assisted verification of an algorithm for self-stabilizing mutual exclusion that is due to Dijkstra [Dij74]. This verification was constructed using PVS. We compare the mechanical verification to the informal proof sketch on which it is based. This comparison yields several observations regarding the challenges of formalizing and mechanically verifying distributed algorithms in general.

Keywords

Self-stabilization, mutual exclusion, theorem proving, PVS.

1 INTRODUCTION

A self-stabilizing algorithm is one that ensures that the system behavior eventually stabilizes to a safe subset of states regardless of the initial state. In Dijkstra's original paper [Dij74] introducing distributed self-stabilization, he gave three self-stabilizing mutual exclusion algorithms and left their proofs as exercises for the reader. Subsequently, Dijkstra [Dij86] proffered a proof of self-stabilization of one of his algorithms while conceding that the proofs were in fact nontrivial. In this paper, we take on the challenge of mechanically verifying the self-stability of one of Dijkstra's other algorithms. This algorithm poses an interesting challenging for formalization and mechanical verification for several reasons. The self-stabilization argument involves a nontrivial amount of mathematics. Self-stability is a convergence property whereas typical attempts at the mechanical verifications of distributed algorithms mainly involve safety properties or simple progress reasoning. The use of model checking in the verification of distributed algorithms, even non-finite-state ones, is a recent trend, but there is no obvious way to apply model checking in this context. It is hard to draw substantive conclusions from the verification of one self-stabilization algorithm but we wish to use this example to touch on some of the important themes in formal verification such as:

1. What is the difference in effort between constructing a convincing informal argument and mechanically verifying it?
2. What are some of the specific obstacles to effective formalization?
3. How much of the formal verification involves only conventional mathematics as opposed to temporal reasoning?
4. Can the temporal reasoning be carried out within the framework of a conventional predicate calculus or is there a need for temporal or modal logics?

The heart of this paper is a careful elucidation of the formalization and mechanically verified proof of self-stability for Dijkstra's algorithm. The informal proof of this algorithm is itself quite interesting. A survey of the literature revealed only one attempt at a proof by Varghese [Var92][1] who presents an informal sketch of an argument that is similar to the one used in this paper. As with any distributed algorithm, the verification requires careful attention to details that are easily overlooked in an informal sketch. Our verification was conducted using PVS but our explanation of the verified proof will be presented in a conventional style thus requiring no prior knowledge of PVS [ORS92].[2]

We only consider the first of Dijkstra's three algorithms for self-stabilizing mutual exclusion. In this algorithm, there are N processes for $N > 1$ numbered 0 to $N-1$ arranged in a unidirectional ring where each process can observe its own state and that of its predecessor. Each process i maintains a counter $v(i)$ of values from 0 to $M-1$, where $N \leq M$. Process 0 is a distinguished process that is enabled when $v(0) = v(N-1)$, and when enabled, it can make a transition by incrementing its counter modulo M. A nonzero process i is enabled when $v(i) \neq v(i-1)$, and when enabled, it can update its counter value so that $v(i) = v(i-1)$. The transitions of the individual processes are interleaved. Despite the absence of a central controller, the system can be shown to converge or *self-stabilize* from an arbitrary initial state to a stable behavior where exactly one process is enabled at any state. The one enabled process in a state in such a stable behavior is allowed exclusive access to its critical section.

2 AN INFORMAL PROOF SKETCH

The reader is invited to set aside this paper and contemplate a proof of self-stability for the above algorithm by showing that any computation leading out of an arbitrary initial state eventually stabilizes to a computation where exactly one process is enabled in a state. In our initial verification attempts,

[1] See the URL `dworkin.wustl.edu/~varghese/PAPERS/dijk.ps.Z`.

[2] The URL `www.csl.sri.com/pvs/examples/self-stability/` contains the PVS specification and proofs.

we had some difficulty merely constructing a reliable, informal proof sketch. Even with a proof sketch that withstood informal scrutiny, we had a lot of difficulty formalizing the needed concepts in a form that was elegant enough to support feasible mechanical verification. We finally restarted our verification with a proof sketch consisting of the following sequence of claims:

1. *There is always at least one enabled process.* Otherwise, for each i, $i < N-1$, $v(i) = v(i+1)$ yet $v(0) \neq v(N-1)$.
2. *The number of enabled processes never increases.* Each transition due to process i affects the enabledness of at most i and $i+1 \, mod \, N$, and definitely disables i.
3. *The enabledness of each process is eventually toggled.* This is a crucial observation. For each i, let $p(i)$ be the sum over the enabled processes j of the unidirectional distance from j to i, i.e., $i - j$ for $j \leq i$, and $N - j + i$ for $j > i$. Then, whenever an enabled process j, $j \neq i$, makes a transition, $p(i)$ decreases. If i is initially enabled but never taken, then $p(i)$ eventually becomes 0. By Claim 1, i is the only remaining enabled process, which must therefore be taken leaving i disabled. If i is initially disabled, then when $p(i)$ is 0, it must be enabled by Claim 1.
4. *Process* 0 *is eventually incremented.* This follows trivially from Claim 3.
5. *Process* 0 *eventually takes on any value below M.* Follows by induction from Claim 4.
6. *There is some value x below M so that for any i, $i \neq 0$, we have $v(i) \neq x$.* This is a consequence of the pigeonhole principle which is the key property used to guarantee self-stability.
7. *Eventually $v(0) = x$ and for all i, $i \neq 0$, we have $v(i) \neq x$.* Let x be chosen as in Claim 6. By Claim 5, eventually $v(0) = x$. Up to (and including) the earliest point where $v(0) = x$, we have $v(i) \neq x$ for any i, $i \neq 0$
8. *Once $v(0) \neq v(i)$ for all i, $i \neq 0$, then process* 0 *is not enabled again until for all processes i, $v(i) = v(0)$.* This is because given Claim 7, there eventually is a prefix of n processes, for any $n < N$, so that $v(i) = v(0)$ holds for $i \leq n$.
9. *The system eventually reaches a stable state with exactly one enabled process.* By Claim 8, there is eventually a state where only process 0 is enabled, and by Claims 1 and 2, the system will remain stable in a state where there is exactly one enabled process.

