A Case-Study in Component-Based Mechanical Verification of Fault-Tolerant Programs[1]

To appear in Fourth Workshop on Self-Stabilization (WSS'99) Austin, Texas, USA

Sandeep S. Kulkarni
Department of Computer and Information Science
The Ohio State University
Columbus Ohio 43210 USA

John Rushby Natarajan Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025
USA

Abstract

In this paper, we present a case study to demonstrate that the decomposition of a fault-tolerant program into its components is useful in its mechanical verification. More specifically, we discuss our experience in using the theorem prover PVS to verify Dijkstra's token ring program in a component-based manner. We also demonstrate the advantages of component based mechanical verification.

Keywords : Component-based verification, Fault-tolerance, Program decomposition, Mechanical verification, Self-stabilization

1 Introduction

In this paper, we argue that the decomposition of a fault-tolerant program into its components is beneficial in its mechanical verification, and that such a decomposition admits reuse of the proofs for other fault-tolerant programs as well as the variations of the given fault-tolerant program.

Arora and Kulkarni [3] have shown that a fault-tolerant program can be decomposed into a fault-intolerant program and a set of 'tolerance'-components, namely *detectors* and *correctors*. Intuitively, a detector is a component that 'detects' whether a given predicate is true in the program state, and it is used for ensuring that the program satisfies its safety specification in the presence of faults. Likewise, a corrector is a component that 'corrects' the system to a state where the given state predicate is true, and it is used for ensuring that the program eventually recovers to a state from where its specification is satisfied.

For example, a fail-safe program, which satisfies only its safety specification in the presence of faults, can be decomposed into a fault-intolerant program and detector(s). Likewise, a self-stabilizing program, which guarantees recovery to a state from where its specification is satisfied, can be decomposed into a fault-intolerant program and corrector(s).

Decomposition of a fault-tolerant program permits the verification of a given property by focusing on the component that is responsible for satisfying it. For example, if we need to show that a program eventually recovers to a state from where it satisfies its specification, we should focus on its corrector components. Likewise, if we are interested in showing that the program satisfies its specification in the absence of faults, we should focus on the corresponding fault-intolerant program. Of course, we will have to show that other components of the program do not interfere with the component of interest. But this proof is typically simpler than the proof required to show that the overall program satisfies the given property. Moreover, if we change some components used in that program, the proofs of other components are not affected. Thus, it is possible for a small change in the program to lead to a small change in the proof.

With the motivation of developing a systematic approach for mechanical verification using program decomposition, we are implementing the theory of detectors and correctors into the theorem prover PVS [11][2]. In this paper, we present a proof of one of Dijkstra's token ring program [5] that has been proved using this theory. Previously, Qadeer and Shankar [13] have verified this token ring program using PVS. While their proof is impressive, it is very specific to one program and, hence, much of their proof-technique cannot be reused to prove other fault-tolerant programs. Moreover, since they focus on the entire program, instead of its components, their proof is more complex than it needs to be. We use this case-study to illustrate how the decomposition of the program into its components can help in making the proofs simple and reusable.

Being self-stabilizing, Dijkstra's program can be decomposed into a fault-intolerant program and a corrector. The fault-intolerant program circulates the token along an initialized ring in the absence of faults. On the other hand, if faults perturb the program from its ideal states, the corrector restores the fault-intolerant program back to some ideal state, from where it continues to circulate the token. This program is self-stabilizing in that even if the faults perturb the program to an arbitrary state, the corrector restores it to

[1]Email: kulkarni@cis.ohio-state.edu, rushby@csl.sri.com, shankar@csl.sri.com Web: `http://www.cis.ohio-state.edu/~kulkarni`, `http://www.csl.sri.com/~rushby`, `http://www.csl.sri.com/~shankar`
This work was partially supported by National Science Foundation grant CCR-9509931 and by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044.

[2]The URL `http://www.cis.ohio-state.edu/~kulkarni/pvs/` contains the PVS specification and proofs.

an ideal state.

In Dijkstra's token ring program, processes $0..N$, $N \geq 1$, are organized in a ring. Each process j maintains a counter $x.j$, $0 \leq x.j < M$ for some $M > 1$. A non-zero process j has a token iff $x.j$ differs from $x.(j-1)$, and process 0 has a token iff $x.0$ is the same as $x.N$. If process j has a token then it passes it to process $j+1 \ mod \ N+1$ by setting $x.j$ to $x.(j-1)$, and if process 0 has a token then it passes it to process 1 by incrementing $x.0$. For any M, $M > 1$, the program guarantees that in the absence of faults there will be exactly one token that is being circulated in the ring. If $M \geq N+1$, the program guarantees that starting from any arbitrary state, the program will reach a state where there is exactly one token which is circulated along the ring.

To decompose Dijkstra's program into a fault-intolerant program and a corrector, we first consider the following question: If we are only interested in a token circulation along an initialized ring, how can the token ring program be simplified? The answer to this question identifies the fault-intolerant program. Next we ask the question about fault-tolerance: what are the ideal states of the resulting fault-intolerant program, and how can it be recovered to these ideal states if the faults perturb it? The answer to this question identifies the corrector. Then, we show how the fault-intolerant program and the corrector can be independently verified in PVS and how they can be shown to be interference-free.

The rest of the paper is organized as follows: In Section 2, we present Dijkstra's token ring program and its decomposition into a fault-intolerant program and a corrector. In Section 3, we show how the token ring program is modeled in PVS. In Section 4 and 5, we present the correctness proof for the fault-intolerant program and the corrector respectively. In Section 6, we show that the corrector and the fault-intolerant program do not interfere with each other. Finally, in Section 7, we discuss the advantages of component-based verification over non-component-based verification, and present concluding remarks in Section 8.

