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 type 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, abstract token-ring systems that are not stabilizing, and then add an abstract "wrapper" to the systems so as to achieve stabilization. The system and the wrapper are then refined to obtain a concrete token-ring system, while preserving stabilization. In fact, the two are refined independently, which demonstrates 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, algorithms, token-ring, graybox design

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 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 the minimum stored bid. The bidding server is tolerant to the corruption of a single stored bid in that it satisfies the specification for $(k-1)$ out of best-k bids.

Consider now a sorted-list implementation of the 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 in the absence of faults, 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 in-

[*]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.

troduced 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 practice refinement tools will preserve selected types of tolerance with respect to selected classes of faults. In this paper, we focus our attention on preserving the tolerance property of stabilization in the face of transient faults.

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 states and final states [if there are any]) 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 [5] is preserved: If A is stabilizing then any convergence refinement, C, of A is also stabilizing.

Our second contribution is to show that convergence refinement enables abstract (or specification-based) design of stabilization, in contrast to traditional methods which enable only concrete (or implementation-based) design. In other words, it enables a non-stabilizing implementation C to be made stabilizing without knowing the implementation details of C but knowing only an abstract specification A that C satisfies (we call this "graybox" design [1]). More specifically, given C that is a convergence refinement of A, first stabilization of A is designed by devising an abstract wrapper W for A. Stabilization of C is then achieved by adding to C any convergence refinement of W; the refined wrapper is oblivious to the implementation details of C.

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 reusability. It moreover offers a design alternative in closed-source situations, where implementation details are not available. In such situations, treating the system as a blackbox can yield a high-cost design. Exploiting a specification may therefore be warranted, and convergence refinement is useful in this process.

Finally, our third contribution is a formal derivation of Dijkstra's [5] little-understood 3-state stabilizing token-ring system –as well as his 4-state system– based on convergence refinements. More specifically, we derive Dijkstra's token-ring systems starting from simple abstract 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 an abstract wrapper component. In the second step, we independently refine the non-stabilizing system and its wrapper. The composition of the resulting concrete system and wrapper 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 and Ghosh [6] 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.

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 system 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. For reasons of space we refer the reader to the full version of this paper [4] for a derivation of Dijkstra's K-state protocol from an abstract unidirectional token-ring system.

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 states such that every state is related to the subsequent one with a transition in T, 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 state sequence c is a *convergence isomorphism* of a state sequence 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

As we showed in [1], 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.

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

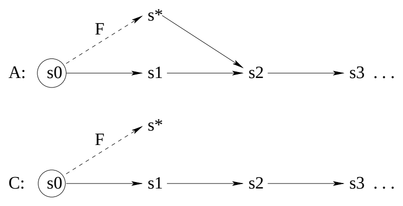


Figure 1. $[C \subseteq A]_{init}$

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 an example of model refinements in Section 3.1 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, A_{init}, 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'. □

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

Proof. The result follows from Lemma 2 and Theorem 1. □

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.) The result follows from the above via Theorem 1. □

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. Given $[W' \preceq W]$ and $(A \;\Box\; W)$ is stabilizing to A, we get $(A \;\Box\; W')$ is stabilizing to A from Lemma 4. Since $[C \preceq A]$, from Lemma 2 we get $[(C \;\Box\; W') \preceq (A \;\Box\; W')]$. The result follows via Theorem 1. □

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.

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.

[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 $R.i$.

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.

