The Stabilizing Token Ring In Three Bits - Article Summary

Marc Segal
July 12, 2006

Abstract

This is a summary of the paper "The Stabilizing Token Ring in Three Bits" / Mohamed G.Gouda and F. Furman Haddix. The paper describes a deterministic algorithm for a self-stabilizing unidirectional token ring in which each process has a fixed number of states which is not dependent on the number of processes. In particular, the state of each one of the processes is represented by three bits. The algorithm is very simple and quite elegant.

1 Motivation

A self-stabilizing system is a system whose states can be divided into two sets: legal state and illegal states. The system must have the following two properties:

1. From a legal state the system can only transition into another legal state.
2. From any initial state the system reaches a legal state within a finite number of steps.

Self-stabilizing systems hold great importance in designing fault-tolerant systems. Such systems, no matter how they are started will stabilize into a legal state within a short period of time. The system will remain in a legal state until a fault occurs. The self-stabilizing nature of the systems enables it to overcome the fault and return to a legal state within a finite number of steps.

1.1 Dijkstra’s algorithms

In his seminal work "Self-stabilizing systems in spite of distributed control", Dijkstra presented three algorithms for a self-stabilizing token rings. We've studied two of these algorithms in class. Of these three algorithms, Dijkstra’s second and third algorithms are bidirectional: every process looks at the states of both his neighbors on the ring. Dijkstra’s first algorithm is unidirectional. However, the processes in this algorithm must maintain a number of states which is dependent on the number of processes in the system. This limitation makes this algorithm less fit for implementation in hardware. The algorithm described in this paper solves this limitation, as it requires a constant number of states for each process. Namely, each process requires only three boolean variables, and so the total number of states for each process is 8.

2 System model

A ring system consists of $n$ processes, which are denoted by $P_0, P_1, ..., P_{n-1}$. Every process $P_i$ can read and write its own state variables. In addition, it can read the values of its left neighbor $P_{(i-1) \mod n}$.

The state of the ring is defined by the values of the state variables of each one of the processes. The following notations are used in this paper regarding the state variables: the variable $x$ of processor $P_i$ is denoted by $x.i$. 

1
An algorithm defines for each process actions that may be executed, given that certain conditions are satisfied. An action is enabled for execution for a process iff its condition is satisfied. A process is enabled if one of its actions is enabled. An enabled action is not necessarily executed. In each step of the computation an arbitrary subset of the enabled processes execute their enabled actions.

A transition \( T = (S, S') \) of a ring is a pair of states \( S \) and \( S' \), such that \( S' \) is generated from \( S \) by executing a subset of the enabled actions in \( S \).

A computation \( C = S_1, S_2, ..., \) is a series of states of the ring such that each state pair \((S_i, S_{i+1}), i > 0\) of successive states is a transition of the ring. The sequence is either infinite or finite. If it is finite, there must be no enabled action in its last state.

3 Incremental development the algorithm

In order to understand how the 3-bits algorithm works, we first try to develop self-stabilizing algorithms with less states.

3.1 Alg1 - a simple 1-bit algorithm

The first attempt we make is to create a 1-bit algorithm. We shall call this first algorithm \( alg1 \). The actions of \( alg1 \) are described in the following tables.

The algorithm actions for \( P_0 \) are described by:

<table>
<thead>
<tr>
<th>Action name</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z.0</td>
<td>( t.0 = t.(n-1) )</td>
<td>( t.0 := \neg t.0 )</td>
</tr>
</tbody>
</table>

The algorithm actions for \( P_i, i > 0 \) are described by:

<table>
<thead>
<tr>
<th>Action name</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z.i</td>
<td>( t.i \neq t.(i-1) )</td>
<td>( t.i := \neg t.i )</td>
</tr>
</tbody>
</table>

In addition we define process \( P_0 \) to hold a token iff \( t.0 = t.n-1 \). We define process \( P_i, 0 < i < n \) to hold a token iff \( t.i \neq t.i-1 \).

This simple algorithm moves the tokens around constantly. It has, however no way of decreasing the number of tokens. It is quite easy to provide an example of a ring with two or more tokens which circulate endlessly, without decreasing the number of tokens.

3.2 Alg2 - a 1-bit algorithm with different behaviour for T-tokens and F-tokens

In order to solve the problems in Alg1, we propose the following algorithm, \( Alg2 \), in which we make a distinction between two types of tokens. We say that a process has a T-token if it holds a token and its \( t \) variable is true. We say that a process has an F-token if it has a token and its \( t \) variable is false.

In the definition of \( Alg2 \)'s actions we make the following distinction: only T-tokens are allowed to travel along the ring, whereas F-tokens are not allowed to travel.

The actions of \( alg2 \) are described in the following tables.