Though this informal proof sketch is fairly convincing, the formalization and mechanical verification of Dijkstra's self-stabilizing mutual exclusion protocol using PVS are not straightforward. The formalization and verification are described in Sections 3 and 4.

3 THE FORMALIZATION OF THE PROBLEM

Both the algorithm and self-stability property are formalized as predicates on sequences of states. The global *state* in this case consists of an array with indices in the subrange 0 to $N - 1$ and with elements in the subrange 0 to

$M-1$. A sequence of such states is a function from the type of natural numbers (i.e., non-negative integers) to the *state* type.

A process i is *enabled* in state v when the curried predicate $E(v)(i)$ holds. For process 0, the predicate $E(v)(0)$ holds when $v(N-1) = v(0)$. For a nonzero process i, $E(v)(i)$ holds when $v(i) \neq v(i-1)$. The curried form $E(v)(i)$ is chosen so that we can also employ the predicate $E(v)$ as (the characteristic predicate of) a set of processes.

A sequence of states σ is a function from natural numbers to the *state* type. A sequence of states σ is a *run*, i.e., $run(\sigma)$ holds, when each pair of adjacent states is in U. Two states v and v' are in the next-state relation U (for *update*), i.e., $U(v, v')$, if there exists a process i that is enabled in state v and $UP(i)(v, v')$ holds. The relation $UP(i)(v, v')$ (for *update process*) holds if (1) $i = 0$ and $v'(0) = v(N-1) + 1 \, mod \, M$ and $v'(j) = v(j)$ for $j \neq 0$, or (2) $i \neq 0$ and $v'(i) = v(i-1)$ and $v'(j) = v(j)$ for $j \neq i$.

The variables u, v, v' range over states. The variables x, y, z range over counter values. All increment operations applied to $v(i)$, x, y, z are assumed to be modulo M. The variables i, j, and k range over process numbers in the subrange 0 to $N-1$. All increment and decrement operations applied to i, j, k are assumed to be modulo N. The various definitions that are used in the proof are summarized in Figure 1.

Term	Definition
$E(v)(i)$	$\begin{cases} v(N-1) = v(0), & \text{if } i = 0 \\ v(i) \neq v(i-1) & \text{otherwise} \end{cases}$
$UP(i)(v, v')$	$\begin{cases} v' = v\{0 \leftarrow v(0)+1\}, & \text{if } i = 0 \\ v' = v\{i \leftarrow v(i-1)\} & \text{otherwise} \end{cases}$
$U(v, v')$	$\exists i : UP(i)(v, v')$
$run(\sigma)$	$\forall n : U(\sigma(n), \sigma(n+1))$
$d(i)(j)$	$\begin{cases} i - j, & \text{if } j \leq i \\ N + i - j, & \text{otherwise} \end{cases}$
$D(v)(i)$	$\sum_{j=0}^{N-1} d(i)(j) : j \in E(v)$
$\sigma[n]$	$\lambda\, m : \ \sigma(m+n)$
$S(v)$	$\{x \mid x < M \wedge \exists i : i \neq 0 \wedge v(i) = x\}$
$prefix(v)(j)$	$(\forall i : i \leq j \equiv v(i) = v(0))$

Figure 1 Summary of Definitions

4 THE MECHANICAL VERIFICATION

We informally describe the steps in the mechanical verification using PVS. It is worth noting that PVS was used interactively to discover the detailed justifications based on the informal sketch above. The elaboration of the informal sketch is not at all intellectually trivial. First, the formalization, namely the choice of representations and the form of definitions and theorems, has to be carried out with a tremendous amount of care since otherwise the formal justifications can easily become unmanageably large. Second, it can be quite difficult to come up with formal justifications even for informal steps that seem intuitively obvious. It is this latter aspect of the informal to formal transition that we wish to highlight in presenting the details of the mechanical verification below.

Initial proof attempts such as the ones described below tend to be verbose as they represent the form in which the proofs were discovered. It is customary to devote some effort to polishing PVS proofs so as to enhance their conciseness while exploiting the available automation. We indicate the number of interactions needed to construct the mechanically checked proof of individual formulas in order to convey a rough idea of the effort needed. This number does not have any precise significance since some interactions consist of commands with many inputs that invoke complex proof strategies, while others consist of simple atomic inference steps. Note that the verification relies on pre-existing PVS libraries for the *mod* operation and for finite cardinalities.[3]

Lemma 4.1 corresponds to Claim 1.

Lemma 4.1 *There is always at least one enabled process. Formally,* $\forall v \ : \exists i \ : E(v)(i)$.

Proof. Suppose that $\forall j : v(j) = v(0)$, then in particular, $v(N-1) = v(0)$, but this would imply that process 0 is enabled. Otherwise, when $\forall j : v(j) = v(0)$ does not hold, there is a least j, $j \neq 0$ such that $v(j) \neq v(0)$ and since $v(j) \neq v(j-1)$, process j is enabled. The second part of this proof is actually formally proved by induction on j.

The mechanization of this entire proof consists of ten interactions. The first two steps are universal quantifier elimination and case splitting. The first case involves five interactions for existential quantifier instantiation and simplification. The second case involves three interactions for induction/simplification and quantifier instantiation followed by simplification. □

Lemma 4.2 is a step in the proof of Lemma 4.3 which corresponds to Claim 2.

[3] The URL `www.csl.sri.com/pvs.html` contains links relevant to PVS including pointers to the groups using PVS, several example verifications, and an extensive bibliography of papers by PVS users.

Lemma 4.2 *An update to an enabled process i removes i and could otherwise at most insert or remove $i+1$ from the set of enabled processes. Formally,*

$$UP(i)(v, v') \supset E(v') = \begin{cases} E(v) - \{i\} \bigcup \{i+1\}, & \textit{if } E(v')(i+1) \\ E(v) - \{i, i+1\}, & \textit{otherwise} \end{cases}$$

Proof. There are two parts to this proof. First, we establish that $i \notin E(v')$. This is trivial when $i \neq 0$ since $v'(i) = v'(i-1)$. When $i = 0$, a previously proved lemma yields $v(0)+1 \neq v(0) \, mod \, M$ which is used to show that $v'(0) \neq v'(N-1)$. The second part of the proof essentially demonstrates that only the enabledness of i and $i+1$ are affected when i is updated. The extensionality rule for set equality is applied to the conclusion which then follows from the lemmas $x < N \supset x \, mod \, N = x$ and $N \, mod \, N = 0$ by definition expansion and simple set-theoretic reasoning.