2 The Token Ring Program and its Decomposition

In this section, we first present the decomposition of Dijkstra's token ring program into a fault-intolerant program and a corrector. Then, we argue that they work in isolation and that they do not interfere with each other. We use the same arguments for mechanical verification in Sections 4, 5 and 6.

Fault-intolerant program. If we are not interested in fault-tolerance, a token ring program can be designed by maintaining a variable $x.j$ (in the range 0..(M-1), where $M > 1$) as follows: Each non-zero process j checks whether $x.j$ is different from $x.(j-1)$. If this condition is true then $x.j$ is set to $x.(j-1)$. Process 0 checks whether $x.0$ is the same as $x.N$. If this condition is true, process 0 increments $x.0$. Thus, the actions of the fault-intolerant program are as follows:

$$\begin{array}{lll} j \neq 0 \ \wedge \ x.j \neq x.(j-1) & \longrightarrow & x.j := x.(j-1) \\ x.0 = x.N & \longrightarrow & x.0 := x.0 + 1 \end{array}$$

The invariant of this program is S, where

$$\begin{array}{rl} S \ = & (\exists j, v : 0 \leq j \leq N, 0 \leq v < M \ : \\ & \quad (\forall k : 0 \leq k < j : x.k = v) \quad \wedge \\ & \quad (\forall k : j \leq k \leq N : x.k = v-1 \ mod \ M)) \end{array}$$

The invariant S characterizes the states where there exists a process j such that the x values of processes $0..(j-1)$ are equal to v, and the x values of processes $j..N$ are equal to $v-1 \ mod \ M$. Thus, process j has the unique token, and only the action at j is enabled in that state. The execution of this action results in a state where process $j+1 \ mod \ N+1$ has the token. Thus, starting from a state where S is true, the fault-intolerant program circulates a unique token along the ring.

Corrector. If the faults perturb the x values maintained at the processes, we need to recover the program to a state where S holds in order to ensure that the token circulation is re-established. This can be achieved by the corrector that lets each non-zero process copy the x value of its predecessor. Thus, the actions of the corrector are as follows:

$$j \neq 0 \ \wedge \ x.j \neq x.(j-1) \qquad \longrightarrow \qquad x.j := x.(j-1)$$

Observe that if the corrector actions execute in isolation, a state is reached where all x values are same, and at that state S is true. Also, if the corrector executes in any state where S is true, S continues to be true in the resulting state.

Note that although the actions of the fault-tolerant program and that of the fault-intolerant program are the same, when dealing with the fault-intolerant program we can assume that the invariant S is true. In this sense, the fault-intolerant program is simpler than the fault-tolerant program. Of course, the actions of the corrector are a subset of the fault-tolerant program and, hence, the corrector is simpler than the fault-tolerant program.

Interference-freedom between the fault-intolerant program and the corrector. Since the corrector is a subset of the fault-intolerant program, it is trivial that the corrector does not interfere with the fault-intolerant program. Likewise, the actions of the fault-intolerant program at non-zero processes are a subset of the corrector and, hence, do not interfere with the corrector. Thus, we only need to show that the action at process 0 does not interfere with the corrector. We prove the interference-freedom as follows:

1. For process 0 to interfere with the corrector, it must execute infinitely often. Otherwise, after 0 stops executing, convergence to S will be achieved.

2. If the action at process 0 executes infinitely often, $x.0$ will take all possible values in the range $0..(M-1)$.

3. If the domain of x is large enough, specifically $M \geq N+1$, then in the initial state, there must be a value in the range $0..(M-1)$ which is not present at any non-zero process.

4. From 2 and 3, it follows that eventually $x.0$ will obtain a value missing in the initial state.

5. After $x.0$ is equal to this missing value, process j will obtain this missing value only after processes $0..(j-1)$ obtain this missing value. Thus, when process 0 executes next (from 1, we know that process 0 will execute next), all processes will have the same x value. Thus, a state where S is true is reached.

3 Modeling of the Token Ring in PVS

In this section, we discuss how we modeled Dijkstra's token ring program in PVS. More specifically, we first define program independent concepts such as states, state predicates, actions, program compositions, etc. Then, we define the actions of the token ring program and its invariant.

State. The state of the program consists of the x values at processes $0..N$, each x value is in the range $0..(M-1)$.

Trace. A trace is an infinite sequence of states. If seq is a trace and i is a natural number then $seq(i)$ denotes the i^{th} element in seq.

Assertion. An assertion is a predicate over states. If P is an assertion and s is a state then $P(s)$ denotes whether P is true in state s.

Action. An action is a relation over states. If A is an action and $s1, s2$ are states then $A(s1, s2)$ denotes whether state $s2$ can be reached by executing A in state $s1$.

Property. A property is a predicate over traces. If R is a property and seq is a trace then $R(seq)$ denotes whether the property R is true of seq.

Notation. Henceforth, we use p and q to denote programs; $s, s0, s1$ and $s2$ to denote program states; seq to denote a trace; S and T to denote assertions; R to denote a property; m, n to denote natural numbers; j, k to denote processes; and $v, v1, v2$ to denote the x values at processes. Moreover, given a state s, $x(s)(j)$ denotes the value of $x.j$ in state s.

Program compositions. In the base case, a program is just a single action. The parallel composition of programs p and q, denoted as $p[\!]q$, is a program consisting of the actions of p and the actions of q. (While we have defined other program compositions used for fault-tolerant programs, we omit them here as they are not used in Dijkstra's token ring program.)