The algorithm actions for \( P_0 \) are described by:

<table>
<thead>
<tr>
<th>Action name</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z.0</td>
<td>( t.0 = t.(n-1) \land t.0 = true )</td>
<td>( t.0 := \neg t.0 )</td>
</tr>
</tbody>
</table>

The algorithm actions for \( P_i, i > 0 \) are described by:
Alg2 has the property that the number of tokens will decrease during its operation until there is only one token. However, than one token may be an F-token which has no possibility to move. We will get token ring algorithm in which only one processor gets the token. This is not a good solution to the token ring problem.

### 3.3 Alg3 - a 2-bit algorithm in which F-tokens can still move

The idea in this algorithm is to allow F-tokens to move, only slower than T-tokens. We add an additional boolean variable named \textit{ready}. We allow an F-token to be passed from a process only if its \textit{ready} variable is true. If an F-token is passed to a token whose \textit{ready} variable is false, that process has to ‘waste’ one action in order to change the value of \textit{ready} without passing the token. The result is that F-tokens need two actions by each process on their way for each hop, as opposed to T-tokens that only need one.

The actions of the algorithm are described by the following tables.

<table>
<thead>
<tr>
<th>Action name</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X.0)</td>
<td>(t.0 \neq t.(n-1))</td>
<td>(\text{ready.0} = \text{false})</td>
</tr>
<tr>
<td>(Y.0)</td>
<td>(t.0 = t.(n-1) \land \neg t.0 \land \neg \text{ready})</td>
<td>(t.0 := \neg t.0; \text{ready.0} = \text{true})</td>
</tr>
<tr>
<td>(Z.0)</td>
<td>(t.0 = t.(n-1) \land (t.0 = \text{true} \lor \text{ready}))</td>
<td>(t.0 := \neg t.0; \text{ready.0} = \text{false})</td>
</tr>
</tbody>
</table>

The algorithm actions for \(P_i, i > 0\) are described by:

<table>
<thead>
<tr>
<th>Action name</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X.i)</td>
<td>(t.i = t.(i-1))</td>
<td>(\text{ready.i} := \text{false})</td>
</tr>
<tr>
<td>(Y.i)</td>
<td>(t.i \neq t.(i-1) \land \neg t.i \land \neg \text{ready})</td>
<td>(t.i := \neg t.i; \text{ready.i} = \text{true})</td>
</tr>
<tr>
<td>(Z.i)</td>
<td>(t.i \neq t.(i-1) \land (t.i = \text{true} \lor \text{ready}))</td>
<td>(t.i := \neg t.i; \text{ready.i} = \text{false})</td>
</tr>
</tbody>
</table>

This algorithm tries to delay F-tokens by using an additional variable, \textit{ready}. As a result, F-tokens need two actions to be executed by each processor they visit for them to move to the next processor. This would have worked, if there existed a guarantee on the scheduler that T-tokens and F-tokens are scheduled fairly. That is, under a token-fair scheduler, in which processes that hold an F-token have the same chance to execute as processes that hold a T-token, this algorithm will self-stabilize. However, if no such guarantee on the scheduler exists, then the scheduler might give a process that holds an F-token the opportunity to execute two actions in a row each time. Obviously, with such scheduling the algorithm will perform the same as Alg2. In order for the \textit{ready} variable trick to work, some control over scheduling is needed, even if the scheduler is not fair. That goal is achieved using enablers.

### 4 The 3-bit Algorithm : Description

Every process has three boolean state variables:

1. \textit{e} - determines if the process holds an \textit{enabler}
2. \textit{t} - determines if the process holds a \textit{token}
3. \textit{ready} - determines the state of the ready flag

The algorithm defines for each process three possible actions, denoted by \(X,Y,Z\). We denote action \(X\) of process \(P_i\) by \(X.i\). The description of each one of the actions contains a condition. An action can only be executed if its condition is satisfied. The conjunction of the conditions of the three actions \(X,Y,Z\) is empty. This means that in every state at most one action can be executed by a processor. This guarantees that the algorithm is deterministic.
4.1 The algorithm for $P_0$

The algorithm actions of $P_0$ are described in the following table:

<table>
<thead>
<tr>
<th>Action name</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X.0$</td>
<td>$e.0 = e.(n-1) \wedge t.0 \neq t.(n-1)$</td>
<td>$e.0 := \neg e.0; \text{ready} := \text{false}$</td>
</tr>
<tr>
<td>$Y.0$</td>
<td>$e.0 = e.(n-1) \wedge t.0 = t.(n-1) \wedge \neg t.0 \wedge \neg \text{ready}$</td>
<td>$e.0 := e.(n-1); t.0 := \neg t.0; \text{ready} := \text{false}$</td>
</tr>
<tr>
<td>$Z.0$</td>
<td>$e.0 = e.(n-1) \wedge t.0 = t.(n-1) \wedge (t.0 \lor \text{ready})$</td>
<td>$e.0 := e.(n-1); t.0 := \neg t.0; \text{ready} := \text{false}$</td>
</tr>
</tbody>
</table>

4.2 The algorithm for $P_i$

The algorithm actions for $P_i$, $i > 0$ are described in the following table:

