Convergence Refinement

Murat Demirbas *Anish Arora* *

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210 USA

Abstract

Refinement tools such as compilers do not necessarily preserve fault-tolerance. That is, given a fault-tolerant program in a high-level language as input, the output of a compiler in a lower-level language will not necessarily be fault-tolerant. In this paper, we identify a special class of refinement, namely "convergence refinement", that preserves the fault-tolerance property of stabilization. We illustrate the use of convergence refinement by presenting the first formal design of Dijkstra's little-understood 3-state stabilizing token-ring system. Our designs begin with simple, high-atomicity token-ring systems that are not stabilizing, and then add a high-atomicity "wrapper" to the systems so as to achieve stabilization. Both the system and the wrapper are then independently refined to obtain a low-atomicity implementation, while preserving stabilization. We demonstrate that convergence refinement is amenable for "graybox" design of stabilizing implementations, i.e., design of system stabilization based solely on system specification and without knowledge of system implementation details.

Keywords : Fault-tolerance, stabilization, refinements, compilers, convergence refinement, atomicity, algorithms, token-ring, graybox design

*Email: {demirbas, anish}@cis.ohio-state.edu; Tel: +1-614-292-1836 ; Fax: +1-614-292-2911 ; Web: http://www.cis.ohio-state.edu/{~ demirbas, ~ anish}; This work was partially sponsored by DARPA contract OSU-RF #F33615-01-C-1901, NSF grant NSF-CCR-9972368, an Ameritech Faculty Fellowship, and a grant from Microsoft Research.

1 Introduction

Refinement tools such as compilers, program transformers, and code optimizers generally do not preserve the fault-tolerance properties of their input programs. Consider, for example, a program that is trivially tolerant to the corruption of a variable `x` in that it eventually ensures `x` is always 0.

```
int x=0;
while(x==x) x=0;
```

The bytecode that a Java compiler produces for this input program is not tolerant.

```
0 iconst_0
1 istore_1
2 goto 7
5 iconst_0
6 istore_1
7 iload_1
8 iload_1
9 if_icmpeq 5
12 return
```

If the value of `x` (i.e., the value of the local variable at position 1) is corrupted after line 7 is executed and before line 8 is executed (i.e., during the evaluation of "`x==x`") then the execution terminates at line 12, thereby failing to eventually ensure that `x` is always 0.

As another example, consider the specification of a "best-k bidding server" component. The server accepts bids during a bidding period via a "`bid(integer)`" method and stores only the highest k bids in order to declare them as winners when the bidding period is over. When the "`bid(`v`)`" method is invoked, the server replaces its minimum stored bid with v only if v is greater than this minimum stored bid. The best-k bidding server is tolerant to the corruption of a single slot in that it satisfies the specification for $(k-1)$ out of best-k bids.

Consider now a sorted-list implementation of the best-k bidding server. The implementation maintains the highest k bids in sorted order with their minimum being at the head of the list. When the "`bid(`v`)`" method is invoked on the implementation, it checks whether v is greater than the bid value at the head of the list, and if so, the head of the list is deleted and v is properly inserted to maintain the list sort order. This implementation, while correct with respect to the specification, does not tolerate the corruption of a single stored bid: If the stored bid at the head of the list is corrupted to be equal to `MAX_INTEGER`, then the implementation prevents new bid values from entering the list, and hence fails to satisfy the specification for $(k-1)$ out of best-k bids.

These examples illustrate that even though an abstract system A is fault-tolerant, it is possible that a refinement C of A may not be fault-tolerant since the extra states introduced in C create additional challenges for the fault-tolerance of C. We are therefore motivated to consider the problem of making refinement tools fault-tolerance preserving. In this paper, we focus our attention on preserving the fault-tolerance property of stabilization.

Contributions of the paper. Our main contribution is to identify a special class of refinement, "convergence refinement", that suffices for preserving stabilization. A concrete system C is a convergence refinement of an abstract system A iff C is a refinement of A with respect to the initial states (i.e., every computation of C that starts from an initial state is a computation of A) and every computation c of C that starts from a non-initial state is a convergence isomorphism of some computation a of A. Convergence isomorphism states that c is allowed only to drop some states (except the initial and final[if any] states) that appear in a.

Intuitively speaking, convergence refinement implies that even in the unreachable states the computations of C track the computations of A, although some states that appear in the computations of A may disappear in the computations of C, and hence, C preserves convergence properties of A. In particular, stabilization [3] is preserved: If A is stabilizing then any convergence refinement, C, of A is also stabilizing.

The second contribution of this paper is a formal derivation of Dijkstra's [3] little-understood 3-state stabilizing token ring system (as well as his K-state and 4-state systems) based on convergence refinements. More specifically, we derive Dijkstra's low-atomicity systems starting from simple high-atomicity token-ring systems. In each case, our derivation has two steps: in the first, we choose a well-known non-stabilizing token-ring system and add stabilization to it via a high-atomicity "wrapper" component. In the second step, we independently refine the non-stabilizing system and the wrapper. The composition of the resulting system and wrapper implementations yield Dijkstra's stabilizing token-ring systems. Proceeding in the same vein, we are able to derive a new 4-state and a 3-state stabilizing token-ring system. We note that although there has been a lot of research on Dijkstra's token-ring systems (see for instance [6]) and Ghosh [5] has presented an informal design of Dijkstra's 3-state system, to the best of our knowledge, this is the first time that any of these systems have been formally derived as refinements.

A feature of our convergence refinements is that any wrapper designed to achieve stabilization for an abstract system is reusable for achieving stabilization for any convergence refinement of the abstract system. Said another way, stabilization for a non-stabilizing system implementation is achieved without knowing its implementation details but knowing solely an abstract specification that the system satisfies. Hence, our third contribution is to show that convergence refinement is suitable for so-called "graybox" design of stabilization [1].

The rest of this paper is organized as follows. In Section 2, we show that convergence refinement is stabilization preserving and is amenable for graybox design of stabilization. In Section 3, we present an abstract bidirectional token-ring and then, in Section 4, derive a 4-state system as a convergence refinement of the abstract bidirectional token-ring system. In Sections 5 and 6, we derive Dijkstra's 3-state system and a new 3-state system again as convergence refinements of the abstract bidirectional token-ring system. We discuss some related work in Section 7, and make concluding remarks in Section 8. We relegate the derivation of Dijkstra's K-state protocol from an abstract unidirectional token-ring system to the Appendix.

2 Convergence Refinement

In this section, after some preliminary definitions, we justify why "convergence refinements" preserve stabilization, and are useful for graybox design of stabilization.

Let Σ be a state space.

Definition. A *system* S is a finite-state automaton (Σ, T, I) where T, the set of transitions, is a subset of $\{(s_0, s_1) : s_0, s_1 \in \Sigma\}$ and I, the set of initial states, is a subset of Σ.

A computation of S is a maximal sequence of T transitions, i.e., if a computation is finite there are no transitions in T that start at the final state.

We refer to an abstract system as a *specification*, and to a concrete system as an *implementation*. For now we assume for convenience that the specification and the implementation use the same state space. At the end of this section, in Section 2.3, we present a generalization that allows the implementation to use a different state space than the specification. Henceforth, let C be an implementation and A a specification.

Definition. C is a *refinement* of A, denoted $[C \subseteq A]_{init}$, iff every computation of C that starts from an initial state is a computation of A.

Definition. C is an *everywhere refinement* [1] of A, denoted $[C \subseteq A]$, iff every computation of C is a computation of A.