The mechanization has three preliminary interactions for quantifier elimination, definition expansion (for UP) and propositional simplification. The next step introduces the case analysis for splitting the proof into two parts. The first part is propositionally simplified into two cases. The first case is proved by introducing a lemma instance followed by simplification and definition expansion. The second case is proved directly by expansion/simplification. The second part of the proof has three interactions for using extensionality, introducing a lemma instance, and brute-force expansion/simplification. The last brute-force step is not entirely pleasant since a lot of essential detail has been buried in the automation. A more careful argument would be to show that when $UP(i)(v, v')$, then for $j \notin \{i, i+1\}$, $j \in E(v')$ iff $j \in E(v)$. □

Lemma 4.3 *A transition never increases the cardinality of the set of enabled processes. Formally, $U(v, v') \supset |E(v')| \leq |E(v)|$.*

Proof. This lemma is of course a simple consequence of Lemma 4.2. There is, however, one minor complication. The cardinality operation applies only to finite sets and the sets $E(v)$ and $E(v')$ have to be shown to be finite. The typechecker generates proof obligation to this effect. This proof obligation is discharged by exhibiting an injection from these sets to some bounded initial segment of the natural numbers. Since the elements of $E(v)$ are already chosen from the subrange of numbers below N, we can choose N as the bound and the identity function on these sets as the required injection. The proof of the lemma follows from Lemma 4.2 using the above proof obligations (which have to be proved separately) as lemmas. This proof also invokes simple cardinality lemmas regarding the operations of adding and removing elements from a finite set: $|E(v) - \{i\}| = |E(v) - 1|$ when $i \in E(v)$, and hence, $|(E(v) - \{i\}) \bigcup \{i+1\}| \leq |E(v)|$.

The mechanization uses three interactions to bring in Lemma 4.2 and the finiteness proof obligations. A single `GRIND` step, the catch-all brute-force strategy[4] completes most of the proof while leaving one trivial subgoal. This is however quite misleading because the arguments to `GRIND` are quite specific about which lemmas and definitions are to be used, and which ones excluded. The main observation is that though such reasoning about cardinalities is entirely trivial in informal terms, there are a number of bookkeeping details that are needed in the formal proof to ensure that all the sets involved are indeed finite. □

We now formally define the measure that is used in Claim 3 to demonstrate that each process is always eventually enabled and eventually disabled. The formalization deviates slightly from the informal script. We can avoid the case analysis in Claim 3 but use the very same measure to show the equivalent fact that each process is always eventually updated. The measure is defined in terms of a summation operation $\sum_{i=0}^{n} f(i) : i \in P$ which takes a function f, a set P, and an upper bound n and adds up the $f(i)$ for $0 \leq i \leq n$ such that $i \in P$ holds. It is easy to prove the following lemmas by induction on n followed by a handful of simplification steps.

Lemma 4.4

$$\sum_{i=0}^{n} f(i) : i \in (P - \{j\}) = \begin{cases} \sum_{i=0}^{n} f(i) : i \in P - f(j), & \text{if } j \leq n \wedge j \in P \\ \sum_{i=0}^{n} f(i) : i \in P, & \text{otherwise} \end{cases}$$

Lemma 4.5

$$\sum_{i=0}^{n} f(i) : i \in (P \bigcup \{j\}) = \begin{cases} \sum_{i=0}^{n} f(i) : i \in P, & \text{if } j > n \vee j \in P \\ \sum_{i=0}^{n} f(i) : i \in P + f(j), & \text{otherwise} \end{cases}$$

Let $d(i)(j)$ be the unidirectional distance from i to j defined as $i - j$ when $j \leq i$ and $N + i - j$, otherwise.

Then for any process i, the measure for i in state v is given by $D(v)(i)$ which is defined as $\sum_{j=0}^{N-1} d(i)(j) : j \in E(v)$. In words, this measure is the sum of the unidirectional distance of the enabled processes from i in state v. This simple definition of the measure function was constructed after several failed attempts at mechanization. In our earlier attempts, we tried to define the measure by summation over the processes k steps away from j and this made all the inductions very complicated.

Lemma 4.6 *The measure $D(v)(j)$ for j decreases with any update to an enabled non-j process. Formally,*

$$i \neq j \wedge \mathit{UP}(i)(v, v') \supset D(v')(j) < D(v)(j).$$

[4]The `GRIND` strategy performs quantifier elimination and instantiation, propositional simplification, rewriting using lemmas as rewrite rules, definition expansion, explicit case analysis according to the case structure in the goal, and does many of these steps repeatedly until no further simplification is possible.

Proof. First, note that $i \in E(v)$. By Lemma 4.2, the goal simplifies to considering $E(v')$ to either be $E(v) - \{i\} \bigcup \{i+1\}$ or $E(v) - \{i\}$. In the first case, we have a further case analysis according to whether $j = i + 1$. If $j = i + 1$, then since $i \in E(v)$, we have by Lemmas 4.2 and 4.4 that $D(v')(j)$ is $D(v)(j) - d(j)(i) + 0$. Since $i \neq j$, we have $d(j)(i) > 0$ and $D(v')(j) < D(v)(j)$. If $j \neq i+1$, then by Lemmas 4.2, 4.4, and 4.5, $D(v')(j)$ is at most $D(v)(j) - d(j)(i) + d(j)(i+1)$. We can prove that when $i \neq j$, $d(j)(i) > d(j)(i+1)$ and hence $D(v')(j) < D(v)(j)$. In the second case when $E(v')$ is $E(v) - \{i\}$ and since $i \in E(v)$ and $D(v')(j) = D(v)(j) - d(j)(i)$, we again have by $d(j)(i) > 0$ that $D(v')(j) < D(v)(j)$.