<table>
<thead>
<tr>
<th>Action name</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X.i$</td>
<td>$e.i \neq e.(i-1) \wedge t.i = t.(i-1)$</td>
<td>$e.i := \neg e.i; \text{ready} := \text{false}$</td>
</tr>
<tr>
<td>$Y.i$</td>
<td>$e.i \neq e.(i-1) \wedge t.i \neq t.(i-1) \wedge \neg t.i \wedge \neg \text{ready}$</td>
<td>$e.i := \neg e.i; \text{ready} := \text{true}$</td>
</tr>
<tr>
<td>$Z.i$</td>
<td>$e.i \neq e.(i-1) \wedge t.i \neq t.(i-1) \wedge (t.i \lor \text{ready})$</td>
<td>$e.i := \neg e.i; t.i := \neg t.i; \text{ready} := \text{false}$</td>
</tr>
</tbody>
</table>

4.3 Logical entities

In order to understand how the algorithm works, it is best to examine not the values of the state variables of the processes themselves, but to examine the logical entities which are induced by these values. We define two logical entities: an enabler, and a token. Following are their definitions:

Process $P_0$ is a distinguished process. For this process we define the following:

1. We say that $P_0$ holds a token if $t.0 = t.(n-1)$
2. We say that $P_0$ holds an enabler if $e.0 = e.(n-1)$

For process $P_i$, $i = 1..n-1$ we define the following:

1. We say that $P_i$ holds a token if $t.i \neq t.(i-1)$
2. We say that $P_i$ holds an enabler if $e.i \neq e.(i-1)$

In addition, we differentiate between two types of tokens: T-tokens and F-tokens. We say that a process holds a T-token if it holds a token and its $t$ variable is $true$. We say that a process holds an F-token if it holds a token and its $t$ variable is $false$.

4.4 Logical entities movement

It can be seen that given the different definitions for tokens and enablers, the actions of $P_0$ and of the other processes are in fact the same:

1. The $X$ action is enabled for a process if it has an enabler but has no token. The result of the $X$ action is that the enabler is passed to the next process, and the $\text{ready}$ flag is lowered.
2. The $Y$ action is enabled for a process if it has an enabler, has an F-token, and its ready flag is off. The result of the $Y$ action is that the enabler is passed to the next process, and the ready flag is raised.
3. The $Z$ action is enabled for a process if it has an enabler, has a T-token, or has an F-token and its ready flag is raised. The result of the $Z$ action is that the token and enabler are passed to the next process and the ready flag is lowered.

From this abstraction of the processes’ actions we can deduce the rules of movement for each one of the logical entities:
1. Enablers can travel freely. Whenever a process that has an enabler executes an action, the enabler is passed to the next process.

2. T-tokens can also travel freely. Whenever a process that has both an enabler and a T-token executes an action, the T-token is passed to the next process.

3. F-tokens’ movement is limited. They will be passed to the next process only if the process that holds them holds an enabler and its ready flag is raised. If the ready flag is not raised, the enabler will be passed on to the next process, but the F-token will continue to hold the F-token.

4.5 The role of the ready flag

It is important to understand the role of the ready flag. Both the $X$ and the $Z$ actions turn this flag off. The $Y$ action turns this flag on. Therefore, during normal operation of the algorithm, the flag will only be turned on for a process, if it executed a $Y$ action last. A process executed the $Y$ action only when it had an F-token, an enabler, and its ready flag was off. So when the process executed the $Y$ action, it had an enabler but the F-token was not passed to the next process. Therefore, the process must still hold the F-token. The next time it gets an enabler, it will have the ready flag turned on, and so the $Z$ action will be enabled. If the process executes it, it will pass the token to the next process. The conclusion of this analysis is that the ready flag serves to slow down the progress of F-tokens. Whereas T-tokens can “ride” a single enabler all the way around the ring, an F-token needs two different enablers for each step it makes. This difference between the behaviour of T-tokens and F-tokens lies at the heart of this algorithm’s self-stabilization property.

5 Proof of Stabilization

We define a legal state of the ring to be a state in which there is only one token. In order to prove that the algorithm is self-stabilizing we shall prove the following theorem:

**Theorem 1:** Every computation of the ring has an infinite suffix in which each state is legal.

However, before we can prove this theorem, we need to prove a series of lemmas.