Definition. A computation c is a *convergence isomorphism* of a computation a iff c is a subsequence of a with at most a finite number of omissions and with the same initial and final(if any) state as a.

For instance, $c = s1\ s3\ s6$ is a convergence isomorphism of $a = s1\ s2\ s3\ s4\ s5\ s6$. However, $c = s1\ s3\ s5\ s6$ is *not* a convergence isomorphism of $a = s1\ s2\ s5\ s6$ since c can only drop states in a, and cannot insert states to a. Intuitively, the convergence isomorphism requirement corresponds to the notion of using similar recovery paths: c should use a similar recovery path with a and not any arbitrary recovery path.

Definition. C is a *convergence refinement* of A, denoted $[C \preceq A]$, iff:

- C is a refinement of A,
- every computation of C is a convergence isomorphism of some computation of A.

Note that convergence refinements are more general than everywhere refinements: $[C \subseteq A] \Rightarrow [C \preceq A]$, but not vice versa.

A fault is a perturbation of the system state. In this paper, we focus on transient faults that may arbitrarily corrupt the process states. The following definition captures a standard tolerance to transient faults.

Definition. C *is stabilizing to* A iff every computation of C has a suffix that is a suffix of some computation of A that starts at an initial state of A.

This definition of stabilization allows the possibility that A is stabilizing to A, that is, A is self-stabilizing.

2.1 Stabilization preserving refinements

Not every refinement is stabilization preserving. That is,

C *refines* A and A *is stabilizing to* A does not imply that C *is stabilizing to* A.

By way of counterexample, consider Figure 1. Here $s0,\ s1,\ s2,\ s3, \ldots$ and s* are states in Σ, and $s0$ is the initial state of both A and C. In both A and C, there is only one computation that starts from the initial state, namely "$s0,\ s1,\ s2,\ s3, \ldots$"; hence, $[C \subseteq A]_{init}$. But "s*, $s2,\ s3, \ldots$" is a computation that is in A but not in C. Letting F denote a transient state corruption fault that yields s* upon starting from $s0$, it follows that although A is stabilizing to A if F occurs initially, C is not.

Figure 1: $[C \subseteq A]_{init}$

We are therefore led to considering everywhere refinements. We had stated in [1] that everywhere refinements are stabilization preserving.

Theorem 0. If $[C \subseteq A]$ and A is stabilizing to B,
then C is stabilizing to B. $\square$

The requirements for everywhere refinements might not be satisfied for every refinement of an abstract system into a concrete one. For instance, every computation of the concrete might not be a computation of the abstract since the execution model of the concrete is more restrictive than that of the abstract. We give examples of model refinements in Section 3 and in the Appendix where a process is allowed to write to the state of its neighbor in the abstract system but not allowed to do so in the concrete system. To address such cases, we consider the more general convergence refinements.

Theorem 1. If $[C \preceq A]$ and A is stabilizing to B,
then C is stabilizing to B. $\square$

Theorem 1 follows immediately from the definitions of stabilization and convergence refinement (C can only drop a finite number of states from the computations of A). Theorem 1 is the formal statement of the amenability of convergence refinements as stabilization preserving refinements.

2.2 Graybox stabilization

Here we focus on the graybox stabilization problem of how to design stabilization to a given implementation C using only its specification A. That is, we want to prove that: If adding a wrapper W to a specification A renders A stabilizing, then adding W to any convergence refinement C of A also yields a stabilizing system. We define a wrapper to be a system over Σ and formulate the "addition" of one system to another in terms of the operator $\Box$ (pronounced "box") which denotes the union of automata.

Lemma 2. If $[C \preceq A]$ and $(A \,\Box\, W)$ is stabilizing to A
then $[(C \,\Box\, W) \preceq (A \,\Box\, W)]$.

Proof: This proof consists of two parts. We prove $[(C \,\Box\, W) \subseteq (A \,\Box\, W)]_{init}$ in the first part, and we prove in the second part that every computation x of $(C \,\Box\, W)$ is a convergence isomorphism of a computation x' of $(A \,\Box\, W)$.

1. $[C \preceq A] \Rightarrow [C \subseteq A]_{init}$. Thus, every computation of C starting from the initial states is a computation of A, and hence $[(C \,\Box\, W) \subseteq (A \,\Box\, W)]_{init}$.

2. Any computation x of $(C \,\Box\, W)$ can be written as $\cdots - CS_i - WS_i - CS_{i+1} - WS_{i+1} - \cdots$ where CS denotes consecutive states produced by C and WS denotes consecutive states produced by W. Since $[C \preceq A]$, C can only drop states from computations of A. Thus, there exists a computation x' of $(A \,\Box\, W)$ of the form $\cdots - AS_i - WS_i - AS_{i+1} - WS_{i+1} - \cdots - A_{init}$ where for all i, CS_i is a convergence isomorphism of AS_i. Since $(A \,\Box\, W)$ is stabilizing to A, x' has a suffix that is a suffix of some computation of A that starts from the initial states. Since $[C \preceq A] \Rightarrow [C \subseteq A]_{init}$, x cannot drop any states from x' after $(A \,\Box\, W)$ stabilizes to A. That is, x can drop only a finite number of states from x',

and hence we conclude that x is a convergence isomorphism of x'. $\Box$

Theorem 3. If $[C \preceq A]$ and $(A \,\Box\, W)$ is stabilizing to A
then $(C \,\Box\, W)$ is stabilizing to A.

Proof:

$[C \preceq A]$
$\Rightarrow$ { $(A \,\Box\, W)$ is stabilizing to A, Lemma 2 }
$[(C \,\Box\, W) \preceq (A \,\Box\, W)]$
$\Rightarrow$ { $(A \,\Box\, W)$ is stabilizing to A, Theorem 1 }
$(C \,\Box\, W)$ is stabilizing to A $\Box$

Theorem 3 states that if a wrapper W satisfies $(A \,\Box\, W)$ is stabilizing to A, then, for any C that satisfies $[C \preceq A]$, $(C \,\Box\, W)$ is stabilizing to A. In fact, after proving Lemma 4, we prove a more general result in Theorem 5.

Lemma 4. If $[W' \preceq W]$ and $(A \,\Box\, W)$ is stabilizing to A
then $(A \,\Box\, W')$ is stabilizing to A.

Proof: Note that $[W' \preceq W]$ and "$(A \,\Box\, W)$ is stabilizing to A" implies $[A \,\Box\, W' \preceq A \,\Box\, W]$. (This proof is similar to the proof of Lemma 2, and hence, is not included here.) Then from Theorem 1 it follows that $[A \,\Box\, W' \preceq A \,\Box\, W]$ and "$(A \,\Box\, W)$ is stabilizing to A" implies $(A \,\Box\, W')$ is stabilizing to A. $\Box$

Theorem 5. If $[C \preceq A]$ and $(A \,\Box\, W)$ is stabilizing to A
then $(\forall W' : [W' \preceq W] : (C \,\Box\, W')$ is stabilizing to $A)$. [1]

Proof:

$[W' \preceq W] \;\wedge\; (A \,\Box\, W)$ is stabilizing to A
$\Rightarrow$ { Lemma 4 }
$(A \,\Box\, W')$ is stabilizing to A
$\Rightarrow$ { $[C \preceq A]$, Lemma 2 }
$[(C \,\Box\, W') \preceq (A \,\Box\, W')] \;\wedge\; (A \,\Box\, W')$ is stabilizing to A
$\Rightarrow$ { Theorem 1 }
$(C \,\Box\, W')$ is stabilizing to A $\Box$

Theorem 5 is the formal statement of the amenability of convergence refinements for graybox stabilization: If W provides stabilization to A, then any convergence refinement W' of W provides stabilization to every convergence refinement C of A.