The mechanical verification carries out a similar case analysis. Of the four cases, three are trivially discharged by the `GRIND` strategy and the remaining case has three further steps using a lemma about the *mod* operation. □

Lemma 4.7 *No non-j process is enabled when the measure for j is 0 in state v. Formally, $D(v)(j) = 0 \supset (\forall i : i \neq j \supset i \notin E(v))$.*

Proof. This is proved by first expanding the definition of D in terms of summation and generalizing the theorem to bound the summation by some n that is below N (rather than $N - 1$) so that the theorem can now be proved by induction. The induction proof is then straightforward since if $i \neq j$, then $d(j)(i) > 0$ so if the summation $\sum_{k=0}^{n+1} d(j)(k) : k \in E(v)$ equals 0, then if $i = n + 1$ it must be because $n + 1 \notin E(v)$. If $i \leq n$, then since $\sum_{k=0}^{n} d(j)(k) : k \in E(v)$ equals 0, the induction hypothesis can be used to demonstrate that $i = j$ or $i \notin E(v)$.

The mechanical proof requires ten interactions that follow the above outline. The default brute-force induction strategy fails primarily because the definition of summation defeats a heuristic that bounds the depth to which recursive definitions are expanded, and also the instantiation heuristic is unable to find the correct instantiation for the quantified variables. □

Up to this point, we have merely proved basic mathematical facts or theorems about a single transition step in a conventional mathematical style. We have not introduced any temporal properties such as invariants or eventualities. We now show the first temporal property (Claim 3) that a process i is eventually updated in any run of the system. A run of the system is a sequence σ such that $\forall n : U(\sigma(i), \sigma(i+1))$ holds. Note that any suffix of a run is also a run. The statement of the theorem is that if σ is a run, then $\exists n : UP(i)(\sigma(n), \sigma(n+1))$. We did this proof in two different ways. In the first attempt, we proved it directly by measure induction on the measure function. In the second case, we established an eventuality rule that was used to prove the goal. Somewhat unexpectedly, the latter proof attempt turned out to be less straightforward than the first one. The main reason is that the overhead of using the eventuality rule outweighed the savings. We explain the verification using the eventuality rule below.

Let the suffix of σ which is the sequence $\sigma(n), \sigma(n+1), \ldots$ be denoted by $\sigma[n]$. The eventuality rule establishes $\exists n : R(\sigma[n])$ for a given trace predicate R by requiring a measure function f on a trace with a well-founded ordering on the range type (such as the usual $<$ relation on natural numbers) such that for any run θ, either $R(\theta)$ or $f(\theta[1]) < f(\theta)$.

Theorem 4.8 *If there is a function f from sequences to natural numbers such that*

$$\forall \theta : run(\theta) \supset R(\theta) \vee f(\theta[1]) < f(\theta)$$

then

$$\forall \sigma : run(\sigma) \supset \exists n : R(\sigma[n]).$$

Proof. The conclusion is proved by measure induction on $f(\sigma)$. By the premise, $R(\sigma)$ or $f(\sigma[1]) < f(\sigma)$. When $R(\sigma)$ holds, we already have the conclusion $R(\sigma[0])$ since $\sigma[0] = \sigma$. When $f(\sigma[1]) < f(\sigma)$ holds, we can apply the induction hypothesis on $\sigma[1]$ to get $R(\sigma[1][n])$ which is the same as $R(\sigma[n+1])$ and hence the conclusion.

The eventuality rule is easily proved in about thirteen steps mainly for invoking measure induction and providing quantifier instantiations, expanding definitions, and applying extensionality to demonstrate equality between sequences. $\square$

Lemma 4.9 *Each process i is eventually updated in any run. Formally, $run(\sigma) \supset \exists n : UP(i)(\sigma(n), \sigma(n+1))$.*

Proof. With the eventuality rule, we can avoid measure induction in the proof that in any run σ, process i is eventually updated, i.e., $\exists n : UP(i)(\sigma(n), \sigma(n+1))$. This removes one major intellectual obstacle from the proof. We still need to identify the measure in terms of σ. In this case, the measure is $D(\sigma(0))(i)$. Four interactions are needed to introduce and properly instantiate the eventuality rule. Four more interactions are needed to show that the conclusion of the eventuality rule $run(\sigma) \supset \exists n : R(\sigma[n])$ can be instantiated to yield the conclusion $run(\sigma) \supset \exists n : UP(i)(\sigma(n), \sigma(n+1))$. The corresponding premise of the eventuality rule

$$UP(i)(\sigma(0), \sigma(1)) \vee D(\sigma(1))(i) < D(\sigma(0))(i)$$

can be proved by first expanding run to get $\exists j : UP(j)(\sigma(0), \sigma(1))$. If $i = j$ then we have $UP(i)(\sigma(0), \sigma(1))$. Otherwise, we have to invoke Lemma 4.6 to get $D(\sigma(1))(i) < D(\sigma(0))(i)$. The mechanical proof of this use of the eventuality rule takes about twelve interactions. Overall, the proof of this lemma takes twenty interactions, whereas the direct proof of this theorem by measure induction and without using the eventuality rule took 21 interactions. $\square$

Lemma 4.10 corresponds to Claim 4.