5.1 Lemma 1 : e-different and t-different processes

**Definition:** A process $P_i$ is e-different in a state $S$ iff $e.i \neq e.((i-1) \mod n)$ in that state.

**Definition:** A process $P_i$ is t-different in a state $S$ iff $t.i \neq t.((i-1) \mod n)$ in that state.

**Definition:** For a state $S$, we define $N_{e-different}(S)$ to be the number of e-different processes in $S$.

**Definition:** For a state $S$, we define $N_{t-different}(S)$ to be the number of t-different processes in $S$.

**Lemma 1:** In every state $S$ of the ring, $N_{e-different}(S)$ and $N_{t-different}(S)$ are even.

**Proof:** We traverse along the ring starting from $P_0$, counting the number of e-different processes we encounter on the way. We claim that for every process $P_i$ we reach while traversing the ring this way, the number of e-different processes we’ve counted is even iff $e.i = e.0$. We’ll prove this by induction. For $P_1$, if we’ve encountered one e-different process, then $e.1 \neq e.0$ and the claim holds. And if we’ve encountered no e-different processes then $e.1 = e.0$ and the claim holds. Now, assume the claim holds for $P_i$. If we’ve encountered an even number of e-different processes until $P_i$, then $e.i = e.0$. If the number of processes we encounter after reaching $P_{i+1}$ is odd, then $P_{i+1}$ is e-different and therefore $e.i + 1 \neq e.i = e.0$. If however, the number of e-different processes is even then $P_{i+1}$ is not e-different and therefore $e.i + 1 = e.i = e.0$. If we’ve encountered an odd number of e-different processes until $P_i$, then $e.i \neq e.0$. If the number of e-different processes we encounter after $P_{i+1}$ is even, then $P_{i+1}$ is e-different and therefore $e.i + 1 \neq e.i \neq e.0$ which leads
us to the conclusion that $e.i + 1 = e.0$. And if the number of $e$-different processes after $P_{t+1}$ is odd then $P_{t+1}$ is not $e$-different and therefore $e.i + 1 = e.i \neq e.0$.

We apply this claim for $P_{n-1}$. If the number of $e$-different processes we’ve encountered between $P_0$ and $P_{n-1}$ is even, then $e.n - 1 = e.0$. We conclude that $P_0$ is not $e$-different and therefore the total number of $e$-different processes $N_{e-different}(S)$ is even. Otherwise, if the number of $e$-different processes between $P_0$ and $P_{n-1}$ is odd, then $e.n - 1 \neq e.0$. Therefore, $P_0$ is $e$-different. The total number of $e$-different processes is one plus the number we encountered and so it is odd. □

5.2 Lemma 2 : the number of tokens and enablers

Definition: For a state of the ring $S$, we define $N_{enablers}(S)$ to be the number of processes that hold an enabler in $S$.

Definition: For a state of the ring $S$, we define $N_{tokens}(S)$ to be the number of processes that hold a token in $S$.

Lemma 2: In every state $S$ of the ring, $N_{enablers}(S)$ and $N_{tokens}(S)$ are odd.

Proof: We identify two cases:

1. $P_0$ is $e$-different. In that case, according to the definition, $P_0$ does not hold an enabler. Therefore, all the processes that hold an enabler come from the last $n - 1$ processes. Since these processes hold an enabler iff they are $e$-different we can tell that their number is $N_{enablers}(S) = N_{e-different}(S) - 1$, which is odd.

2. $P_0$ is not $e$-different. In that case, we can tell that among the last $n - 1$ processes there are $N_{e-different}(S)$ processes which have enablers. We can add to that $P_0$, which according to the definition holds an enabler too. The total number of enablers is: $N_{enablers}(S) = N_{e-different}(S) + 1$ which again is odd.

□

5.3 Lemma 3 : adjacent token types

Lemma 3: Let $S$ be a state of the ring. Let $P_i, P_j, j > i$ be two processes that hold tokens in $S$, so that no process from $P_{i+1}, ..., P_{j-1}$ holds a token. Then, either $P_i$ holds a T-token and $P_j$ holds an F-token, or $P_i$ holds a T-token and $P_j$ holds a T-token.

Proof: Let $x = t.i$. Since none of the processes $P_{i+1}, P_{i+2}, ..., P_{j-1}$ hold a token then it must be true that $x = t.i = t.(i+1) = ... = t.(j-1)$. Since $P_j$ holds a token, $t.j = \neg t.(j-1) = \neg x$. Obviously, if $P_i$ holds a T-token, $x = true$ and then, $t.j = false$, meaning $P_j$ holds an F-token. On the other hand, if $P_i$ holds an F-token, then $x = false$ and $t.j = true$, meaning $P_j$ holds a T-token. □

5.4 Lemma 4 : extreme token types

Lemma 4: Let $S$ be a state of the ring. Let $P_i, P_j, j > i$ be two processes that hold tokens in $S$, so that no process from $P_{i+1}, ..., P_{j-1}$ holds a token. Then, either both $P_i$ and $P_j$ hold a T-token, or both $P_i$ and $P_j$ hold an F-token.

Proof: Let $x = t.j$. Since none of the processes $P_{i+1}, P_{i+2}, ..., P_{j-1}$ hold a token, we conclude that $x = t.j = t.(j+1) = ... = t.(n-1)$. We now differentiate between two cases:

1. If $i = 0$, $P_0$ holds a token, and therefore $t.0 = t.(n-1) = x$. We conclude that since the value of the variable $t$ of $P_i$ and $P_j$ is equal, they must both hold the same type of token.