---

[1] A formula $(op\ i : R.i : X.i)$ denotes the value obtained by performing the (commutative and associative) op on the $X.i$ values for all i that satisfy $R.i$. As special cases, where op is conjunction, we write $(\forall i : R.i : X.i)$, and where op is disjunction, we write $(\exists i : R.i : X.i)$. Thus, $(\forall i : R.i : X.i)$ may be read as "if $R.i$ is true then so is $X.i$", and $(\exists i : R.i : X.i)$ may be read as "there exists an i such that both $R.i$ and $X.i$ are true". Where $R.i$ is true, we omit $R.i$. If X is a statement then $(\forall i : R.i : X.i)$ denotes that X is executed for all i that satisfy $R.i$. This notation is adopted from [4].

2.3 Refinement between different state spaces

The definitions and theorems introduced in this section assumed for the sake of convenience that C and A use the same state space. However, as the examples presented in the introduction illustrate, the state space of the implementation can be different than that of the specification since the implementations often introduce some components of states that are not used by the specifications.

This is handled by relating the states of the concrete implementation with the abstract specification via an abstraction function. The abstraction function is a *total* mapping from Σ_C, the state space of the implementation C, *onto* Σ_A, the state space of the specification A. That is, every state in C is mapped to a state in A, and correspondingly, every state in A is an image of some state in C.

All definitions and theorems in Section 2 are readily extended with respect to the abstraction function.

3 Stabilizing the Bidirectional Token-Ring

In this section, we start with a simple, fault-intolerant abstract bidirectional token-ring system, BTR, and then design two dependability wrappers, $W1$ and $W2$, in order to render BTR stabilizing. $W1$ ensures that always there exists at least one token in the system and $W2$ ensures that the extra tokens in the system are eventually removed.

3.1 Bidirectional token-ring problem

The abstract system BTR consists of processes $\{0,..,N\}$ arranged on a bidirectional ring. Let $\uparrow t.j$ denote that "process j received the token from $j-1$", and $\downarrow t.j$ denote that "process j received the token from $j+1$". Note that $\downarrow t.N$ and $\uparrow t.0$ are undefined for BTR.

We use guarded-command language to specify systems. The actions for 0 –bottom process–, for N –top process–, and for all j such that $(j \neq 0 \ \wedge \ j \neq N)$ are as follows.

$$\begin{array}{lcl}
\uparrow t.N & \longrightarrow & \uparrow t.N := \mathit{false};\ \downarrow t.(N-1) := \mathit{true} \\
\downarrow t.0 & \longrightarrow & \downarrow t.0 := \mathit{false};\ \uparrow t.1 := \mathit{true} \\
\uparrow t.j & \longrightarrow & \uparrow t.j := \mathit{false};\ \uparrow t.(j+1) := \mathit{true} \\
\downarrow t.j & \longrightarrow & \downarrow t.j := \mathit{false};\ \downarrow t.(j-1) := \mathit{true}
\end{array}$$

Remark. We assume that at any process j at most one token may reside at a time. That is, if j had a token and a new token with the same direction as the former is propagated to j, the new token overwrites the old token. (End of Remark.)

Initially, there is a unique token in the system. The invariant I of BTR can be written as $I1 \ \wedge \ I2 \ \wedge \ I3 \ \wedge \ I4$ where

$$
\begin{aligned}
I1 &\equiv (\exists j :: \uparrow t.j \ \vee \ \downarrow t.j) \\
I2 &\equiv (\forall j, k :: ((\uparrow t.j \ \wedge \ \uparrow t.k) \ \vee \ (\uparrow t.j \ \wedge \ \downarrow t.k) \ \vee \ (\downarrow t.j \ \wedge \ \downarrow t.k)) \ \Rightarrow \ j = k) \\
I3 &\equiv (\forall j :: \neg(\uparrow t.j \ \wedge \ \downarrow t.j)) \\
I4 &\equiv (\forall j :: \uparrow t.j \ and \downarrow t.j \ occurs \ with \ equal \ frequency \ )
\end{aligned}
$$

$I1$ states that there exists a token in the system, $I2$ and $I3$ state that at most one process can have a token and only one token, and thus, I states that there is a unique token in the system. $I4$ states that the token changes direction for each successive round.

System models. The abstract system model permits a process j to read and write to its state and the states of its right $(j \oplus 1)$ and left $(j \ominus 1)$ neighbors in one atomic step. The concrete system model is more restrictive: j can read its state and the states of its right $(j \oplus 1)$ and left $(j \ominus 1)$ neighbors but can write only to its own state.

Bidirectional token ring (BTR) problem. Identify refinements, C, of BTR in the concrete system model such that $[C \subseteq BTR]_{init}$ and $(\forall W :: (BTR \ \Box \ W)$ is stabilizing to $BTR \ \Rightarrow \ (C \ \Box \ W)$ is stabilizing to $BTR)$.

From Theorem 5, it follows that any concrete system C that satisfies $[C \ \preceq \ BTR]$ is a solution to the BTR problem.

3.2 Stabilization wrappers for BTR

We add two wrappers $W1$, $W2$ in order to stabilize BTR to $I1$, $(I2 \ \wedge \ I3)$ respectively. We do not need a wrapper to correct $I4$ because $I4$ follows from BTR after $I1 \ \wedge \ I2 \ \wedge \ I3$ is established.

$W1$ secures $I1$ (i.e., there exists at least one token in the system) as follows:

$$
\boxed{W1 :: \quad (\forall j : j \neq N : \neg\uparrow t.j \ \wedge \ \neg\downarrow t.j) \quad \longrightarrow \quad \uparrow t.N := true}
$$

$W2$ secures eventually $(I2 \ \wedge \ I3)$, there exists at most one token in the system, by ensuring at every process j that if ever $\uparrow t.j$ and $\downarrow t.j$ are truthified at the same state, then both of the tokens are deleted. This way, it is clear that tokens moving on opposite directions (toward each other) will cancel each other and their numbers will decrease. If there are multiple tokens all going in one direction, then eventually the tokens will bounce from either top or bottom process and this case reduces to the previous case.

$$W2 :: \ \uparrow t.j \ \wedge \ \downarrow t.j \quad \longrightarrow \quad \uparrow t.j := \mathit{false};\ \downarrow t.j := \mathit{false}$$

Theorem 6. $(BTR \,[]\, W1 \,[]\, W2)$ is stabilizing to BTR. $\square$

4 A 4-state solution to the BTR problem

Consider the following mapping that transforms BTR to an equivalent system BTR_4 that uses two boolean variables $c.j$ and $up.j$ at every process j to simulate $\uparrow t.j$ and $\downarrow t.j$. For every process the mappings between c, up variables and $\uparrow t$, $\downarrow t$ are given as follows.

$$\begin{array}{rcl}
\uparrow t.N & \equiv & c.N \neq c.(N-1) \ \wedge \ up.(N-1) \\
\downarrow t.0 & \equiv & c.0 = c.1 \ \wedge \ \neg up.1 \\
\mathit{For\ all}\ \ j & : & j \neq 0 \ \wedge \ j \neq N, \\
\uparrow t.j & \equiv & c.j \neq c.(j-1) \ \wedge \ up.(j-1) \ \wedge \ \neg up.j \\
\downarrow t.j & \equiv & c.j = c.(j+1) \ \wedge \ \neg up.(j+1) \ \wedge \ up.j
\end{array}$$

We also map $up.N = false$ and $up.0 = true$. Thus, the resulting actions for BTR_4 are as follows.

$$\begin{array}{rcl}
c.N \neq c.(N-1) \ \wedge \ up.(N-1) & & \\
 & \longrightarrow & c.N := c.(N-1);\ up.(N-1) := true \\
c.0 = c.1 \ \wedge \ \neg up.1 & & \\
 & \longrightarrow & c.0 := \neg c.1;\ up.1 := false \\
c.j \neq c.(j-1) \ \wedge \ up.(j-1) \ \wedge \ \neg up.j & & \\
 & \longrightarrow & c.j := c.(j-1);\ up.j := true; \\
 & & c.(j+1) := \neg c.j;\ up.(j+1) := false \\
c.j = c.(j+1) \ \wedge \ \neg up.(j+1) \ \wedge \ up.j & & \\
 & \longrightarrow & up.j := false; \\
 & & c.(j-1) := c.j;\ up.(j-1) := true
\end{array}$$

The initial states of BTR_4 follow from those of BTR using the mapping. BTR_4 uses the same abstract execution model as BTR.