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

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

$$I1 \;\equiv\; (\exists j :: \uparrow t.j \;\vee\; \downarrow t.j)$$

$$I2 \;\equiv\; (\forall j,k :: ((\uparrow t.j \;\wedge\; \uparrow t.k) \;\vee\; (\uparrow t.j \;\wedge\; \downarrow t.k) \;\vee\; (\downarrow t.j \;\wedge\; \downarrow t.k)) \Rightarrow j = k)$$

$$I3 \;\equiv\; (\forall j :: \neg(\uparrow t.j \;\wedge\; \downarrow t.j))$$

$$I4 \;\equiv\; (\forall j :: \uparrow t.j \text{ and } \downarrow \text{t.j occur with equal frequency})$$

$I1$ 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 and left neighbors in one atomic step. The concrete system model is more restrictive: j can read its state and the states of its right and left 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 $(\forall W :: (BTR \ \square \ W)$ is stabilizing to $BTR \ \Rightarrow \ (C \ \square \ W)$ 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 $I1$ and $(I2 \ \wedge \ I3)$ respectively. We do not need a wrapper to correct $I4$ because $I4$ follows from BTR after $I1 \ \wedge \ I2 \ \wedge \ I3$ is established.

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

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

$W2$ guarantees 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.

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

Theorem 6 $(BTR \ \square \ W1 \ \square \ W2)$ 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.

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

We also map $up.N = false$ and $up.0 = true$. The actions for BTR_4 follow from BTR via the mapping:

```
$c.N \neq c.(N-1) \ \wedge \ up.(N-1)$
        $\longrightarrow \ c.N := c.(N-1); \ up.(N-1) := true$
$c.0 = c.1 \ \wedge \ \neg up.1$
        $\longrightarrow \ c.0 := \neg c.1; \ up.1 := false$
$c.j \neq c.(j-1) \ \wedge \ up.(j-1) \ \wedge \ \neg up.j$
        $\longrightarrow \ c.j := c.(j-1); \ up.j := true;$
            $c.(j+1) := \neg c.j; \ up.(j+1) := false$
$c.j = c.(j+1) \ \wedge \ \neg up.(j+1) \ \wedge \ up.j$
        $\longrightarrow \ up.j := false;$
            $c.(j-1) := c.j; \ up.(j-1) := true$
```

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 $W1'$:

```
$(\forall j : j \neq N : up.j) \ \wedge \ c.(N-1) \neq c.N$
        $\longrightarrow \ c.N := \neg c.(N-1); \ up.(N-1) := true$
```

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.

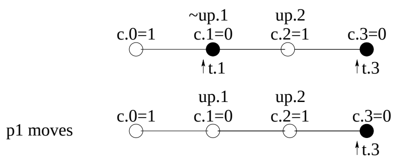
$W2$ states that if a process j has ${\uparrow}t.j$ and ${\downarrow}t.j$ it will drop both of them. $W2'$ is also trivial since using the mapping we get $({\uparrow}t.j \ \wedge \ {\downarrow}t.j \equiv false)$. That is, in BTR_4 j cannot possess ${\uparrow}t.j$ and ${\downarrow}t.j$ at the same time.

4.2 Refinement of BTR_4

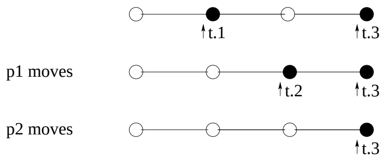
The concrete execution model does not allow writing to the states of the neighboring processes, thus, the actions of BTR_4 are too coarse grained for the concrete execution model. We refine BTR_4 into $C1$ by commenting ("//") out the clauses in BTR_4 that violate the restrictions of the concrete execution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

```
$c.N \neq c.(N-1) \ \wedge \ up.(N-1)$
        $\longrightarrow \ c.N := c.(N-1); \ //(up.(N-1))$
$c.0 = c.1 \ \wedge \ \neg up.1$
        $\longrightarrow \ c.0 := \neg c.0; \ //(\neg up.1)$
$c.j \neq c.(j-1) \ \wedge \ up.(j-1) \ \wedge \ \neg up.j$
        $\longrightarrow \ c.j := c.(j-1); up.j := true;$
            $//(c.(j+1) \neq c.j \ \wedge \ \neg up.(j+1))$
$c.j = c.(j+1) \ \wedge \ \neg up.(j+1) \ \wedge \ up.j$
        $\longrightarrow \ up.j := false;$
            $//(c.(j-1) = c.j \ \wedge \ up.(j-1))$
```

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 7 $[C1 \preceq BTR]$.
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 8 $C1 \; [] \; W1' \; [] \; W2'$ 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 7, and Theorem 6. $\square$

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

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

Interested reader may note that $(C1 \; [] \; W1' \; [] \; W2')$ can further be optimized (by relaxing the guards of the first and third actions) to Dijkstra's 4-state stabilizing token-ring system below.