2. Otherwise, $P_0$ doesn’t hold a token, and therefore $t.0 = \neg t.(n-1) = \neg x$. Since none of the processes $P_i, ..., P_{i-1}$ have a token, we have $t.0 = t.1 = ... = t.(i-1)$. And since $P_i$ holds a token, $t.i = \neg t.(i-1)$. We conclude that $t.i = \neg \neg x = x$, and therefore $P_i$ holds the same kind of token as $P_j$. □
5.5 Lemma 5: monotonicity of the number of tokens

Lemma 5: Let \( T = (S, S') \) be a transition of the ring. Then \( N_{\text{tokens}}(S') \leq N_{\text{tokens}}(S) \) and 
\( N_{\text{enablers}}(S') \leq N_{\text{enablers}}(S) \)

Proof: Let \( P_j \) be a process so that \( P_j \) has a token in \( S' \), but does not have a token in \( S \). There are two possibilities: either in the transition \( (S, S') \) the value of \( t.(i - 1 \mod n) \) changed and \( t.i \) remained constant, or \( t.(i - 1 \mod n) \) remained constant and \( t.i \) changed value. However, the value of the variable \( t \) changes only as part of the \( Z \) action, and this action is only executed if a process has a token. Since process \( P_i \) has no token in \( S \), it could not have executed action \( Z \) and so \( t.i \) can’t change in the transition \( (S, S') \). We conclude that process \( P_{i-1 \mod n} \) must have had a token in \( S \) and executed action \( Z.(i - 1 \mod n) \) in the transition \( (S, S') \). Let us now traverse the ring backwards until we find the first processor that does not execute action \( Z \) in the transition \( (S, S') \). There must be such a processor, because \( P_i \) is such a processor. So, we define:
\( j = \min \{1 < k \leq n \mid t.(i - k \mod n)(S) = t.(i - k \mod n)(S') \} \). And process \( P_j \) according to this definition is that process. By definition, \( P_{j+1 \mod n} \) holds a token in \( S \) and executes the \( Z \) action in the transition \( (S, S') \). Since in that action it changes the value of its \( t \) variable, then in \( S' \) it must have \( t.(j + 1 \mod n)(S') = t.(j + 1 \mod n)(S) = t.j(S) = t.j(S') \). We conclude that \( P_{j+1 \mod n} \) does not hold a token in \( S' \).

We’ve shown that for every process \( P_i \) that gains a token in the transition \( (S, S') \), there is another process \( P_i' \) which loses a token in that transition. The conclusion is that \( N_{\text{tokens}}(S') \leq N_{\text{tokens}}(S) \).

The proof that \( N_{\text{enablers}}(S') \leq N_{\text{enablers}}(S) \) is similar. □

5.6 Lemma 6: infinite computation

Lemma 6: Every computation \( C \) of the ring is infinite.

Proof: Assume by way of contradiction that there is a finite computation \( C \) of the algorithm. Then, by definition, \( C \) has a final state \( S \) in which no action is enabled. By lemma 2, \( N_{\text{enablers}}(S) \) is odd, and therefore there must be at least one processor in \( S \) that holds an enabler. But, according to the algorithm, every enabled processor has exactly one enabled action. Therefore, there exists a processor in \( S \) that has an enabled action. Contradiction. □

5.7 Lemma 7: infinite computation for each process

Lemma 7: In every computation \( C \), every process \( P_i \) on the ring has at least one action which is executed infinitely many times.

Proof: Let \( C \) be a computation of the algorithm. By lemma 6, \( C \) is infinite. Since the number of processes is finite, there must exist a process which executes an infinite number of actions in \( C \). Assume by way of contradiction that not all processes execute an infinite number of actions. Then there must exist a process \( P_i \) so that \( P_i \) executes an infinite number of actions in \( C \), but \( P_{i-1 \mod n} \) executes a finite number of actions in \( C \). Let \( C' = S_0, S_1, ... \) be an infinite suffix of \( C \) so that \( P_{i-1 \mod n} \) does not execute any action in \( C' \). Since \( P_{i-1 \mod n} \) does not execute any action in \( C' \), then the value of its \( e \) variable is constant in \( C' \). Let \( S_k \) be the first state in \( C' \) so that \( P_i \) executes an action in \( (S_{k-1}, S_k) \). Assume w.l.o.g that \( i \neq 0 \). Then, in \( S_k \) it must be that \( e.i = e.(i - 1 \mod n) \). Since \( P_i \) executes an infinite number of actions, there must be a state \( S_{k'}, k' > k \), so that \( P_i \) executes an action in the transition \( (S_{k'-1}, S_{k'}) \). Let \( S'_k \) be the first such state. Since \( P_i \) executes an action in the transition \( (S_{k'-1}, S_{k'}) \), it must be enabled in \( S_{k'-1} \). Therefore, in \( S_{k'-1} \) it must be that \( e.i \neq e.(i - 1 \mod n) \). Since the value of \( e.(i - 1 \mod n) \) is constant, it must be that the value of \( e.i \) changed between \( S_k \) and \( S_{k'-1 \mod n} \). But, the value of the variable \( e \) can only change as a result of executing an action, and \( P_i \) executes no action between \( S_k \) and \( S_{k'-1 \mod n} \). Contradiction. Therefore, all processes execute an infinite number of actions in \( C \). □
5.8 Token annihilation