4.1 Refinement of wrappers

We now consider refinements of $W1$ and $W2$ for BTR_4.

$W1$ states that $(\forall j : j \neq N : \neg\uparrow t.j \ \wedge \ \neg\downarrow t.j) \longrightarrow \uparrow t.N := true$. When we apply the mapping on $W1$, we get $W1'$:

$$\begin{array}{lll} (\forall j : j \neq N : up.j) \;\wedge\; c.(N-1) \neq c.N & \longrightarrow & c.N := \neg c.(N-1); \\ & & up.(N-1) := true \end{array}$$

It turns out that $W1'$ is a trivial wrapper since the guard of $W1'$ already implies that $c.N \neq c.(N-1) \;\wedge\; up.(N-1)$. Thus $W1'$ is vacuously implemented.

$W2$ states that if a process j has $\uparrow t.j$ and $\downarrow t.j$ it will drop both of them. $W2'$ is also trivial since using the mapping we get $(\uparrow t.j \;\wedge\; \downarrow t.j \equiv false)$. That is, in BTR_4 j cannot possess $\uparrow t.j$ and $\downarrow t.j$ at the same time.

Lemma 7. $(BTR_4 \;\Box\; W1' \;\Box\; W2')$ is stabilizing to BTR.
Proof.

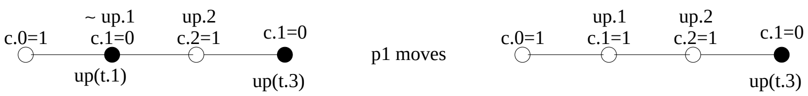
true
$\Rightarrow$ {Theorem 6 }
$BTR \;\Box\; W1 \;\Box\; W2$ is stabilizing to BTR
$\Rightarrow$ { $[W1' \subseteq W1]$, $[W2' \subseteq W2]$, $\Box$ operator }
$[W1' \;\Box\; W2' \subseteq W1 \;\Box\; W2] \;\wedge\; (BTR \;\Box\; W1 \;\Box\; W2)$ is stabilizing to BTR
$\Rightarrow$ { Lemma 4 }
$(BTR \;\Box\; W1' \;\Box\; W2')$ is stabilizing to BTR
$\Rightarrow$ { Theorem 3, $[BTR_4 \subseteq BTR]$ }
$(BTR_4 \;\Box\; W1' \;\Box\; W2')$ is stabilizing to BTR. $\square$

4.2 Refinement of BTR_4

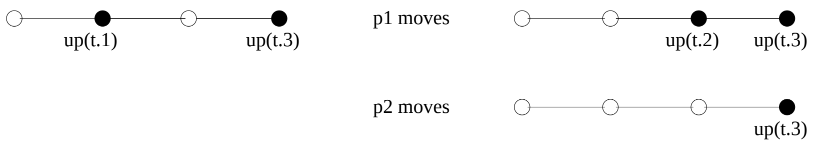
The concrete execution model does not allow writing to the states of the neighboring processes, thus, the actions of BTR_4 are too coarse grained for the concrete execution model. We refine BTR_4 into $C1$ by commenting ("//") out the clauses in BTR_4 that violate the restrictions of the concrete execution model.

$$\begin{array}{rcl} c.N \neq c.(N-1) \;\wedge\; up.(N-1) & & \\ & \longrightarrow & c.N := c.(N-1);\;\; //(up.(N-1)) \\ c.0 = c.1 \;\wedge\; \neg up.1 & & \\ & \longrightarrow & c.0 := \neg c.0;\;\; //(\neg up.1) \\ c.j \neq c.(j-1) \;\wedge\; up.(j-1) \;\wedge\; \neg up.j & & \\ & \longrightarrow & c.j := c.(j-1); up.j := true; \\ & & //(c.(j+1) \neq c.j \;\wedge\; \neg up.(j+1)) \\ c.j = c.(j+1) \;\wedge\; \neg up.(j+1) \;\wedge\; up.j & & \\ & \longrightarrow & up.j := false; \\ & & //(c.(j-1) = c.j \;\wedge\; up.(j-1)) \end{array}$$

In the legitimate states of $C1$ the conditions in the comments are satisfied by the computations of $C1$. However, $C1$ might not satisfy these conditions in every state since the concrete system model is more restrictive than the abstract. In the illegitimate states, where these conditions might not be satisfied, computations of $C1$ might correspond to compressed forms of computations of BTR. Consider the following transition of the concrete.



Starting from a state where $\uparrow t.1$ and $\uparrow t.3$ holds, a state with only $\uparrow t.3$ is true is reached in one transition. This corresponds to a compression of the following transitions of BTR:



Lemma 8. $[C1 \preceq BTR]$.

Proof. Any compression performed by $C1$ only results in a token loss and $C1$ cannot perform any compressions when the token-ring contains less than two tokens. Since there are finite number of tokens to begin with, and since process actions do not create new tokens (they just propagate the existing tokens), $C1$ can do only a finite number of compressions. In BTR, starting from a state with k (s.t., $k > 0$) tokens, any state with l (s.t., $k \geq l > 0$) tokens is reachable. Thus, any computation of $C1$ can be written as a compression of some computation of BTR. Since we also have $[C1 \subseteq BTR]_{init}$, $C1$ is a convergence refinement of BTR. $\square$

Theorem 9. $C1 \;[]\; W1' \;[]\; W2'$ is stabilizing to BTR.

Proof. Since $[W1' \subseteq W1]$ and $[W2' \subseteq W2]$, we have $[W1' \;[]\; W2' \subseteq W1 \;[]\; W2]$. The result then follows from Theorem 5, Lemma 8, and Theorem 6. $\square$

The resulting system $(C1 \;[]\; W1' \;[]\; W2')$ is as follows.

$$
\begin{array}{rcl}
c.(N-1) \neq c.N \;\wedge\; up.(N-1) & \longrightarrow & c.N := c.(N-1) \\
c.1 = c.0 \;\wedge\; \neg up.1 & \longrightarrow & c.0 := \neg c.0 \\
c.(j-1) \neq c.j \;\wedge\; up.(j-1) \;\wedge\; \neg up.j & \longrightarrow & c.j := c.(j-1); up.j := true \\
c.(j+1) = c.j \;\wedge\; \neg up.(j+1) \;\wedge\; up.j & \longrightarrow & up.j := false
\end{array}
$$

Interested reader may note that $(C1 \;[]\; W1' \;[]\; W2')$ can further be optimized to Dijkstra's 4-state stabilizing token-ring system below.