CanExecute. Program p can execute in state $s1$ iff there exists a state $s2$ such that $p(s1, s2)$ is true.

$$CanExecute(p)(s1) \quad = \quad (\exists s2 :: p(s1, s2))$$

Next. The predicate $Next(p)(s1, s2)$ denotes whether state $s2$ can be reached by execution of some action of p. If no action of p is enabled in state $s1$ then $Next(p)(s1, s2)$ is true iff $s1 = s2$.

$$\begin{aligned} Next(p)(s1, s2) \quad =& \\ & (CanExecute(p)(s1) \ \wedge \ p(s1, s2)) \\ \vee\ & (\neg CanExecute(p)(s1) \ \wedge \ s1 = s2) \end{aligned}$$

Computation. A computation of a program p is a trace $s_0, s_1, ...$ such that for each n, $Next(p)(s_n, s_{n+1})$ is true. Thus, the predicate characterizing 'seq is a computation of program p' is represented as follows:

$$run(p)(seq) \quad = \quad \forall n :: Next(p)(seq(n), seq(n+1))$$

Satisfies. Program p satisfies a property R iff for every trace that is a computation of p, $R(seq)$ is true. Thus, $satisfies(p)(R)$ is defined as follows:

$$satisfies(p)(R) \quad = \quad \forall seq :: run(p)(seq) \Rightarrow R(seq)$$

We use two types of properties in the proof of Dijkstra's token ring program, closure and convergence.

Closure. The property $closed(S)$ is the set of all traces $s_0, s_1, ...$ where for each $n, n \geq 0$, if S is true at s_n then S is true s_{n+1}. Thus, $closed(S)$ is defined as follows:

$$closed(S)(seq) \quad = \quad \forall n :: S(seq(n)) \Rightarrow S(seq(n+1)))$$

Convergence. The property $converges(T, S)$ is the set of all traces $s_0, s_1, ...$ where $closed(S)$ and $closed(T)$ are true, and if there exists $n, n \geq 0$, for which T is true at s_n then there exists m, $m \geq n$, for which S is true at s_m. Thus, $converges(T, S)$ is defined as follows:

$$\begin{aligned} & converges(T, S)(seq) \quad = \\ & \quad closed(T)(seq) \ \wedge \ closed(S)(seq) \ \wedge \\ & \quad \forall n :: T(seq(n)) \ \Rightarrow \ (\exists m : m \geq n : S(seq(m))) \end{aligned}$$

Num_steps. Given an action ac, a trace seq, and a natural number n, the number of times action ac is executed until the n^{th} state is defined as follows:

$$\begin{aligned} & num\_steps(ac)(seq, n) \quad = \\ & \begin{array}{ll} 0 & \text{if } n = 0 \\ num\_steps(ac)(seq, n-1) + 1 & \text{if } ac(seq(n-1), seq(n)) \\ num\_steps(ac)(seq, n-1) & \text{otherwise} \end{array} \end{aligned}$$

Corrector. The corrector action at a non-zero process j is executed only in states where $x.j$ differs from $x.(j-1)$. The execution of this action results in a state where $x.j$ has the same value as that of $x.(j-1)$ and the other x values remain unchanged. Thus, corrector action at j is defined as follows:

$$\begin{aligned} corr(j)(s0, s1) \quad = \quad & x(s0)(j) \neq x(s0)(j-1) \quad \wedge \\ & x(s1)(j) = x(s0)(j-1) \quad \wedge \\ & \forall k : k \neq j : x(s1)(k) = x(s0)(k) \end{aligned}$$

The corrector consists of the actions at all non-zero processes. We, therefore, use parallel composition of $corr(j)$, $0 < j \leq N$, to define the corrector, $corr\_prog$, as follows :

$$corr\_prog \quad = \quad ([\!] j : j \neq 0 : corr(j))$$

Action at process 0. The action at process 0 is executed only in states where $x.0$ is the same as $x.N$. The execution of this action results in a state where the value of $x.0$ is one greater than its initial value (in mod M arithmetic) and the other x values remain unchanged. Thus, the action at process 0 is defined as follows:

$$\begin{aligned} & action\_zero(s0, s1) \quad = \\ & \quad x(s0)(0) = x(s0)(N) \ \wedge \\ & \quad x(s1)(0) = x(s0)(0) + 1 \ mod\ M \ \wedge \\ & \quad \forall j : j \neq 0 : x(s1)(j) = x(s0)(j) \end{aligned}$$

Note that the fault-intolerant program consists of the parallel composition of the action at process 0 and the corrector. Thus, the fault-intolerant program is $action\_zero[\!]corr\_prog$.

j has a token. We define the predicate, 'j has a token in state s' as follows:

$$\begin{aligned} token(s)(j) \quad = \quad & (j = 0 \ \wedge \ x(s)(0) = x(s)(N)) \\ \vee\ & (j \neq 0 \ \wedge \ x(s)(j) \neq x(s)(j-1)) \end{aligned}$$

Invariant of the fault-intolerant program. Finally, we define the invariant of the fault-intolerant program, $corr\_pred$, as follows:

$$
\begin{array}{lll}
corr\_pred(s) & = & \\
\quad (\exists j, v :: & \forall k : k < j : x(s)(k) = v & \wedge \\
& \forall k : k \geq j : x(s)(k) = v - 1\ mod\ M) &
\end{array}
$$

Remark. Although in this presentation, we have given a specific instantiation for the program state, it is initially defined as an uninterpreted type, and then instantiated suitably for the token ring program. This allows program independent concepts such as traces, assertions to be reused for different programs.