We define token annihilation to be a transition $T = (S, S')$ in which the following three conditions are satisfied:

1. There are two adjacent processes $P_i$ and $P_{(i+1) \mod n}$ so that both processes hold a token in $S$
2. $P_i$ executes action $Z.i$ in the transition from $S$ to $S'$
3. $P_{(i+1) \mod n}$ either executes no action in the transition from $S$ to $S'$ or executes action $Y.i + 1$.

After the token annihilation occurs, process $P_{i+1}$ does not hold a token. To see that this is indeed the case, let’s consider the values of the $t$ variable in $P_{i-1}, P_i, P_{i+1}$. We assume here w.l.o.g that $1 < i < n-1$. Since in $S$ both $P_i$ and $P_{i+1}$ have tokens we have in $S$, $t.(i-1) \neq t.i \neq t.(i+1)$. Since in the transition from $S$ to $S'$, $P_i$ executes action $Z.i$ and $P_{i+1}$ executes no action, the value of $t.i$ is negated in the transition, whereas the value of $t.(i+1)$ does not change. We conclude that in $S'$, $t.i = t.(i+1)$.

**Definition:** A computation $C = S_0, S_1, ...$ is called annihilation-free if for every couple of successive states in $C$, $S_i$ and $S_{i+1}$, the transition $(S_i, S_{i+1})$ is not a token-annihilation.

5.9 Lemma 8 : No process holds a token forever

**Lemma 8:** Let $C = S_0, S_1, ...$ be an annihilation-free computation on the ring. Let $S_k$ be a state in $C$ in which process $P_i$ holds a token. Then, there exists $m > k$ so that in the transition $(S_m, S_{m+1})$ process $P_i$ executes action $Z.i$, and in state $S_{m+1}$ the token is passed to process $P_{(i+1) \mod n}$.

**Proof:** Let us assume by way of contradiction that process $P_i$ does not execute action $Z.i$ in any subsequent transition in $C$ after $S_k$. By lemma 7, process $P_i$ executes an infinite number of actions in $C$. Since $P_i$ holds a token in $S_k$, and the computation is annihilation-free, then the value of $z.(i-1 \mod n)$ can’t be changed in any transition without the value of $z.i$ changing in the same transition. Therefore, $P_i$ will continue to hold a token in every subsequent state in $C$ until it executes action $Z.i$. Since we know that $P_i$ executes an infinite number of actions in $C$, and as long as $P_i$ holds a token it can only execute either action $Z.i$ or $Y.i$, we have to show that it is impossible for $P_i$ to only execute $Y.i$. And so, let $S_k', k' > k$ be the first state in $C$ so that $P_i$ executes an action in the transition $(S_k, S_{k'})$. If the action that $P_i$ executes is $Z.i$ then we’re done. Otherwise, $P_i$ must execute action $Y.i$. As a result of this action $ready.i = true$ in $S_{k'}$. In every subsequent state after $S_{k'}$, until $P_i$ executes an action, $P_i$ holds a token and $ready.i = true$. The only enabled action for $P_i$ in such a condition is $Z.i$. Therefore, we conclude that there must exist a state $S_{k''}, k'' > k'$ so that $P_i$ executes action $Z.i$ in the transition $(S_{k''}, S_{k''})$. □

5.10 Lemma 9 : tokens travel around the ring

**Lemma 9:** Let $C = S_0, S_1, ...$ be an annihilation-free computation on the ring. If in state $S_m$ process $P_i$ holds a token, then there exists a series of states $S_m = S_{m_0}, S_{m_1}, ..., S_{m_n}$, $m_0 < m_1 < ... < m_n$ in $C$ so that for every $j = 1...n$:

1. The token is passed to $P_{(i+j) \mod n}$ in the transition $(S_{m_{j-1}}, S_{m_j})$
2. If $i + j = n$ then in the transition $(S_{m_{j-1}}, S_{m_j})$ the token is passed inverted
3. If $i + j \neq n$ the token is passed unchanged