$$
\begin{array}{rcl}
c.(N-1) \neq c.N & \longrightarrow & c.N := c.(N-1) \\
c.1 = c.0 \;\; \wedge \;\; \neg up.1 & \longrightarrow & c.0 := \neg c.0 \\
c.(j-1) \neq c.j & \longrightarrow & c.j := c.(j-1); up.j := true \\
c.(j+1) = c.j \;\; \wedge \;\; \neg up.(j+1) \;\; \wedge \;\; up.j & \longrightarrow & up.j := false
\end{array}
$$

5 A 3-state implementation of BTR

We define a mapping that transforms BTR to an equivalent system BTR_3 that uses a 3-valued counter $c.j$ at every process j to simulate $\uparrow t.j$ and $\downarrow t.j$.

$$
\begin{array}{rcl}
\uparrow t.N & \equiv & c.(N-1) = c.N \oplus 1 \\
\downarrow t.0 & \equiv & c.1 = c.0 \oplus 1 \\
For\ all\ \ j & : & j \neq 0 \;\; \wedge \;\; j \neq N, \\
\uparrow t.j & \equiv & c.(j-1) = c.j \oplus 1 \\
\downarrow t.j & \equiv & c.(j+1) = c.j \oplus 1
\end{array}
$$

Above, $\oplus$ denotes addition operation under modulo 3. BTR_3 follows from the above mapping and uses the same abstract execution model as BTR. Below $\ominus$ denotes subtraction operation under modulo 3.

$$
\begin{array}{rcl}
c.(N-1) = c.N \oplus 1 & \longrightarrow & c.N := c.(N-1) \oplus 1 \\
c.1 = c.0 \oplus 1 & \longrightarrow & c.0 := c.1 \oplus 1 \\
c.(j-1) = c.j \oplus 1 & \longrightarrow & c.j := c.(j-1);\;\; c.(j+1) := c.j \ominus 1 \\
c.(j+1) = c.j \oplus 1 & \longrightarrow & c.j := c.(j+1);\;\; c.(j-1) := c.j \ominus 1
\end{array}
$$

5.1 Refinement of wrappers

We now consider convergence refinements of $W1$ and $W2$ for BTR_3.

$W1$ states that $(\forall j : j \neq N : \neg\uparrow t.j \;\; \wedge \;\; \neg\downarrow t.j) \longrightarrow \uparrow t.N$. When we apply the mapping on $W1$ we get $W1'$:

$$
\begin{array}{lcl}
(\forall j,k : j,k \neq N : c.j = c.k) \;\; \wedge & & \\
\qquad c.N \neq c.(N-1) \oplus 1) & \longrightarrow & c.N := c.(N-1) \ominus 1 \\
 & & //\ i.e.,\;\; c.(N-1) = c.N \oplus 1
\end{array}
$$

$W1'$ is still a global wrapper because the guard of $W1'$ is over the states of all j. We can approximate $W1'$ by using a local wrapper $W1''$ at process N:

$$c.(N-1) = c.0 \;\wedge\; c.N \neq c.(N-1) \oplus 1 \quad \longrightarrow \quad c.N := c.(N-1) \ominus 1$$

$W1''$ is enabled in some states where the abstract $W1$ is not, and hence, is not an everywhere refinement of the abstract wrapper. Thus, we need to prove that $W1''$ does not interfere with the wrapper $W2$. The argument is as follows. $W1'$ is enabled only in the illegitimate states, thus, $(c.(N-1) = c.N \;\wedge\; c.N = c.0)$ implies that the number of tokens in the system is either equal to zero or more than or equal to two. We observe that if the guard $(c.(N-1) = c.N \;\wedge\; c.N = c.0)$ of $W1''$ is infinitely often enabled, then it eventually implies that the number of tokens in the system is equal to zero: From the guard $(c.(N-1) = c.N \;\wedge\; c.N = c.0)$ it follows that between two consecutive executions of $W1'$ process 0 should execute once, that is, a token is bounced up. Therefore, between two consecutive executions of $W1''$, $W2$ executes at least once, and thus, for every extra token that $W1''$ generates, two tokens are deleted by $W2$.

$W2$ states that if j has both $\uparrow t.j$ (i.e., $c.(j-1) = c.j \oplus 1$) and $\downarrow t.j$(i.e., $c.(j+1) = c.j \oplus 1$) then both tokens are deleted. $W2'$ follows directly from the mapping:

$$\begin{array}{lcl} c.(j-1) = c.j \oplus 1 \;\wedge\; c.(j+1) = c.j \oplus 1 & \longrightarrow & c.j := c.(j-1) \\ & & // \; or \; c.j := c.(j+1) \end{array}$$

Lemma 10. $[BTR_3 \;[]\; W1'' \;[]\; W2']$ is stabilizing to BTR. $\square$

5.2 Refinement of BTR_3

The concrete execution model does not allow writing to the states of the neighboring processes, thus, the actions of BTR_3 are too coarse grained for the concrete execution model. We refine BTR_3 into $C2$ by commenting ("//") out the clauses in BTR_3 that violate the restrictions of the concrete execution model.

$$\begin{array}{rcll} c.(N-1) = c.N \oplus 1 & \longrightarrow & c.N := c.(N-1) \oplus 1 & \\ c.1 = c.0 \oplus 1 & \longrightarrow & c.0 := c.1 \oplus 1 & \\ c.(j-1) = c.j \oplus 1 & \longrightarrow & c.j := c.(j-1) & // \; [c.j = c.(j+1) \oplus 1] \\ c.(j+1) = c.j \oplus 1 & \longrightarrow & c.j := c.(j+1) & // \; [c.j = c.(j-1) \oplus 1] \end{array}$$

Note that it is possible to merge $W1''$ with the guard of first action in $C2$ and embed $W2'$ in the third and fourth actions of $C2$. Thus, the resulting system $(C2 \;[]\; W1'' \;[]\; W2')$ is as follows.

$$
\begin{array}{|rcl|}
\hline
c.(N-1) = c.0 \;\wedge\; c.(N-1) \oplus 1 \neq c.N & \longrightarrow & c.N := c.(N-1) \oplus 1 \\
c.1 = c.0 \oplus 1 & \longrightarrow & c.0 := c.1 \oplus 1 \\
c.(j-1) = c.j \oplus 1 & \longrightarrow & \text{if } (c.(j-1) = c.(j+1)) \\
 & & \text{then } c.j := c.(j-1) \\
 & & \text{else } c.j := c.(j-1) \\
c.(j+1) = c.j \oplus 1 & \longrightarrow & \text{if } (c.(j-1) = c.(j+1)) \\
 & & \text{then } c.j := c.(j+1) \\
 & & \text{else } c.j := c.(j+1) \\
\hline
\end{array}
$$

The above system is equal to Dijkstra's 3-state stabilizing token-ring system:

$$
\begin{array}{|rcl|}
\hline
c.(N-1) = c.0 \;\wedge\; c.(N-1) \oplus 1 \neq c.N & \longrightarrow & c.N := c.(N-1) \oplus 1 \\
c.1 = c.0 \oplus 1 & \longrightarrow & c.0 := c.1 \oplus 1 \\
c.(j-1) = c.j \oplus 1 & \longrightarrow & c.j := c.(j-1) \\
c.(j+1) = c.j \oplus 1 & \longrightarrow & c.j := c.(j+1) \\
\hline
\end{array}
$$

Lemma 11. $[C2 \;[\!]\; W1'' \;[\!]\; W2' \preceq BTR_3 \;[\!]\; W1'' \;[\!]\; W2']$.