4 Verification of the Fault-Intolerant Program

To prove the correctness of the fault-intolerant program, we need to show (1) $corr\_pred$ is closed in the fault-intolerant program, and (2) if the token is at process j and an action of the fault-intolerant program is executed then the token is at process $j+1\ mod\ N+1$ in the resulting state.

In Theorem 4.3, we show that $corr\_pred$ is closed in the fault-intolerant program. In this proof, we use Lemmas 4.1 and 4.2 which show that $corr\_pred$ is closed in the action of process 0 and the actions of non-zero process respectively. Finally, we show the token circulation property in Theorem 4.5.

Lemma 4.1 In the computation of $action\_zero$ alone, $corr\_pred$ is closed. Formally,

$$satisfies(action\_zero)(closed(corr\_pred))$$

Proof. After eliminating the quantifiers and expanding the definitions, we need to show that if $corr\_pred$ is true in the n^{th} state of the computation then it is true in $(n+1)^{th}$ state of that computation. To this end, we first do a case split on the process that has the token in the n^{th} state: In the case, where process 0 has the token, i.e., the x values of processes $0..N$ are $v-1\ mod\ M$ for some v, we show that execution of $action\_zero$ results in a state where the x values of $1..N$ remain v–$1\ mod\ M$ and the x value of 0 is v, i.e., $corr\_pred$ is true. In the case, where process j, $j \neq 0$, has the token, we show that $x.0$ is v and $x.N$ is $v-1\ mod\ M$ for some v and, hence, $action\_zero$ is disabled, i.e., the $(n+1)^{th}$ state is identical to the n^{th} state and, hence, $corr\_pred$ is true in $(n+1)^{th}$ state. □

Lemma 4.2 In the computation of the action at a non-zero process, $corr\_pred$ is closed. Formally,

$$\forall j :: satisfies(corr(j))(closed(corr\_pred))$$

Proof. The proof of this lemma is similar to that of Lemma 4.1. We show that if process j has the token then in the resulting state process $j+1\ mod\ N+1$ has the token, and if any other process has the token, the execution results in a stuttering. In either case, $corr\_pred$ is true. □

Theorem 4.3 In the computation of the fault-intolerant program, $corr\_pred$ is closed. Formally,

$$satisfies(action\_zero[\!]corr\_prog)(closed(corr\_pred))$$

Proof. This lemma is proved by using Lemmas 4.1 and 4.2 and the following property about parallel composition: if an assertion S is closed in programs p and q then it is closed in $p[\!]q$. □

Lemma 4.4. At least one action of the fault-intolerant program is enabled in any program state. Formally,

$$\forall s :: CanExecute(action\_zero[\!]corr\_prog)(s)$$

Proof. We prove this lemma by first doing a case-split on whether all x values are equal. If all x values are equal, it follows that $x.0 = x.N$ and, hence, $action\_zero$ is enabled. If all x values are not equal, we induct on the processes to find the first process, say j, such that $x.j$ differs from $x.0$. Since $x.(j-1) = x.0$ and $x.j \neq x.0$, it follows that process j is enabled. □

Theorem 4.5 Starting from a state where $corr\_pred$ is true, if the token is at process j then the execution of an action of the fault-intolerant program results in a state where the token is at process $j+1\ mod\ N+1$. Formally,

$$
\begin{array}{lll}
\forall j :: & & token(s1)(j) \\
& \wedge & corr\_pred(s1) \\
& \wedge & Next(action\_zero[\!]corr\_prog)(s1, s2) \\
\Rightarrow & & \\
& & (j \neq N \Rightarrow token(s2)(j+1)) \\
& \wedge & (j = N \Rightarrow token(s2)(0))
\end{array}
$$

Proof. Lemma 4.4 shows that execution of the fault-tolerant program does not result in stuttering. We then show that if process j has the token no other process is enabled. Finally, we show, as in Lemmas 4.1 and 4.2, that the execution of the action at j results in a state where $j+1\ mod\ N+1$ has the token. □

5 Verification of the Corrector

To prove that $corr\_prog$ satisfies its specification, we need to show (1) $corr\_pred$ is closed in $corr\_prog$, and (2) starting from any state, in the execution of $corr\_prog$ alone a state is reached where $corr\_pred$ is true. Note that (1) follows from Lemma 4.2. In this section, we prove that the corrector satisfies property (2) based on the following observation:

Observation. If only the actions at processes $j..N$ execute in a computation then eventually the x values of processes $j-1..N$ will be identical and the actions of processes $j..N$ will be disabled. □

If $j = 1$, the actions at processes $j..N$ are the same as the actions of the corrector and, hence, in the execution of the corrector, eventually the x values of all processes will be identical. Thus, convergence to $corr\_pred$ is achieved.

In order to obtain the proof of the above observation in PVS, we first define the program consisting of actions of processes $j..N$ and an assertion characterizing the states where the x values of processes $j..N$ are equal. Then, we provide a proof of the above observation in Lemma 5.2. Finally, we prove the convergence property in Theorem 5.3.