**Proof:** Let $S_m$ be a state in $C$ so that $P_i$ holds a token in $S_m$. By lemma 8, there must exist a state $S_{m_1}, m_1 > m$, so that in the transition $(S_{m_1}, S_{m_2})$ $P_i$ executes action $Z.i$ and passes the token to process $P_{i+1 \mod n}$. Since the computation is annihilation-free either $P_{i+1 \mod n}$ does not hold a token in $S_{m_1-1}$, or it holds a token and executes the $Z$ action. Let us examine two cases:
1. If \(i = n - 1\) : if we denote by \(x\) the value of \(t.i\) in \(S_{m_1-1}\), then the value of \(i.t\) in \(S_{m_1}\) is \(\neg x\). If \(P_0\) does not hold a token in \(S_{m_1-1}\) then \(t.0 = \neg x\) in \(S_{m_1-1}\) and it remains so in \(S_{m_1}\). We get that in \(S_{m_1}\), the token was passed from \(P_{n-1}\) to \(P_0\) inverted. If \(P_0\) holds a token in \(S_{m_1-1}\) and executes action \(Z\) in the transition, then \(t.0 = x\) in \(S_{m_1-1}\) and \(t.0 = \neg x\) in \(S_{m_1}\). Again, the token was passed from \(P_{n-1}\) to \(P_0\) inverted.

2. If \(i \neq n - 1\) : if we denote by \(x\) the value of \(t.i\) in \(S_{m_1-1}\), then the value of \(i.t\) in \(S_{m_1}\) is \(\neg x\). If \(P_{i+1} \mod n\) does not hold a token in \(S_{m_1-1}\) then \(t.(i+1 \mod n) = x\) in \(S_{m_1-1}\) and it remains so in \(S_{m_1}\). We get that in \(S_{m_1}\), the token was passed from \(P_i\) to \(P_{i+1} \mod n\) unchanged. If, however, \(P_{i+1} \mod n\) holds a token in \(S_{m_1-1}\) then \(t.(i+1 \mod n) = \neg x\) in \(S_{m_1-1}\) and it changes to \(t.(i+1 \mod n) = x\) in \(S_{m_1}\). Again, the token was passed unchanged from \(P_i\) to \(P_{i+1} \mod n\).

We can repeat the step from \(S_m\) to \(S_{m_1}\) \(n\) times to build the series \(S_m = S_{m_0}, S_{m_1}, ... S_{m_n}\) as required. □

### 5.11 Lemma 10: the ready flag and F-tokens

**Lemma 10:** Every annihilation-free computation \(C\) has an infinite suffix \(C'\) in which for every state \(S\) in \(C'\), for every process \(P_i\), if \(\text{ready}.i = \text{true}\) then \(P_i\) holds an F-token in \(S\)

**Proof:** By lemma 7, every process executes an infinite number of action in \(C\). Let \(C'\) be an infinite suffix of \(C\) so that every process has already executed at least one action in \(C\) before \(C'\). Let \(S\) be a state in \(C'\) and let \(P_i\) be some process, so that \(\text{ready}.i = \text{true}\). The value of the \(\text{ready}\) variable can only change as a result of executing an action. The \(X\) and \(Z\) actions set its value to \(\text{false}\), and the \(Y\) action sets its value to \(\text{true}\). Therefore, the last action executed by \(P_i\) before \(S\) is the \(Y\) action. Let \(S'\) be the state in \(C\) so that \(P_i\) executes \(Y.i\) in the transition \((S', S'')\) and \(P_i\) executes no other action between \(S''\) and \(S\). By the definition of the algorithm, \(P_i\) holds an F-token in \(S'\). Since the computation is annihilation-free, in no state between \(S'\) and \(S\) can process \(P_{i-1}\) execute its \(Z\) action. It can therefore not change the value of its \(t\) in any transition between \(S'\) and \(S\). Therefore, \(P_i\) holds the F-token in all states between \(S'\) and \(S\). We conclude that \(P_{n-1}\) holds an F-token in \(S\). □

### 5.12 Lemma 11: the ready flag an F-tokens - continued

**Lemma 10:** Every annihilation-free computation \(C\) has an infinite suffix \(C'\) such that the following two conditions are satisfied:

1. In every transition \((S, S')\) in which an F-token is passed from process \(P_{i-1}\) to process \(P_i\), then \(\text{ready}.i = \text{false}\) in \(S'\)

2. In every transition \((S, S')\) in which an T-token is passed inverted from process \(P_{n-1}\) to process \(P_0\), then \(\text{ready}.0 = \text{false}\) in \(S'\)