Lemma 4.10 *Process* 0 *is eventually incremented in any run. Formally,*

$$run(\sigma) \supset \exists n : \sigma(n)(0) = \sigma(0)(0) + 1.$$

Proof. This claim is a consequence of Lemma 4.9 but not obviously so. We know from Lemma 4.9 that process 0 will eventually be updated but only the first such update will lead to the desired increment. We also need to ensure that none of the intervening transitions affects the state of process 0. If σ is a run, let m be given by Lemma 4.9 so that $UP(0)(\sigma(m), \sigma(m+1))$ holds. We instead prove that

$$\forall m, \sigma : run(\sigma) \wedge UP(0)(\sigma(m), \sigma(m+1)) \supset \exists n : \sigma(n)(0) = \sigma(0)(0) + 1$$

by induction on m. If m is 0 then we can let n be 1 since by the definition UP, $\sigma(1)(0) = \sigma(0)(0) + 1$. In the induction step when $m > 0$, we know by the definition of run that $UP(i)(\sigma(0), \sigma(1))$ holds for some i. If i is 0, then once again, we can let n be 1 since by the definition of UP, $\sigma(1)(0) = \sigma(0)(0) + 1$. If $i \neq 0$ then $\sigma(1)(0) = \sigma(0)(0)$. Since in $\sigma[1]$, the occurence of $UP(0)(\sigma(m), \sigma(m+1))$ is reachable in $m-1$ steps, we can instantiate the induction hypothesis by $\sigma[1]$. Since we know that $run(\sigma[1])$ and $UP(0)(\sigma[1](m-1), \sigma[1](m))$ hold, we can conclude by the induction hypothesis that there is an n' such that $\sigma[1](n')(0) = \sigma[1](0)(0) + 1$ and since $\sigma(1)(0) = \sigma(0)(0)$ we have $\sigma(n'+1)(0) = \sigma(0)(0) + 1$ yielding $n' + 1$ as the desired witness for n.

The mechanical proof essentially follows each step of the above outline. The proof could not be carried out more automatically since some of the quantifier instantiations were not found by the simple syntactic matching techniques used by PVS to instantiate quantifiers. This is the first example of a theorem that is fairly obvious but whose proof requires a fair amount of formal machinery for extracting the induction scheme that underlies the intuitive reasoning and for carefully instantiating quantifiers. □

Lemma 4.10 can be used to prove Lemma 4.11 which corresponds to Claim 5.

Lemma 4.11 *Process* 0 *can eventually take on any counter value. The formal statement is*

$$\forall x : run(\sigma) \supset \exists n : \sigma(n)(0) = x.$$

Proof. The proof is by measure induction on the number of increments needed to reach x from $\sigma(0)(0)$. If none are needed, then we can clearly choose n to be 0. Otherwise, by Lemma 4.10 we have an n' such that $\sigma(n')(0) = \sigma(0)(0) + 1$. We can now apply the induction hypothesis to $\sigma[n']$ since we one fewer increment is needed to reach x. By the induction hypothesis, we have an n'' such that $\sigma[n'](n'')(0) = x$. The witness n for the main conclusion then can be chosen to be $n' + n''$.

The primary points where the mechanical proof has to be carried out delicately are in some of the definition expansions, the quantifier instantiations, and in some steps involving properties of *mod*. Although the proof involves 27 interactions, many of these involve simple but important details such as type correctness proof obligations. The outline of the proof matches the above description and the details are quite straightforward. $\square$

The pigeonhole principle (Claim 6) is a classic example of a theorem that is substantially more obvious than its proof. The principle can be stated in many forms but the one that is useful to us is shown below. Informally, the principle states that if each of $n+1$ pigeons is assigned one of n pigeonholes, then some hole must contain at least 2 pigeons.

Lemma 4.12 (Pigeonhole principle) *There is no injection from the segment of numbers below $n+1$ to the segment of numbers below n.*

Proof. The proof is by contradiction using a straightforward induction on n. If $n = 0$ then the range type of the required injection is empty so there is no function to such a range type. Otherwise, if $n \neq 0$ and there is an injection f from the segment below $n+1$ to the segment below n, then we can construct an injection from the segment below n to the segment below $n-1$ as follows. Define a function g over the domain type of the segment below n so that $g(k)$ is $f(k)$ for $f(k) < f(n)$ and $f(k) - 1$ otherwise. The range type of g is the segment below $n-1$. The function g can be shown to also be an injection by the the case analysis used in its definition. This contradicts the induction hypothesis that there is no injection from the segment below n to the segment below $n-1$.

The primary creative input in the mechanical proof is the definition of g. Otherwise, the proof is a straightforward induction followed by an easy case analysis. This proof required 16 interactions.[5] $\square$

The following is yet another example of a seemingly obvious theorem that does not have a simple formal argument. Let $S(v)$ denote the set $\{x | x < M \wedge \exists i : i \neq 0 \wedge v(i) = x\}$.

Lemma 4.13 *The nonzero processes do not contain all the possible counter values. The formal statement of the theorem is $\exists x : x < M \wedge x \notin S(v)$.*

[5]The PVS proof should be compared with a similar proof carried out by Boyer and Moore [BM84] using the Boyer-Moore theorem prover. In the latter proof, the injection is represented as a list with some complicated conditions that constrain it to behave as an injection.

Proof. The intuitive reason is that there are at most $N-1$ nonzero processes and hence at most $N-1$ distinct counter values. If there are M possible counter values where $N \leq M$, then there must be some x below M such that $x \notin S(v)$.

A detailed description of the mechanized proof is as follows. First, there exists an injection f from $S(v)$ to the segment below $N-1$. Such a function f can be defined as the inverse of the mapping from i to $v(i+1)$ since $S(v)$ is the image set of such a mapping. Note that the inverse function f^{-1} is defined on the image of f so that $f^{-1}(j) = i$ for some i such that $f(i) = j$. An inverse of any mapping is injective on the image of the mapping.

If the conclusion of the theorem is false, then $S(v)$ coincides with the segment below M and the identity function I is an injection from the segment below M to $S(v)$. Since the composition of injections is an injection, we have an injection $f \circ I$ from the segment below M to the segment below $N-1$ and hence the segment below $M-1$. This contradicts the pigeonhole principle (Lemma 4.12).

The mechanical proof carries out the above construction but a large number of type correctness proof obligations are generated due to the sophisticated use of predicate subtyping. The proof has 51 interactions out of which all but 15 interactions deal with simple type correctness proof obligations. □

At this point we have established that there is a counter value that does not occur in the nonzero processes and we also know that through successive increments, process 0 can take on any counter value. The next step is to show (Claim 7) that process 0 can acquire a counter value that does not occur in any of the nonzero processes, and furthermore once this value is acquired, it is propagated along a growing prefix of processes until all processes have the same counter value (Claim 8). At this point, process 0 is the only enabled process and the system has entered a stable state.