We define $corr\_above(j)$, and $same\_as\_N(j)$ as follows:

Definition $corr\_above(j)$**.** For any j, $j \neq 0$, $corr\_above(j)$ is the program consisting of the actions at processes $j..N$. Formally,

$$corr\_above(j) = ([\!]k : j \leq k \leq N : corr(k)) \quad □$$

Definition $same\_as\_N(j)$**.** For any j, $same\_as\_N(j)$ is an assertion which is true in state s iff the x values of processes $j..N$ are identical in state s. Formally,

$$same\_as\_N(j)(s) = \forall k : j \leq k \leq N : x(s)(k) = x(s)(N) \quad \square$$

Lemma 5.1 If $x.j$ is the same as $x.(j-1)$ in some state in the computation of $corr\_above(j)$, then this condition continues to be true in the rest of the computation. Formally,

$$\forall seq, j, n : j \neq 0 : \; run(corr\_above(j))(seq) \wedge x(seq(n))(j-1) = x(seq(n))(j)) \Rightarrow \forall m : m \geq n : x(seq(m))(j-1) = x(seq(m))(j)$$

$\square$

Lemma 5.2 In the computation of the corrector actions at processes $j..N$, a state is reached where the x values of processes $(j-1)..N$ are identical. Formally,

$$\forall seq, j :: \; run(corr\_above(j))(seq) \Rightarrow \exists n :: same\_as\_N(j-1)(seq(n)))$$

Proof. We prove this lemma by measure-induction on the j, where the measure used is $N-j$. In the base case, $j=N$, we do a case split on whether $corr(N)$ is enabled in the initial state: If $corr(N)$ is enabled, we show that in the successor state, $same\_as\_N(N-1)$ is true. If $corr(N)$ is disabled, we show that in the initial state $same\_as\_N(N–1)$ is true.

In the induction case, we do a case-split on whether the action at process j executes in the computation of $corr\_above(j)$. If j executes in the n^{th} state, we show that the suffix of the computation from the $(n+1)^{th}$ state is a computation of $corr\_above(j+1)$. Therefore, there exists a state, say s, where $same\_as\_N(j)$ is true. Moreover, since the values of $x.j$ and $x.(j–1)$ are equal in the $(n+1)^{th}$ state, by Lemma 5.1, it follows that the values of $x.j$ and $x.(j–1)$ are equal in state s. Thus, $same\_as\_N(j-1)$ is true in state s.

If j never executes in the computation of $corr\_above(j)$, we show that that computation is also a computation of $corr\_above(j+1)$. Therefore, there exists a state, say s, in this computation where $same\_as\_N(j)$ is true. Thus, the actions of j+1..N are disabled in state s. Since the program tries to execute an action unless all its actions are disabled, it follows that the action at j must also be disabled. It follows that $same\_as\_N(j-1)$ is true in s. $\square$

Theorem 5.3 The computation of the corrector eventually converges to $corr\_pred$. Formally,

$$satisfies(corr\_prog)(converges(true, corr\_pred))$$

Proof. We prove this lemma by instantiating $j = 1$ in Lemma 5.2. From Lemma 5.2, it follows that in the computation of the corrector alone, eventually a state is reached where all x values are identical and, hence, $corr\_pred$ is true in that state. Moreover, the closure of $corr\_pred$ follows from Lemma 4.2. $\square$

6 Interference-Freedom Between the Corrector and the Fault-Intolerant Program

After we showed that the fault-intolerant program and the corrector satisfy their specification in isolation, we proceed to show that they do not interfere with each other. As mentioned in Section 2, towards this end, we show that the action at process 0 does not interfere with the corrector. Our proof follows the outline discussed in Section 2. More specifically, in Lemma 6.1, we show that process 0 executes infinitely often. Then, in Lemma 6.4, we show that there exists a value that is different from the x values of all non-zero processes. Subsequently, in Lemma 6.6, we show that eventually process 0 gets this missing value, and in Theorem 6.8, we conclude that the action at process 0 does not interfere with the corrector.

Lemma 6.1 In the computation of the corrector and the action at process 0, either process 0 executes infinitely often or a state is reached where $corr\_pred$ is true. Formally,

$$\forall seq, n :: \; run(action\_zero [\!] corr\_prog)(seq) \Rightarrow \forall m :: (\exists n : n \geq m : action\_zero(seq(n), seq(n+1))) \vee \exists m :: corr\_pred(seq(m))$$

Proof. Note that process 0 executes infinitely often iff given any number m, it executes in the n^{th} state for some $n \geq m$. Thus, to prove this lemma we need to show that either (1) there exists a number n, $n \geq m$, such that $action\_zero$ executes in the n^{th} state, or (2) there exists a state where $corr\_pred$ is true. We prove this lemma by a case-split on whether the suffix of the computation starting from m^{th} state is a computation of $corr\_prog$. If that suffix is a computation of $corr\_prog$, by Theorem 5.3, it is straightforward to show that (2) is true. If the suffix is not a computation of $corr\_prog$, there exists a number n, $n \geq m$, such that the $(n+1)^{th}$ state is not obtained by executing $corr\_prog$ in the n^{th} state. Since in the n^{th} state either $action\_zero$ executes or $corr\_prog$ executes, it follows that in the n^{th} state $action\_zero$ executes, i.e., (1) is true. $\square$

Lemma 6.2 In the computation of the corrector and the action at process 0, the value of $x.0$ in the n^{th} state of the computation is equal to the sum of the initial value of $x.0$ and the number of steps taken by process 0. Formally,

$$\forall seq :: \; run(action\_zero [\!] corr\_prog)(seq) \Rightarrow x(seq(n))(0) = (x(seq(0))(0) + num\_steps(action\_zero)(seq, n)) \; mod \; M$$

Proof. We prove this lemma by induction on the length of the computation. In the initial state, this condition is trivially satisfied. In the induction case, we do a case-split on whether process 0 executes or whether $corr\_prog$ executes. In each case, the proof is straightforward. $\square$

Lemma 6.3 In the computation of the corrector and the action at process 0, either $x.0$ takes on all possible values in the range $0..(M-1)$ or a state is reached where $corr\_pred$ is true. Formally,

$$\forall seq :: run(action\_zero [\!] corr\_prog)(seq) \Rightarrow \forall v : 0 \leq v < M : (\exists n :: x(seq(n))(0) = v) \vee \exists n :: corr\_pred(seq(n))$$