Proof. Similar to Lemma 8. □

Theorem 12. $C2 \;[\!]\; W1'' \;[\!]\; W2'$ is stabilizing to BTR.

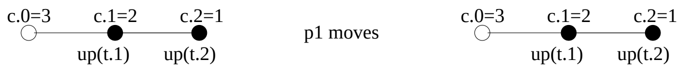
Proof. Follows from Theorem 1, Lemma 11, and Lemma 10. □

6 A new 3-state stabilizing token-ring

In Section 5 we had presented a 3-state implementation $C2$. In this section, we present another 3-state implementation of BTR, $C3$, that uses the same mapping as in Section 5. The system $C3$ is as follows.

$$
\begin{array}{|rcll|}
\hline
c.(N-1) = c.N \oplus 1 & \longrightarrow & c.N := c.(N-1) \oplus 1 & \\
c.1 = c.0 \oplus 1 & \longrightarrow & c.0 := c.1 \oplus 1 & \\
c.(j-1) = c.j \oplus 1 & \longrightarrow & c.j := c.(j+1) \oplus 1 & // \; c.(j-1) \neq c.j \oplus 1 \\
c.(j+1) = c.j \oplus 1 & \longrightarrow & c.j := c.(j-1) \oplus 1 & // \; c.(j+1) \neq c.j \oplus 1 \\
\hline
\end{array}
$$

In the illegitimate states, when the conditions in the brackets are not satisfied, $C3$ takes τ steps (stuttering) as seen in the following case.



Because of the stuttering, $C3$ does not perform any compression of the computations of BTR. Hence, Lemma 13 is trivially satisfied.

Lemma 13. $[C3 \ \preceq \ BTR]$. □

Since we have used the same mapping for $C1$ and $C3$, the same wrappers $W1''$ and $W2'$ that we developed for $C2$ are applicable without any modification for $C3$. Thus, our new 3-state stabilizing system is as follows.

$$
\begin{array}{rcl}
c.(N-1) = c.0 \ \wedge \ c.(N-1) \oplus 1 \neq c.N & \longrightarrow & c.N := c.(N-1) \oplus 1 \\
c.1 = c.0 \oplus 1 & \longrightarrow & c.0 := c.1 \oplus 1 \\
c.(j-1) = c.j \oplus 1 & \longrightarrow & \text{if } (c.(j-1) = c.(j+1)) \\
 & & \text{then } c.j := c.(j-1) \\
 & & \text{else } c.j := c.(j+1) \oplus 1 \\
c.(j+1) = c.j \oplus 1 & \longrightarrow & \text{if } (c.(j-1) = c.(j+1)) \\
 & & \text{then } c.j := c.(j+1) \\
 & & \text{else } c.j := c.(j-1) \oplus 1
\end{array}
$$

Theorem 14. $C3 \ [] \ W1'' \ [] \ W2'$ is stabilizing to BTR.
Proof. Follows from Theorem 3, Lemma 13, and Lemma 10. □

Next we show that our new 3-state stabilizing system above can be refined further to obtain Dijkstra's 3-state system. To this end we use a more aggressive version of $W2'$ that deletes $\uparrow t.j$ when $\uparrow t.(j+1)$ also holds in that state (similarly the wrapper deletes $\downarrow t.j$ when $\downarrow t.(j-1)$ is also true). The resulting system is as follows.

$$
\begin{array}{rcl}
c.(N-1) = c.0 & & \\
\wedge \ c.(N-1) \oplus 1 \neq c.N & \longrightarrow & c.N := c.(N-1) \oplus 1 \\
c.1 = c.0 \oplus 1 & \longrightarrow & c.0 := c.1 \oplus 1 \\
c.(j-1) = c.j \oplus 1 & \longrightarrow & \text{if } (c.(j-1) = c.(j+1)) \\
 & & \text{then } c.j := c.(j-1) \\
 & & \text{else if } (c.j = c.(j+1) \oplus 1) \\
 & & \qquad \text{then } c.j := c.(j-1) \\
 & & \qquad \text{else } c.j := c.(j+1) \oplus 1 \\
c.(j+1) = c.j \oplus 1 & \longrightarrow & \text{if } (c.(j-1) = c.(j+1)) \\
 & & \text{then } c.j := c.(j+1) \\
 & & \text{else if } (c.j = c.(j-1) \oplus 1) \\
 & & \qquad \text{then } c.j := c.(j+1) \\
 & & \qquad \text{else } c.j := c.(j-1) \oplus 1
\end{array}
$$

Since $K = 3$, we have $(\ (c.(j-1) = c.j \oplus 1)\ \wedge\ (c.(j-1) \neq c.(j+1))\ \wedge\ (c.j \neq c.(j+1) \oplus 1)\ )\ \Rightarrow\ (\ c.j = c.(j+1)\ \wedge\ c.(j-1) = c.j \oplus 1\ )$. Thus, the above system can be rewritten as Dijkstra's 3-state system:

$$
\boxed{
\begin{array}{rcl}
c.(N-1) = c.0\ \wedge\ c.(N-1) \oplus 1 \neq c.N & \longrightarrow & c.N := c.(N-1) \oplus 1 \\
c.1 = c.0 \oplus 1 & \longrightarrow & c.0 := c.1 \oplus 1 \\
c.(j-1) = c.j \oplus 1 & \longrightarrow & c.j := c.(j-1) \\
c.(j+1) = c.j \oplus 1 & \longrightarrow & c.j := c.(j+1)
\end{array}
}
$$

7 Related Work

In this section, we discuss some related work on fault-tolerance preserving refinements.

In [1], we had presented another stabilization preserving refinement, namely everywhere-eventually refinement. C is said to be an *everywhere-eventually refinement* of A iff (1) $[C \subseteq A]_{init}$, and (2) every computation of C is an arbitrary finite prefix from the state space Σ followed by a computation of A. It follows from this definition that if A is stabilizing to B, any everywhere refinement C of A is also stabilizing to B.

Everywhere-eventually refinement is more permissive than convergence refinement. That is, every convergence refinement C of A is an everywhere-eventually refinement of A, but not vice versa. C may use a different recovery path than A and still be an everywhere-eventually refinement of A, however, that is not the case for convergence refinements. For example, let A be an abstract program that stabilizes to state $s0$ using a recovery path consisting of odd numbered states (such as $s*\ s3\ s1\ s0$). A concrete program C that uses a recovery path consisting of even numbered states to reach state $s0$ (such as $s*\ s4\ s2\ s0$) is an everywhere-eventually refinement of A but not a convergence refinement of A.

Convergence refinement, by virtue of being more restrictive than everywhere-eventually refinement, is more amenable for the design of graybox stabilization. In order for the graybox wrapping theorem, Theorem 3, to be valid for everywhere-eventually refinements, the wrapper W should truthify the condition $[\Sigma^* A\ [\!]\ W\ \subseteq\ \Sigma^*(A\ [\!]\ W)]$, i.e., W can only add computations to a system and is not allowed to remove any computation from any system. However, there are useful wrappers that do not satisfy this condition. In this paper, by using convergence refinement, which does not have such restrictions on W, we are able to achieve graybox stabilization for a more general class of wrappers.

Liu and Joseph [8] have considered designing fault-tolerance via transformations. In their work, an abstract program A is refined to a more concrete implementation C and then based on the refined program C and the fault actions F that are introduced in the refinement process, further precautions (such as using a checkpointing&recovery protocol) are taken to render C fault-tolerant. They focus on the faults introduced during the refinement, while we focus on the faults that exist in the abstract program. Also, they design the tolerance based on the concrete program, while we design our wrappers based on the abstract program.