Let $\mathit{prefix}(v)(j)$ be defined as $(\forall i : i \leq j \equiv v(i) = v(0))$. The first step is to show that process 0 can acquire a counter value that is different from the counter value of any other process. We know by Lemma 4.13 that there is a value x below M that does not occur as the counter value of any nonzero process. We know by Lemma 4.11 that process 0 does eventually take on value x in the nth state of the computation, for some n, but the lemma does not imply that no other process has value x at this point. We need to prove the statement that 0 acquires a counter value that is distinct from those of other processes by induction on the number of steps n given by Lemma 4.11.

Lemma 4.14 *In any run σ, eventually process 0 acquires a counter value that is distinct from those of the nonzero processes. Formally,*

$$\forall n, \sigma : \mathit{run}(\sigma) \wedge \sigma(n)(0) = x \wedge x \notin S(\sigma(0)) \supset (\exists n' : (\forall i : i = 0 \equiv \sigma(n')(i) = x)).$$

Proof. Note that the formal statement has an extra antecedent asserting that

$\sigma(n)(0) = x$ which we know can be discharged by Lemma 4.11, but is required here since the proof is by induction on n. In the base case when $n = 0$, the proof follows easily by letting n' be 0 since $\sigma(0)(0) = x$ but $x \notin S(\sigma(0))$. In the induction step, when $n > 0$ we instantiate σ in the induction hypothesis with $\sigma[1]$. This yields an induction hypothesis

$$\begin{aligned} &run(\sigma[1]) \wedge \sigma[1](n-1)(0) = x \wedge x \notin S(\sigma[1](0)) \\ &\quad \supset \quad (\exists n'' : (\forall i : i = 0 \equiv \sigma[1](n'')(i) = x)). \end{aligned}$$

If $x \in S(\sigma[1](0))$, then it must be the case $\sigma(0)(0) = x$ and the transition from $\sigma(0)$ to $\sigma(1)$ is on process 1. In this case, we let n' be 0 since $\sigma(0)(0) = x$ but $x \notin S(\sigma(0))$. Otherwise, when $x \notin S(\sigma[1])$, we obtain an n'' from the induction hypothesis so that $(\forall i : i = 0 \equiv \sigma[1](n'')(i) = x)$. We let n' be $n'' + 1$ to prove the required conclusion.

The corresponding PVS proof is constructed with 25 interactions. □

The next claim is needed in the induction step for showing that if we have a homogeneous prefix from 0 to j where $j < N - 1$, then it can be extended to a homogeneous prefix from 0 to $j + 1$. Note that the claim in the previous sentence is incorrect unless the definition of *prefix* ensures that the prefix value does not occur outside the prefix.

Lemma 4.15 *If process $j + 1$ is eventually updated in a run where there initially is a homogeneous prefix from 0 to j, a homogeneous prefix from 0 to $j + 1$ is eventually obtained. Formally,*

$$\begin{aligned} &\forall n, \sigma : run(\sigma) \wedge UP(j+1)(\sigma(n), \sigma(n+1)) \wedge prefix(\sigma(0))(j) \\ &\quad \supset \quad (\exists n' : prefix(\sigma(n'))(j+1)). \end{aligned}$$

Proof. This claim is proved by induction on n. When $n = 0$, $UP(j+1)(\sigma(0), \sigma(1))$ holds. If we take n' to be 1, then the conclusion follows by noting that $\sigma(1)(j+1) = \sigma(0)(j) = \sigma(0)(0)$ and for $i > j + 1$, $\sigma(1)(i) = \sigma(0)(i) \neq \sigma(0)(0)$. The quantifier instantiation in the proof of the base case is quite tricky and the mechanical proof requires six interactions.

In the induction step when $n > 0$, we have to show that the conclusion follows from the induction hypothesis

$$\begin{aligned} &\forall \sigma' : run(\sigma') \wedge UP(j+1)(\sigma'(n-1), \sigma'(n)) \wedge prefix(\sigma'(0))(j) \\ &\quad \supset \quad (\exists n'' : prefix(\sigma'(n'')(j+1))). \end{aligned}$$

We have by $run(\sigma)$ that there is some k such that $UP(k)(\sigma(0), \sigma(1))$ holds. This k cannot be in the subrange from 0 to j since these processes are not enabled in $\sigma(0)$. If $k = j + 1$, then we can repeat the reasoning carried out in the base case. If $k > j+1$, then we instantiate the induction hypothesis with $\sigma[1]$ for σ'. Clearly $run(\sigma[1])$ and $UP(j+1)(\sigma[1](n-1), \sigma[1](n))$ follow from the corresponding formulas in the induction conclusion. The antecedent of the induction hypothesis, $prefix(\sigma[1](0))(j)$, also follows from the corresponding formula since $\forall i : i < j \supset \sigma(1)(i) = \sigma(0)$. The induction hypothesis therefore yields an n'' such that $prefix(\sigma[1](n''))(j+1)$ holds. We can therefore prove the induction conclusion by instantiating n' with $n''+1$.

This entire proof is fairly sizable and requires 33 interactions. □

Lemma 4.16 *In any run, it is always possible to reach a state where the processes from* 0 *to* j *have the same counter value and this counter value does not occur in the processes numbered from* $i+1$ *to* $N-1$*. The formal statement is*

$$\forall j, \sigma : run(\sigma) \supset \exists n : prefix(\sigma(n))(j).$$

Proof. The proof is by induction on j. In the base case, when $j = 0$, we know by Lemma 4.13 that there is at least one x such that $x \notin S(\sigma(0))$. We know by Lemma 4.11 that there is an n'' where $\sigma(n'')(0) = x$. We can use Lemma 4.14 with n'' for n to get an n' such that $prefix(\sigma(n'))(0)$ holds.