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

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 $W1'$:

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

$W1'$ 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:

$$\begin{array}{lcl} c.(N-1) = c.0 \wedge & & \\ c.N \neq c.(N-1) \oplus 1 & \longrightarrow & c.N := c.(N-1) \ominus 1 \end{array}$$

$W1''$ 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:

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

Lemma 9 $[BTR_3 \;\Box\; W1'' \;\Box\; W2']$ is stabilizing to BTR. □

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.

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

It is possible to merge $W1''$ with the guard of the first action in $C2$ since $W1''$ ensures that "$(c.(N-1) = c.0 \;\; \wedge \;\; c.N \neq c.(N-1) \oplus 1) \;\Rightarrow\; c.(N-1) = c.N \oplus 1$" and BTR_3 satisfies in the absence of faults that "$(c.(N-1) = c.0 \;\; \wedge \;\; c.N \neq c.(N-1) \oplus 1) \equiv c.(N-1) = c.N \oplus 1$". Note also that we can embed $W2'$ in the third and fourth actions of $C2$. Thus, the resulting system $(C2 \;\Box\; W1'' \;\Box\; W2')$ is as follows.

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

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

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

Lemma 10 $[C2 \;\Box\; W1'' \;\Box\; W2' \preceq BTR_3 \;\Box\; W1'' \;\Box\; W2']$.
Proof. Similar to Lemma 7. □

Theorem 11 $C2 \;\Box\; W1'' \;\Box\; W2'$ is stabilizing to BTR.
Proof. Follows from Theorem 1, Lemma 9, and Lemma 10. □

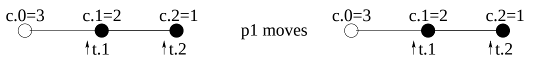
6 A new 3-state stabilizing token-ring

In Section 5.2 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.

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

Recall that in $C2$ we implemented $c.(j-1) \neq c.j \oplus 1$ and $c.(j+1) \neq c.j \oplus 1$ by setting $c.j := c.(j-1)$, $c.j := c.(j+1)$ respectively and due to the limitations of the execution model commented out the predicates $c.(j+1) = c.j \ominus 1$ and $c.(j-1) = c.j \ominus 1$. On the contrary, in $C3$, we implement $c.(j+1) = c.j \ominus 1$ and $c.(j-1) = c.j \ominus 1$ by setting $c.j := c.(j+1) \oplus 1$, $c.j := c.(j-1) \oplus 1$ respectively and due to the limitations of the execution model comment out $c.(j-1) \neq c.j \oplus 1$ and $c.(j+1) \neq c.j \oplus 1$.

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 12 is trivially satisfied.