Fault-tolerance preserving refinements have been studied in the context of atomicity refinement [10], whereas here we have studied them in the more general context of computation-model refinement. Also, the fault-tolerance preserving refinements presented in [10] are everywhere refinements; here we present a more general type of fault-tolerance preserving refinement, convergence refinement.

McGuire and Gouda [9] have also dealt with fault-tolerance preserving refinements of abstract specifications. They have developed an execution model that can be used in translating abstract network protocol specifications written in a guarded-command language into C programs using Unix sockets. While their framework solves the fault-tolerance preserving refinement problem for a guarded-command to a C program by producing everywhere refinements, the problem remains open for the refinements from a C program to an executable code.

Leal [7] has also observed that refinement tools are inadequate for preserving fault-tolerance. The focus of his work is on defining the semantics of tolerance preserving refinements of components. Whereas, in our work, we have focused on sufficient conditions for fault-tolerance preserving refinements. In his set-up, the state space of the specification and the implementation is the same, whereas in our work they can be different.

The graybox approach has received limited attention in the previous work on dependability. In particular, we can point to [1,2,11] which reason at a graybox level.

8 Concluding Remarks

In this paper, we have investigated stabilization preserving refinements, and, more specifically, have identified convergence refinement as a sufficient condition for preserving stabilization. We have illustrated the use of convergence refinements by deriving several low-atomicity stabilizing token-ring implementations (i.e., a 4-state system, Dijkstra's 3-state and K-state token-ring systems) from high-atomicity stabilizing token-ring systems.

In contrast to traditional designs of stabilizing systems that are implementation-based, we have demonstrated specification-based design of stabilization via convergence refinement. That is, given the fact that a concrete program C is a convergence refinement of an abstract system A, from Theorem 5, it follows that, by inspecting A, we can design a wrapper W to render A stabilizing and then reuse W to achieve stabilization of C. As we have demonstrated through the token-ring examples, the wrapper can achieve stabilization for other refinements of the non-stabilizing system as well. This sort of design—where the fault-tolerance design for a concrete system can be based solely on knowledge of an abstract specification—is called graybox design. Since specifications are typically more succinct than implementations, graybox stabilization offers the promise of scalability. Also, since specifications admit multiple implementations and since system components are often reused, graybox stabilization offers the promise of scalability and reusability. It moreover offers an alternative in closed-source situations, where implementation details are not available. In such situations, treating the system as a blackbox may sometimes yield a high-cost design. Exploiting a specification may therefore be warranted, and convergence refinement is useful in this process.

In closing, we note that convergence refinements are sufficient conditions for stabilization preserving refinements. An interesting direction for future research is to identify necessary conditions for stabilization (and other fault-tolerance) preserving refinements.

References

[1] A. Arora, M. Demirbas, and S. S. Kulkarni. Graybox stabilization. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 389–398, July 2001.

[2] A. Arora, S. S. Kulkarni, and M. Demirbas. Resettable vector clocks. *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 269–278, August 2000.

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

[4] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[5] S. Ghosh. Understanding self-stabilization in distributed systems, Part I. Technical Report 90-02, Computer Science Department, University of Iowa, 1990.

[6] S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilization (WSS'99) Austin, Texas, USA*, pages 33–40, June 1999.

[7] William Leal. *A Foundation for Fault Tolerant Components*. PhD thesis, The Ohio State University, 2001.

[8] Z. Liu and M. Joseph. Transformations of programs for fault-tolerance. *Formal Aspects of Computing*, 4(5):442–469, 1992.

[9] T. M. McGuire. Correct implementation of network protocols from abstract specifications. PhD Thesis in progress, http://www.cs.utexas.edu/users/mcguire/research/html/dp/.

[10] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing, special issue in Self-Stabilization*, 2001, to appear.

[11] J. Rushby. Calculating with requirements. *Invited paper presented at 3rd IEEE International Symposium on Requirements Engineering*, pages 144–146, January 1997.

Appendix: Stabilizing the Unidirectional Token-Ring

In this section, we first present a simple, fault-intolerant abstract unidirectional token-ring system, UTR, and derive a concrete token-ring system, C, that is a convergence refinement of UTR. Then, we show that the dependability wrappers, $W1$ and $W2$, that render UTR stabilizing also provide stabilization to C, and that $(C \;\Box\; W1 \;\Box\; W2)$ results in Dijkstra's K-state stabilizing token-ring system. The dependability wrapper $W1$ ensures that always there exists at least one token in the system and $W2$ ensures that the extra tokens in the system are eventually removed.

Unidirectional token-ring problem

The abstract system UTR consists of processes $\{0,..,N\}$ arranged on a unidirectional ring. Let $t.j$ denote that "the token is at process j" (i.e., j is privileged). The action at every process j is as follows.

$$\boxed{\quad t.j \quad \longrightarrow \quad t.j := \mathit{false}; \;\; t.(j \oplus 1) := \mathit{true} \quad}$$

Above, $\oplus$ denotes $+$ operation under modulo $N+1$. If the token is at process j, j guarantees that in the next step the token resides at $j \oplus 1$.

Remark. We assume that at any process j at most one token may reside at a time. That is, if j had a token and a new token is propagated to j, the new token overwrites the old token. (End of Remark.)

Initially, there is a unique token in the system. The invariant I of UTR can be written as $I1 \;\wedge\; I2$ where

$$\begin{array}{rcl} I1 & \equiv & (\exists j :: t.j) \\ I2 & \equiv & (\forall j,k :: t.j \;\wedge\; t.k \;\Rightarrow\; j = k) \end{array}$$

$I1$ states that there exists a token in the system, $I2$ states that at most one process can have a token, and thus, I states that there is a unique token in the system.

System models. The abstract system model permits a process j to read and write to its state and the state of its right neighbor $(j \oplus 1)$ in one atomic step. The concrete system model is more restrictive: j can read only the state of its left neighbor $(j \ominus 1)$ and write only to its own state.

Unidirectional token-ring (UTR) problem. Identify refinements, C, of UTR in the concrete system model such that $[C \subseteq UTR]_{init}$ and $(\forall W :: (UTR \,[]\, W)$ is stabilizing to $UTR \;\Rightarrow\; (C \,[]\, W)$ is stabilizing to $UTR)$.

From Theorem 5, it follows that any concrete system C that satisfies $[C \preceq UTR]$ is a solution to the UTR problem.

Stabilization wrappers for UTR

We add two wrappers $W1$ and $W2$ in order to stabilize UTR to $I1$ and $I2$ respectively.

$W1$ secures $I1$ (i.e., there exists at least one token in the system) as follows:

$$\boxed{(\forall j : j \neq 0 : \neg t.j) \quad \longrightarrow \quad t.0 := \mathit{true}}$$

In order to achieve $I2$, there exists at most one token in the system, $W2$ marks the token "uniquely" at a distinguished process, say 0, and asserts that 0 does not execute a propagate token action unless it receives the token it has sent with the unique identifier.

Since we are designing stabilizing fault-tolerance, $W2$ can be refined by requiring only that 0 *eventually* marks the token uniquely. Since there are $N+1$ processes, there can be at most N different values in the system (not including process 0). Because of the propagate token action at every process, it is guaranteed that 0 will receive back the token it has sent. If 0 maintains a FIFO queue, Q, of size N to store the values it receives from process N, and marks the token with a value v such that $v \notin Q$, then the token marker v is guaranteed to be unique after 0 receives N values from process N. This is because, when 0 receives N values from process N, Q is guaranteed to contain all possible token values that could have initially existed in the system, and when 0 sends a different token from those in Q that token is guaranteed to be unique.