**Proof:** Let \(C'\) be an infinite suffix of \(C\) as in lemma 10.

1. Let \((S, S')\) be a transition in which an F-token is passed from process \(P_{i-1}\) to process \(P_i\), \(i > 0\). Assume by way of contradiction that in \(S'\), \(\text{ready}.i = \text{true}\). Then, either \(\text{ready}.i = \text{true}\) in \(S\), or it changed during the transition \((S, S')\). If \(\text{ready}.i = \text{true}\) in \(S\) then by lemma 9, \(P_i\) holds an F-token in \(S\). Since the value of \(\text{ready}.i = \text{true}\) does not change in the transition \((S, S')\), \(P_i\) does not execute any action in that transition. But since \(P_{i-1}\) executes action \(Z.i - 1\) in that transition we have token annihilation in contradiction to the way \(C\) was defined. If, however \(\text{ready}.i = \text{false}\) in \(S\), and it \(\text{ready}.i = \text{true}\) is \(S'\), then \(P_i\) executes the \(Y.i\) action in the transition \((S, S')\). Again, we get a token annihilation in the transition \((S, S')\) in contradiction to the way \(C\) was defined. We conclude that \(\text{ready}.i = \text{false}\) in \(S'\).
2. Let \((S, S')\) be a a transition in which an T-token is passed inverted from process \(P_{n-1}\) to process \(P_0\). Assume by way of contradiction that in \(S'\), \(ready.0 = true\). Then, either \(ready.0 = true\) in \(S\), or it changed during the transition \((S, S')\). If \(ready.0 = true\) in \(S\) then by lemma 9, \(P_0\) holds an F-token in \(S\). Since the value of \(ready.0 = true\) does not change in the transition \((S, S')\), \(P_0\) does not execute any action in that transition. But since \(P_{n-1}\) executes action \(Z.n - 1\) in that transition we have token annihilation in contradiction to the way \(C\) was defined. If, however \(ready.0 = false\) in \(S\), and it \(ready.0 = true\) is \(S'\), then \(P_0\) executes the Y.0 action in the transition \((S, S')\). Again, we get a token annihilation in the transition \((S, S')\) in contradiction to the way \(C\) was defined. We conclude that \(ready.0 = false\) in \(S'\).

\(\square\)

5.13 Theorem 1: proof

We are now equipped with enough lemmas to prove Theorem 1:

**Theorem 1:** Every computation of the ring has an infinite suffix in which each state is legal.

**Proof:** Let \(C\) be a computation of the algorithm. By lemma 6, \(C\) is an infinite computation. By lemma 5, the number of tokens and enablers cannot decrease in \(C\). Let \(C'\) be an infinite suffix of \(C\) in which the number of tokens and enablers is constant. \(C'\) is obviously annihilation-free. Let \(C''\) be an infinite suffix of \(C'\) which satisfies the conditions of lemma 11. Let the numbers of tokens in \(C''\) be \(t\). Let us assume by way of contradiction that \(t > 1\). Let \(S\) be the first state in \(C''\). By lemma 3, there must be a T-token in \(S\). By lemma 9, the T-token will be passed unchanged until it reaches \(P_{n-1}\). From there it will be passed inverted to \(P_0\). By lemma 11, when the T-token is passed to \(P_0\), we have \(ready.0 = false\). Let \(S_0\) be the first state after the T-token is passed inverted to \(P_0\). Since a token cannot be passed without an enabler, \(P_0\) holds an enabler in \(P_0\). We denote the token that \(P_0\) holds in \(S_0\) by \(U\). We denote the enabler that \(P_0\) holds in \(S_0\) by \(E\). By lemma 9, \(U\) will eventually be passed unchanged from \(P_0\) to \(P_1\) and from there it will make the entire journey until it reaches \(P_{n-1}\) and then is passed inverted back to \(P_0\). Looking at how \(U\) travels, we can see that in order to get from one process to the other it needs one enabler. For example, in \(S_0\), \(ready.0 = false\). When \(P_0\) executes an action, it will be the Y action which will set \(ready.0 = true\). But then, \(E\) will pass to \(P_1\). \(U\) will have to be joined by another enabler in order to execute Z.0 and pass on to \(P_1\). That way, every token will take \(U\) to the next processor, and set the \(ready\) flag in that processor.

In total, \(U\) will need \(n\) enablers in order to reach \(P_0\). Since the number of enablers is limited by the number of processors, this means that \(U\) will be joined by the enabler \(E\) once again before it reaches \(P_0\) again. Since \(t > 0\) there must be a token after \(U\) in \(S\). By lemma 3, this token must be a T-token. Let this token be denoted by \(V\). Since the number of tokens is odd, there must be yet another token after \(V\). Let this token be denoted by \(W\). We now follow the path of the enabler \(E\) after \(S\). Since it can only advance forward, there is no token annihilation, and it cannot race past a token, \(E\) will eventually join \(V\). If \(E\) joins \(V\) before \(V\) reaches \(P_0\), it will take it all the way to \(P_0\), then leave it there inverted. If not, it will join it when it is already inverted and will only take it one hop. In any case, when \(E\) passes \(V\), it is after \(V\) has reached \(P_0\) inverted. But since \(W\) was the token after \(V\), then when \(V\) is inverted then \(W\) is also already inverted. Therefore, when \(E\) has passed \(V\) it still has \(W\) as a T-token between it and \(U\). So, in order for \(E\) to get to \(U\) it must first join \(W\). When it does, it will take \(W\) all the way until \(P_{n-1}\). But that is a contradiction, because \(E\) cant reach \(P_{n-1}\) twice before \(U\) reaches it once. We conclude that \(t = 1\) and so the system is in a legal state in \(S\) and will remain in a legal state in every subsequent state. \(\square\)