Proof. We prove this lemma by using Lemmas 6.1 and 6.2 . In Lemma 6.2, if the value of $x.0$ in the initial state is $v0$ then after process 0 executes $(v - v0) \; mod \; M$ steps, the value of $x.0$ will be v. By Lemma 6.1, either process 0 executes infinitely often or a state is reached where $corr\_pred$ is true. In the former case, we know that process 0 executes $(v - v0)\ mod\ M$ times and, hence, the value of $x.0$ is eventually v. In the latter case, the lemma is trivially true. □

Lemma 6.4 If $M \geq N+1$, then in any state there exists a value, say v, in the range $0..(M-1)$ such that the x values of all non-zero processes are different from v. Formally,

$$\forall s :: (\exists v : 0 \leq v < M : (\forall j : j \neq 0 : x(s)(j) \neq v))$$

Proof. Note that this lemma essentially states the pigeonhole principle: There are at most N distinct x values of non-zero processes and, hence, if $M \geq N+1$, there must exist a value that is different from the x values of all non-zero processes. We prove this lemma in the following steps:

(1) $|\{x.j : j \neq 0\}|$ is at most N,

(2) $(|\{v : 0 \leq v < M\} - \{x.j : j \neq 0\}|)$ is non-zero if $M \geq N+1$.

We use the set library in PVS in our proof of (1) and (2). This library defines various operations with sets such as union, intersection, difference, cardinality, etc, and provides some standard lemmas about them.

Given a state s, we define the set of x values of non-zero processes upto j, $nonz\_set\_upto(s)(j)$ as follows:

$$nonz\_set\_upto(s)(j) = \begin{cases} \{\} & \text{if } j = 0 \\ nonz\_set\_upto(s)(j-1) \cup \{x(s)(j)\} & \text{otherwise} \end{cases}$$

By induction on j, we then prove that the cardinality of $nonz\_set\_upto(s)(j)$ is atmost j: The base case, $j = 0$, is trivial since $nonz\_set\_upto(s)(0)$ is the empty set. For the induction case, we use the fact that $nonz\_set\_upto(j+1) = nonz\_set\_upto(j) \cup \{j+1\}$ and that the cardinality of the union is no greater than the sum of cardinalities. Now, observe that (1) is trivially true if we instantiate $j = N$.

To prove (2), we use the fact that the for any two sets X and Y, $|X - Y| \geq |X| - |Y|$. Letting X be the set $0..(M-1)$, and Y be the set x values of non-zero processes, we show that in any state there exists a missing value, i.e., a value that is different from the x values of all non-zero processes. □

To identify some missing value in state s, we define a constant $missing(s)$ which denotes some arbitrary value that is missing in state s .

Definition. Given a state s, $missing(s)$ is some arbitrary value in the set $(\{v : 0 \leq v < M\} - \{x.j : j \neq 0\})$ □

Remark. Note that the definition of $missing(s)$ is sensible only if $M \geq N+1$. Thus, all theorems that use this definition rely on this assumption. For brevity, however, we will not explicitly specify this assumption in the subsequent theorems. In PVS, we define $M \geq N+1$ as an axiom so that it can be omitted in the statement of the theorems.

Lemma 6.5 Let s be any state in the computation of the corrector and the action at process 0. The x value of any non-zero process in s is either present in the initial state of that computation or it is generated by process 0 in a state preceding s. Formally,

$$\begin{array}{l} \forall seq, n :: \\ \quad run(action\_zero [\!] corr\_prog)(seq) \\ \Rightarrow \\ \quad \forall j : j \neq 0 : \quad (\exists k :: x(seq(n))(j) = x(seq(0))(k)) \\ \qquad\qquad \vee\ (\exists m : m < n : \\ \qquad\qquad\qquad x(seq(n))(j) = x(seq(m))(0)) \end{array}$$

Proof. We prove this lemma by induction on the length of the computation. The base case, the initial state, is trivial; the x values of all non-zero processes are present in the initial state.

For the inductive case, our proof obligation is that if the x value of a non-zero process is changed in the $(n+1)^{th}$ state then that new value is either present in the initial state or it is generated by process 0 in an earlier state. Towards this end, we first do a case-split on which process executes in the n^{th} step: If process 0 executes in the n^{th} step, the x values of non-zero processes remain unchanged. Thus, the proof obligation is trivially satisfied. If a non-zero process, say j, executes in the n^{th} step, only the value of $x.j$ is changed it is set to $x.(j-1)$. We then do a case-split on whether $j = 1$ or $j \neq 1$. If $j = 1$, we show that the value of $x.j$ in the $(n+1)^{th}$ state is generated by process 0 in the n^{th} state. If $j \neq 1$, by induction on the value of $x.(j-1)$, it follows that the new value of $x.j$ is either present in the initial state or it is generated by process 0 in an earlier state. □

Lemma 6.6 In the computation of the corrector and the action at process 0, a state is reached that satisfies one of the following conditions: (1) $x.0$ is equal to a value missing in the initial state and the x values of non-zero processes are different from $x.0$, or (2) $corr\_pred$ is true. Formally,

$$\begin{array}{l} \forall seq :: \quad run(action\_zero [\!] corr\_prog)(seq) \\ \quad \Rightarrow \\ \qquad \exists n :: x(seq(n))(0) = missing(seq(0)) \ \wedge \\ \qquad\quad (\forall j : j \neq 0 : \\ \qquad\qquad x(seq(n))(j) \neq missing(seq(0))) \\ \qquad \vee\ \exists n :: corr\_pred(seq(n)) \end{array}$$