Next we turn to the induction step when $j > 0$. The formula to be proved here is

$$run(\sigma) \wedge prefix(j-1)(\sigma(n)) \supset (\exists m : prefix(\sigma(m))(j)).$$

By instantiating σ in Lemma 4.9 with $\sigma[n]$ we get an n'' at which process j is updated and $UP(j)(\sigma[n](n''), \sigma[n](n''+1))$ holds. We can therefore apply Lemma 4.15 with $\sigma[n]$ for σ, n'' for n, and $j-1$ for j, to obtain an n' where $prefix(\sigma[n](n'))(j)$ holds. Thus, $n+n'$ serves a witness for m in our desired conclusion. □

At this point, we have all the ingredients needed to prove the self-stabilization property (Claim 9). Again, one might think that this is an obvious consequence of the claims presented thus far but the details are by no means straightforward.

Theorem 4.17 *In any run, there is a point beyond which the cardinality of the set of enabled processes is exactly one. The theorem is formally stated as*

$$\forall \sigma : run(\sigma) \supset (\exists m : \forall n : |E(\sigma[m](n))| = 1).$$

Proof. The first step in the proof is to invoke Theorem 4.16 with $N-1$ for j to obtain a state $\sigma(m)$ where the counter values of all the processes are identical. This leaves us with the task of proving that if $prefix(\sigma[m](0))(N-1)$ then $\forall n : |E(\sigma[m](n))| = 1$. This is proved by induction on n. In the base case, we first show that $E(\sigma[m](0)) = \{0\}$. This follows by applying extensionality and using the homogeneous prefix property twice: once on i to show that for any

i, if $i \in E(\sigma[m](0))$ then $i = 0$, and again on 0 to show that $0 \in E(\sigma[m](0))$. The second step is a trivial one of showing that the cardinality of the set $\{0\}$ is 1, but it requires the use of some lemmas concerning finite cardinalities. The mechanical verification of the base case takes ten interactions.

In the induction step, we need to show that the cardinality of enabled process remains 1 under a transition from $\sigma[m](n)$ to $\sigma[m](n+1)$. We know by Lemma 4.3 that $|E(\sigma[m](n+1))| \leq |E(\sigma[m](n))|$ which means $|E(\sigma[m](n+1))|$ is either 0 or 1. It cannot be 0, since this contradicts Lemma 4.1 which asserts that there is always at least one enabled process. Hence the conclusion. The induction step takes up about eleven interactions, and about eight more interactions are needed to discharge the type-correctness proof obligations that come up, particularly those that require the demonstration that certain intermediate set constructions are finite, i.e., possess a bijection to a finite segment of the natural numbers. $\square$

5 DISCUSSION

Related work. The study of self-stabilizing algorithms was initiated by Dijkstra in 1974 [Dij74]. He presented three algorithms for distributed N-process mutual exclusion arranged in a ring. These algorithms ensured that the system converged to stable behavior with at most one privileged or active process in any state. The study of self-stabilizing algorithms was dormant for a while following Dijkstra's work but there has been a recent flurry of activity in the subject. Schneider [Sch93] presents a survey of the issues in the design of self-stabilizing algorithms. Arora and Gouda [AG93] describe a uniform formal framework for verifying fault-tolerance and self-stability. Varghese [Var92] presents various systematic techniques for designing fault-tolerant algorithms including one based on the algorithm studied in this paper. He also provides an informal proof sketch for the correctness of this algorithm that is loosely similar to the one described here.

Prasetya [Pra97] verifies a self-stabilizing minimum-cost routing algorithm in a variant of the UNITY logic [CM88] that is formalized in the HOL proof checking system [GM93]. He presents an elegant development of the theory needed to verify this algorithm but reports a prohibitively high level of verification effort. His verification needed 8900 lines of specification and proof for the basic theories, 9300 lines for formalizing various aspects of the UNITY logic, and 5500 lines for the verification itself. Even though his verification takes some implementation details such as the communication mechanisms into account, the algorithm he verifies is roughly comparable in complexity to the one verified here. Since Prasetya's verification differs from ours in style and content, a sensible comparison of the two proof efforts is not viable. However, in the following discussion, we would like to dispel the impression that the mechanical verification of distributed algorithms, particularly self-stabilizing ones, necessarily requires an unreasonable amount of time and effort.

The challenge of formalization. Having struggled with the formalization of this self-stabilization problem, we certainly do not feel that formalization is a trivial matter. It took us quite a bit of effort to resolve even simple questions such as how to state the pigeonhole principle in its most relevant and usable form, how to formulate the measure function, and when to use measure induction rather than ordinary induction. The proof effort is extremely sensitive to the exact form of the formalization. It took us several false starts and not-so-near misses before we finally arrived at the formalization presented here. The construction of informal proof outline given by the claims in Section 2 was guided by insights gained from the failed attempts but was written without the aid of mechanization.

The final formalization does not dilute the intuitive clarity of the informal presentation. We have given a detailed informal outline of the formal steps in the mechanical verification. Nothing in this outline runs counter to informal intuitions, and indeed the formal proof is a faithful elaboration of these informal intuitions. Several serious gaps in the informal proof sketch were addressed in the formalization without loss of intuitive clarity. It is, of course, quite easy to construct informal or formal definitions and proofs that are completely obscure. However, it is not the case that the clarity of an informal presentation must be lost in formalization.

The formal script is not unreasonably long. The entire specification is fewer than 200 lines including generous amounts of white space and blank lines. The proof script consisting of all of the interactive commands for all of the proofs, including those of the type-correctness proof obligations that were generated and proved automatically, is fewer than 600 lines when pretty-printed with lots of white space and blank lines. This level of conciseness and faithfulness to the informal outline would be impossible without the use of the automation that is available in PVS in terms of typechecking, proof obligation generation, rewriting, decision procedures, and proof strategies. Even this proof script could be made more robust and concise through modest improvements to the automation that is available in PVS.

The mechanization was first fully completed with about four to six days of effort, and another two days were spent streamlining some of the proofs. Automated tools provide quite a bit of help in getting the details right by highlighting both syntactic and semantic errors. In the case of PVS, we use it to actively explore and discover proofs and not merely check existing proofs. This activity is quite enjoyable. The parts that are routine and mechanical are usually handled by means of the automation. The construction of a crisp and taut formal argument requires a number of mental leaps that are essentially similar to the creative effort needed to construct a convincing informal proof. In making these leaps, the mechanization helps construct explanations why some of these leaps are problematic and how they can be refined.