Theorem 15. $(UTR \; [] \; W1 \; [] \; W2)$ is stabilizing to UTR. $\square$

A K-state solution to the UTR problem

Consider the following mapping that transforms UTR to an equivalent system UTR_K that uses uses a K-valued counter $c.j$, at every process j to simulate tokens.

$$
\boxed{
\begin{array}{rcl}
t.0 & \equiv & c.0 = c.N \\
(\forall j : j \neq 0 : \;\; t.j & \equiv & c.j \neq c.(j-1))
\end{array}
}
$$

Note that the above mapping satisfies the remark in the UTR problem. The actions for UTR_K follows from the mapping and UTR.

$$
\boxed{
\begin{array}{rcl}
c.0 = c.N & \longrightarrow & c.0 := c.0 \oplus 1; \\
 & & if\ (c.0 = c.1)\ then\ c.1 := c.0 \ominus 1 \\
c.j \neq c.(j-1) & \longrightarrow & c.j := c.(j-1); \\
 & & if\ (c.j = c.(j+1))\ then\ c.(j+1) := c.j \ominus 1 \\
c.N \neq c.(N-1) & \longrightarrow & c.N := c.(N-1); c.0 := c.N
\end{array}
}
$$

Above, $\oplus$ and $\ominus$ denote addition and subtraction operations under modulo K, the bound on c values. The bound, K, on c should allow the processes to distinguish between their privileged and non-privileged states, and thus, K should be at least equal to 2. The initial states of C follows from the initial states of UTR using the mapping.

Refinement of wrappers

We now consider refinements of $W1$ and $W2$ for UTR_K.

$W1$ states that $(\forall j : j \neq 0 : \neg t.j) \Rightarrow t.0$. Observe that $(\forall j : j \neq 0 : \neg t.j) \Rightarrow (c.0 = c.N)$. Since $t.0$ is defined as $c.0 = c.N$, $W1$ is already implemented in UTR_K.

$W2$ requires 0 to maintain a queue to store the values it receives from process N, and to eventually mark the token uniquely. We implement $W2$ using the following mapping:

$$
\begin{array}{rcl}
K - 1 & \geq & Q.size \\
lastval.0 & = & Q.top \\
c.0 := lastval.0 + 1 & \equiv & distinct\ token\ t,\ s.t.\ (t \notin Q)
\end{array}
$$

It is obvious that 0 sends K different values to the system since 0 sets $c.0$ to $c.0 + 1$ $(mod\ K)$ each time it propagates a token to process 1. Using the above mapping, Q can be discarded as follows. The elements in Q are now in increasing order, each 1 apart from the previous (due to the way the distinct token is created). Thus, using the top element of Q (i.e., $lastval.0$) is sufficient for creating a distinct token that does not exist in Q. From the mapping K is at least $N + 1$ (i.e., the number of processes in the system). Therefore, when 0 sends K different values to the system it is guaranteed that 0 has sent a unique value to the system. Note that had $K \leq N$, this guarantee would not hold since the system might already contain all the K possible values in N processes (i.e., the processes other than 0).

In sum, $W2'$ requires 0 to store the value of the last token it has sent (i.e., always $lastval.0 = c.0$) and not to execute a propagate token action unless it receives the token with the value it has sent. $W2$ then sends a new token to the system by incrementing $lastval.0$ (i.e., $c.0$) by 1.

Lemma 16. $[UTR_K \ \square\ W1' \ \square\ W2']$ is stabilizing to UTR. $\square$

Our stabilizing unidirectional token ring system $(UTR_K \ \square\ W1' \ \square\ W2')$ is as follows.

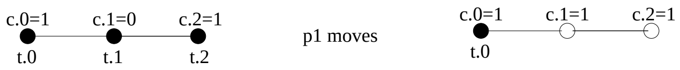
$$
\begin{array}{rcl}
c.0 = c.N \ \ \wedge \ \ lastval.0 = c.N & \longrightarrow & c.0 := c.0 \oplus 1;\ lastval.0 := c.0; \\
 & & if\ (c.0 = c.1)\ then\ c.1 := c.0 \ominus 1 \\
c.j \neq c.(j-1) & \longrightarrow & c.j := c.(j-1); \\
 & & if\ (c.j = c.(j+1))\ then\ c.(j+1) := c.j \ominus 1 \\
c.N \neq c.(N-1) & \longrightarrow & c.N := c.(N-1); c.0 := c.N
\end{array}
$$

8.1 Refinement of UTR_K

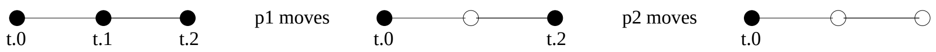
The concrete execution model does not allow writing to the states of the neighboring processes, thus, the actions of UTR_K are too coarse grained for the concrete execution model. We refine UTR_K into C by commenting ("//") out the clauses in UTR_K that violated the restrictions of the concrete execution model.

$$
\begin{array}{rcll}
c.0 = c.N & \longrightarrow & c.0 := c.0 \oplus 1 & //\ [c.1 \neq c.0] \\
c.j \neq c.(j-1) & \longrightarrow & c.j = c.(j-1) & \\
 & & \multicolumn{2}{l}{//\ [if\ (j \neq N)\ then\ (c.(j+1) \neq c.j)\ else\ c.0 = c.N]}
\end{array}
$$

In the legitimate states of C the conditions in the comments are satisfied by the computations of C. However, in every state C cannot satisfy these conditions since the concrete system model is more restrictive than the abstract. In the illegitimate states, where these conditions might not be satisfied, computations of C might correspond to compressed forms of computations of UTR_K. Consider the following transition of the concrete.



Starting from a state where all $p0$, $p1$, $p2$ have tokens, in one transition a state where only $p0$ has a token is reached. This corresponds to a compression of the following transitions of UTR_K:



Lemma 17. $[C \preceq UTR_K]$.

Proof. Any compression performed by C only results in a token loss and C cannot perform any compressions when the token-ring contains less than two tokens. Since there are finite number of tokens to begin with, and since process actions do not create new tokens (they just propagate the existing tokens), C can do only a finite number of compressions. Thus, C satisfies stability constraint.

In UTR, and hence in UTR_K, starting from a state with k (s.t., $k > 0$) tokens, any state with l (s.t., $k \geq l > 0$) tokens is reachable. Thus, any computation of C can be written as a compression of some computation of UTR_K, and hence, C is a convergence refinement of UTR_K. $\square$

Theorem 18. $C \;[]\; W1' \;[]\; W2'$ is stabilizing.

Proof. Follows from Theorem 3, Lemma 17 and Lemma 16. $\square$

Therefore, our stabilizing unidirectional token ring implementation ($C \;[]\; W1' \;[]\; W2'$) is as follows.

$$\boxed{\begin{array}{rcl} c.0 = c.N \;\wedge\; lastval.0 = c.N & \longrightarrow & c.0 := c.0 \oplus 1; lastval.0 := c.0 \\ c.j \neq c.(j-1) & \longrightarrow & c.j = c.(j-1) \end{array}}$$

Since $lastval.0$ is always equal to $c.0$, it can be omitted. As a result, we get Dijkstra's K-state token-ring system:

$$\boxed{\begin{array}{rcl} c.0 = c.N & \longrightarrow & c.0 := c.0 \oplus 1 \\ c.j \neq c.(j-1) & \longrightarrow & c.j = c.(j-1) \end{array}}$$

where $\oplus$ denotes addition operation under modulo K, s.t. $(K \geq N + 1)$.