Proof. From Lemma 6.3, in the computation of the corrector and the action at process 0, a state is reached where either $x.0 = missing(seq(0))$ is true or $corr\_pred$ is true. In the latter case, Lemma 6.6 is trivially satisfied. In the former case, we induct on the length of the computation to show that there exists a state, say s, such that $x.0 = missing(seq(0))$ is true in state s, and $x.0 = missing(seq(0))$ is false in all states preceding s in the computation. We then use Lemma 6.5 to show that in state s, $x.0$ is different from the x values of all non-zero processes. By the construction of s, $x.0$ is never equal to $missing(seq(0))$ in any state preceding s. Moreover, by definition of $missing(seq(0))$, it is not present in the initial state at any non-zero process. Thus, from Lemma 6.5, it follows that in state s, the value of $x.0$ is different from the x values of non-zero processes. □

Lemma 6.7 If the corrector executes starting from a state where $x.0$ differs from the x values of all non-zero processes then in any state of that computation if $x.0$ is the same as $x.j$ then the x values of processes $0..j$ are the same as $x.0$. Formally,

$$\begin{array}{l} \forall seq :: \quad run(corr\_prog)(seq) \\ \qquad \wedge\ (\forall j : j \neq 0 : x(seq(0))(j) \neq x(seq(0))(0)) \\ \quad \Rightarrow \\ \qquad (\forall n, j :: x(seq(n))(j) = x(seq(0))(0) \Rightarrow \\ \qquad\quad (\forall k : 0 \leq k \leq j : \\ \qquad\qquad x(seq(n))(k) = x(seq(0))(0))) \end{array}$$

Proof. We prove this result by induction on the length of the computation as well. In the induction case, let $j1$ be a process that executes in the n^{th} step, and let $j2$ be any process that satisfies $x.j2 = x.0$ in the $(n+1)^{th}$ state. To prove that in the $(n+1)^{th}$ state the x values of $0..j2$ are the same as $x.0$, we do a case-split on whether $j2 < j1$, $j2 = j1$, or $j2 > j1$.

In the first case, we show that the x values of processes $0..j2$ remain unchanged and, hence, $x.j2 = x.0$ must be true in the n^{th} state. Therefore, in the $(n+1)^{th}$ state of the computation, the x values of processes $0..j2$ are the same as $x.0$.

In the second case, we show that it must be the case that in the n^{th} state, $x.(j2-1)$ is the same as $x.0$. Hence, in the n^{th} state, the x values of $0..(j2-1)$ are the same as $x.0$. Since in the $(n+1)^{th}$ state $x.j2$ is the same as $x.0$ and the x values of processes $0..(j2-1)$ remain unchanged, it follows that in the $(n+1)^{th}$ state the x values of processes $0..j2$ are the same as $x.0$.

In the third case also, $x.j2$ remains unchanged. Thus, in the n^{th} state, the $x.j2 = x.0$ is true. Therefore, in the n^{th} state $x.j = x.0$ and $x.(j1-1) = x.0$ is also true. Thus, the action of process $j1$ is disabled. □

Theorem 6.8 The action at process 0 does not interfere with the corrector, i.e., the computation of $action\_zero [\!] corr\_prog$ converges to $corr\_pred$. Formally,

$$satisfies(action\_zero [\!] corr\_prog)\\(converges(true, corr\_pred))$$

Proof. We use Lemmas 6.1, 6.4 and 6.7 to prove the above lemma. From 6.4, a state, say s, is reached where $x.0$ differs from the x values of all non-zero processes. From Lemma 6.1, process 0 executes after s. Until 0 executes for the first time, the corresponding computation is a computation of the corrector. Moreover, when 0 executes $x.0$ is the same as $x.N$. Hence, by Lemma 6.7, the x values of all processes are identical. It follows that when 0 executes for the first time after state s, $corr\_pred$ is true. □

7 Discussion

Related work. Since Dijkstra presented the self-stabilizing token ring program in 1974, it has been proved using various techniques [1, 6, 10, 13, 15]. Of these, the proofs by Qadeer and Shankar [13] and Merz [10] have been verified by a theorem prover. Merz constructs a complicated variant function–consisting of the enabled processes, the distance between the x value of the process 0 and the missing value, etc.—and shows that it decreases in every step. In terms of number of interactions required with the theorem prover, it outperforms the proof presented in this paper as well as that by Qadeer and Shankar. However, these reduced interactions come at a very high cost; the creativity required to find this variant function. Also, that proof is hard to comprehend since it does not match with the intuitive understanding of the token ring program.

Qadeer and Shankar closely follow the proof by Varghese [15], and their proof is simpler than that by Merz. However, since they try to prove the properties of the entire program, some of their proofs are more complex than they need to be. For example, they prove that each process eventually gets the token using the following variant: $p(j) = sum\{k : k$ has a token : $(i-j)\ mod\ N + 1\ \}$. One of the reasons they need such a variant function is that they are trying to prove that *starting from an arbitrary state* eventually each process will get the token. However, this property is more general than necessary; one only needs to prove that *after the invariant is established*, eventually each process will get the token. Since we prove the token circulation property only in the invariant states, we do not need such a variant function.

In related work on mechanical verification of self-stabilization, Prasetya [12] has verified a self-stabilizing routing program in a variant of UNITY logic [4] using the theorem prover HOL [7]. He also presents an elegant development of the theory needed in the verification but he seems to require a prohibitively high level of verification effort.