Lemma 12 $[C3 \preceq BTR]$. $\Box$

Next, we consider refinement of the wrappers $W1$ and $W2$ for $C3$. Even though we have independently refined BTR_3 into $C3$ and $W1'$ into $W1''$, the theory of convergence refinement enables us to conclude that $C3 \;\Box\; W1'' \;\Box\; W2'$ preserves the stabilization property of $BTR_3 \;\Box\; W1' \;\Box\; W2'$, and hence, that of $BTR \;\Box\; W1 \;\Box\; W2$. That is, since convergence refinement is amenable for graybox design of stabilization, the same wrappers $W1''$ and $W2'$ that we developed in Section 5.1 are applicable without any modification for $C3$. Thus, our new 3-state stabilizing system is as follows.

```
$c.(N-1) = c.0 \;\wedge$
$c.(N-1) \oplus 1 \neq c.N$   $\longrightarrow$  $c.N := c.(N-1) \oplus 1$
      $c.1 = c.0 \oplus 1$    $\longrightarrow$  $c.0 := c.1 \oplus 1$
$c.(j-1) = c.j \oplus 1$      $\longrightarrow$  if $(c.(j-1) = c.(j+1))$
                                  then $c.j := c.(j-1)$
                                  else $c.j := c.(j+1) \oplus 1$
$c.(j+1) = c.j \oplus 1$      $\longrightarrow$  if $(c.(j-1) = c.(j+1))$
                                  then $c.j := c.(j+1)$
                                  else $c.j := c.(j-1) \oplus 1$
```

Theorem 13 $C3 \;\Box\; W1'' \;\Box\; W2'$ is stabilizing to BTR.

Proof. Follows from Theorem 3, Lemma 9, and Lemma 12. $\Box$

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 and similarly deletes $\downarrow t.j$ when $\downarrow t.(j-1)$ is also true. The resulting system is as follows.

```
$c.(N-1) = c.0 \;\wedge$
$c.(N-1) \oplus 1 \neq c.N$  $\longrightarrow$  $c.N := c.(N-1) \oplus 1$
      $c.1 = c.0 \oplus 1$   $\longrightarrow$  $c.0 := c.1 \oplus 1$
$c.(j-1) = c.j \oplus 1$     $\longrightarrow$  if $(c.(j-1) = c.(j+1))$
                                 then $c.j := c.(j-1)$
                                 else if $(c.j = c.(j+1) \oplus 1)$
                                     then $c.j := c.(j-1)$
                                     else $c.j := c.(j+1) \oplus 1$
$c.(j+1) = c.j \oplus 1$     $\longrightarrow$  if $(c.(j-1) = c.(j+1))$
                                 then $c.j := c.(j+1)$
                                 else if $(c.j = c.(j-1) \oplus 1)$
                                     then $c.j := c.(j+1)$
                                     else $c.j := c.(j-1) \oplus 1$
```

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:

```
$c.(N-1) = c.0 \;\wedge$
$c.(N-1) \oplus 1 \neq c.N$   $\longrightarrow$   $c.N := c.(N-1) \oplus 1$
       $c.1 = c.0 \oplus 1$   $\longrightarrow$   $c.0 := c.1 \oplus 1$
  $c.(j-1) = c.j \oplus 1$    $\longrightarrow$   $c.j := c.(j-1)$
  $c.(j+1) = c.j \oplus 1$    $\longrightarrow$   $c.j := c.(j+1)$
```

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 and verification 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 $[\Sigma^* A \; \Box \; W \subseteq \Sigma^*(A \; \Box \; W)]$, 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 [3, 10], whereas here we have studied them in the more general context of computation-model refinement. Also, the fault-tolerance preserving refinements presented in [3, 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.

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 stabilizing token-ring implementations (i.e., 3-state and 4-state token-ring systems) from an abstract stabilizing token-ring system.

In contrast to traditional designs of stabilizing systems that are implementation-based (whitebox), we have demonstrated specification-based (graybox) design of stabilization via convergence refinement. Graybox approach offers the promise of scalable, reusable and low-cost design of stabilization since any wrapper component designed to achieve stabilization for an abstract specification is reusable for achieving stabilization for any convergence refinement of the specification.

In future work, we will focus on devising refinement tools and methodologies that accommodate common classes of faults and feasible types of fault-tolerances.

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] J. Beauquier, A. K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. *International Symposium on Distributed Computing*, pages 223–237, 2000.

[4] M. Demirbas and A. Arora. Convergence refinement. *Technical report OSU-CISRC-3/02-TR06, Ohio State University*, March 2002.

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

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

[7] W. 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. *13th International Symposium on Distributed Computing(DISC)*, 1999.

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