The main conclusion, one that has been observed earlier by de Bruijn [dB80] and others, is that the machine-checked proofs are of manageable size and linearly track the informal argument, and with modern automation, the expansion factor can actually be made quite small.

The challenge of mechanical verification. There is a widespread belief that mechanized deductive verification is too hard. Even those who advocate formal techniques are skeptical about the value of mechanical verification. The primary objection is that the achievable level of automation is inadequate for productive proof construction. In the case of the self-stabilization proof, even with the extensive automated support in PVS, we were repeatedly confronted with subgoals that were obvious but where the automation in PVS was unable to expand just the right definitions or supply the right instantiations for quantified variables. Through experiments such as these, we are constantly learning about the problems and drawbacks of the current automated support. There is no reason why the automation cannot be improved to a point where most obvious subgoals are indeed proved automatically.

The existing state of automated support is not an indication of what is in fact possible. To quote Dana Scott, *"formalization is an experimental science"* and the same holds for mechanization. Current automated tools are already being used quite effectively by a large body of users. The progress so far in understanding the tradeoffs between automation and interaction has been quite encouraging.

Is self-stability hard to verify? Prasetya [Pra97] claims the verification of self-stability to be especially hard. We can only speak of this particular experience. The main challenge for us was in coming up with a clean and crisp informal proof sketch. The mechanization based on this proof sketch was certainly not a mere formality, but there is no reason to believe that there is anything especially hard about deductively verifying self-stability. Self-stability is a global progress property of a system and requires reasoning about global progress measures which makes it different from the usual local progress properties that are proved for individual processes, but the techniques used are not that dissimilar.

How much of the proof was conventional mathematical reasoning? We would estimate that more than half the verification involved conventional mathematical reasoning about sets, cardinalities, modulo arithmetic, injective maps, pigeonhole principle, and well-founded measure functions. There were only four truly temporal proofs and even some of these involved a fair amount of combinatorial reasoning.

Would a special-purpose temporal formalism have helped? Although we initially felt that this proof would not have derived a significant benefit from a special-purpose temporal formalism [MP92, CM88] because large parts of the proof involved conventional mathematical reasoning, we did find some situations where a temporal framework would have been handy. After we completed our first pass at the verification, we observed that all the temporal properties following Lemma 4.9 employed instances of a single temporal rule, a generalization of the `WHILE` rule, which allows $\Diamond(p \wedge q)$ to be derived from $\Diamond q$ and $p \wedge \Box(p \wedge \neg q \supset \bigcirc p)$. We have not tried out the proof with this proof rule, but believe that the ability to identify and employ such powerful and generic proof rules provides sufficient methodological justification for embedding a temporal framework within the general-purpose one used here. Temporal logic can be quite easily embedded in the higher-order logic of PVS.

Could model checking be used? Model checking is often the easiest way to automatically verify a distributed algorithm. PVS has support for model checking but we were unable to reduce the problem here to one that could be model checked. While there are some finite-state techniques for verifying parametric systems of the form shown here [CGL94, WL89], they do not appear to be useful for this example. We were also unable to reduce the other two protocols presented by Dijkstra [Dij74] to model checkable form even though these are systems composed only of finite-state processes.

We pose this as an open problem: *Find a way to combine deduction and model checking to verify this and other self-stabilization properties/algorithms, or show that such a reduction is impossible?*

6 CONCLUSION

We have presented a proof of one of Dijkstra's self-stabilizing mutual exclusion algorithms that has been verified using PVS. We started with an informal proof sketch and constructed a formalization and mechanical verification that preserves the structure and enhances the clarity of the original proof sketch. This serves as a useful existence proof that the challenges of formalization and mechanization are not unreasonably daunting. Dijkstra's algorithm is particularly interesting since its justification employs a nontrivial amount of combinatorial mathematics. This kind of mathematics has typically been thought to be easier to present informally than formally. We have shown that this is not the case and that the formalization can closely follow the informal development. We have also shown exactly where the formal development loses succinctness when compared with the informal one. Such an analysis is useful in developing automated decision procedures and strategies that can more effectively close the informal/formal gap.

Acknowledgment. This work was supported by the Air Force Office of Scientific Research under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the National Science Foundation, or the U.S. Government. We are grateful to John Rushby for his encouragement, advice, and careful reading of drafts of this paper, to Sam Owre for help with PVS and LaTeX, and to Nikolaj Bjørner and the anonymous reviewers for their insightful comments and suggestions.

REFERENCES

[AG93] A. Arora and M. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, November 1993.

[BM84] R. S. Boyer and J. S. Moore. Proof checking the RSA public key encryption algorithm. *American Mathematical Monthly*, 91(3):181–189, 1984.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 589–606. Academic Press, New York, NY, 1980.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[Dij86] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1:5–6, 1986.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification*. Springer-Verlag, New York, NY, 1992.

[ORS92] 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, June 1992. Springer-Verlag.

[Pra97] I. S. W. B. Prasetya. Mechanically verified self-stabilizing hierarchical algorithms. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 399–415, Enschede, The Netherlands, April 1997. Springer-Verlag.

[Sch93] M. Schneider. Self stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[Var92] G. Varghese. *Self-Stabilization by Local Checking and Correction*. PhD thesis, MIT, 1992.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 68–80, Grenoble, France, 1989. Springer Verlag.

Shaz Qadeer is a graduate student in Electrical Engineering and Computer Sciences at the University of California at Berkeley. His research interests are formal methods applied to verification and synthesis of concurrent hardware and software systems. He graduated with a B.Tech. in Electrical Engineering and received the President's Gold Medal from the Indian Institute of Technology, Kanpur in 1994.

He received an M.S. in Electrical Engineering and Computer Sciences from the University of California at Berkeley in 1997.

Natarajan Shankar is a computer scientist at the SRI International computer science laboratory in Menlo Park. His main research interests are in formal methods and automated deduction. He graduated with a B.Tech. in Electrical Engineering from IIT Madras in 1980, and a Ph.D. in Computer Science from the University of Texas at Austin in 1986.