Advantages of component-based mechanical verification. Fault-tolerant programs are often tricky and so need strong assurance; mechanical verification is a very strong form of assurance but previous examples were tours-de-force that required great insight and talent and are not readily transferable to other problems or other people. By way of contrast, our component-based approach is systematic and offers some hope of making these verifications routine. The detector-correctors theory and its application to Dijkstra's token ring program shows that the effort required as well as the amount of invention is reduced. We find that the advantages of component-based mechanical proofs are the same as that of component-based non-mechanical proofs. We discuss some of these advantages below.

Reusability for a variation of the token ring program. The modification of a component in the program preserves the correctness proofs of other components. We find that this property is useful in the mechanical verification of the resulting program as well. For example, observe that if the action at process 0 is changed so that $x.0$ is incremented by k (instead of 1), where k is relatively prime to M, the self-stabilization is preserved. After we proved the correctness of Dijkstra's token ring program, we verified the self-stabilization property of this new program and found that it took approximately 30 additional minutes to obtain the new proof (compared to approximately 4-5 days for the initial proof), and most of the proof was reused.

Reusability of proofs for other fault-tolerant programs. Lemma 6.1 shows that either process 0 executes infinitely often or the correction predicate is established. This proof only depends on the fact that the corrector satisfies its specification in isolation, and not on the actual programs and predicates involved. We, therefore, have extracted a simple interference-freedom lemma that is applicable in other programs. Likewise, Lemma 5.2, only depends upon the ordering between the corrector actions. Such a ordering exists in various programs—including most tree based programs. Therefore, the same proof technique can be used in those programs as well. Also, lemmas that relate to program compositions or interference-freedom techniques such as superposition and eventual termination can be reused in other fault-tolerant programs.

Role of assumptions. Observe that our proof clearly shows the assumption $M \geq N+1$ is not required for the correctness of the fault-intolerant program or the corrector; it is required only to prove that they do not interfere with each other. Thus, if we were to weaken this assumption—say because it is possible to prove stabilization when $M \geq N$—we will need to redo only the proofs that depend on this assumption, namely Lemmas 6.4, 6.6 and 6.8. Likewise, if we could relax this assumption, say by providing higher atomicity to process 0, we could reuse most of the proof.

8 Conclusion and Future Work

In this paper, we presented a component-based proof of Dijkstra's self-stabilizing token program that has been verified in PVS. To prove correctness of this self-stabilizing program, we needed to show two properties: (1) in the absence of faults, the program circulates a token along the ring, and (2) in the presence of faults, the program eventually recovers to a state from where the token circulation is restored. Following our philosophy of program decomposition, we decomposed the fault-tolerant program into the corresponding fault-intolerant program and the corrector. Then, we proved that property (1) is satisfied by focusing on the fault-intolerant program, and considering its execution starting from the invariant states. Subsequently, we proved property (2) by focusing on the corrector, and considering its execution starting from all states. Finally, we showed that the fault-intolerant program and the corrector do not interfere with each other.

Our case study illustrates that the advantages of program decomposition in non-mechanical proofs also apply to mechanical verification. It shows that by focusing on the component responsible for satisfying the property at hand, the proof of the required property is simplified. Also, it shows that the component-based approach readily supports design exploration as modifications to a program often permits the reuse of proofs. Moreover, it demonstrates that mechanical verification of fault-tolerant programs is less of a tour-de-force and more of a straightforward activity.

Regarding future work, we plan to investigate whether other techniques such as phased reasoning [14] based on convergence stairs [8] and hierarchical design of components offer the same advantage in mechanical verification as they do in non-mechanical verification. We also plan to investigate the use of program decomposition in mechanical verification of multitolerant programs [2], i.e., programs that tolerate multiple types of faults with possibly a different type of tolerance to each fault. Multitolerant programs can be decomposed into a fault-intolerant program and components responsible for tolerating each type of fault. Thus, the proof of tolerance property to a given type of fault can be simplified by focusing only on the components responsible for providing tolerance to that type of fault.

References

[1] A. Arora. *A foundation of fault-tolerant computing*. PhD thesis, The University of Texas at Austin, 1992.

[2] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[3] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *International Conference on Distributed Computing Systems*, pages 436–443, May 1998. An extended version of this paper is submitted to IEEE Transactions on Computers.

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

[5] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

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

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

[8] M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[9] Y. Lakhnech and M. Siegel. Deductive verification of stabilizing systems. *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 201–216, 1997.

[10] S. Merz. Mechanical verification of self-stabilizing token ring. Personal communication, 1998.

[11] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[12] I. S. W. B. Prasetya. Mechanically verified self-stabilizing hierarchical algorithms. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), volume 1217 of Lecture Notes in Computer Science*, pages 399–415, 1997.

[13] S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In David Gries and Willem-Paul de Roever, editors, *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)*, pages 424–443, Shelter Island, NY, June 1998. Chapman & Hall.

[14] M. Siegel and F. Stomp. Extending the limits of sequentially phased reasoning. In P. S. Thiagarajan, editor, *Foundations of software technology and theoretical computer science, Lecture Notes in computer science 880*, 1994.

[15] G. Varghese. *Self-stabilization by local checking and correction*. PhD thesis, MIT/LCS/TR-583, 1993.

Symbols

Variable	Used as
p, q	program
$s, s0, s1, s2$	state
seq	trace
S, T	assertion
R	property
m, n	natural number
j, k	process, domain $0..N$
$v, v1, v2$	x value for a process, domain $0..(M-1)$

Expression	Meaning
$x(s)(j)$	The value of $x.j$ in state s
$seq(n)$	n^{th} state in the sequence seq