Specification-based Design of Self-Stabilization

Murat Demirbas, *Member, IEEE,* and Anish Arora, *Member, IEEE*

Abstract—Research in system stabilization has traditionally relied on the availability of a complete system implementation. As such, it would appear that the scalability and reusability of stabilization is limited in practice. To redress this perception, in this paper, we show for the first time that system stabilization may be designed knowing only the system specification but not the system implementation. We refer to stabilization designed thus as *specification-based design of stabilization* and identify "local everywhere specifications" and "convergence refinements" as being amenable to the specification-based design of stabilization. Using our approach, we present the design of Dijkstra's 4-state stabilizing token-ring system starting from an abstract fault-intolerant token-ring system. We also present an illustration of automated design of specification-based stabilization on a 3-state token-ring system.

Index Terms—Self-stabilization, Fault-tolerance preserving refinements, Distributed systems.

1 Introduction

Self-stabilization research [7], [8], [10], [11], [12] focuses on how systems deal with arbitrary state corruption. A stabilizing system is such that every computation of the system, upon starting from an arbitrary state, eventually reaches a state from where the computation is "correct". The motivation for considering arbitrary state-corruption is to account for the unanticipated faults that results from complex interactions of faults and system components. Self-stabilization offers a formal state-based verification and design technique that shuns case-by-case analysis of faults and recovery in favor of a uniform mechanism. To this end, research in self-stabilization has traditionally relied on the availability of a complete system implementation. The standard approach uses knowledge of all implementation variables and actions to exhibit an "invariant" condition such that if the system is properly initialized then the invariant is always satisfied and if the system is placed in an arbitrary state then continued execution of the system eventually reaches a state from where the invariant is always satisfied.

The apparently intimate connection between stabilization and the details of implementation has raised the following serious concerns: (1) Stabilization is not feasible for many applications whose implementation details are not available, for instance, closed-source applications. (2) Even if implementation details are available, stabilization is not scalable as the complexity of calculating the invariant of large implementations may be exorbitant. (3) Stabilization lacks reusability since it is specific to a particular implementation.

Towards addressing these concerns, in this paper, we show for the first time that system stabilization can be achieved without knowledge of implementation details. We eschew "whitebox" knowledge—of system implementation—in favor of "graybox" knowledge—of system specification—for the design of stabilization. Since specifications are typically more succinct than implementations, specification-based design of fault-tolerance offers the promise of scalability when the design effort for adding fault-tolerance is proportional to the size of the specification. Also, since specifications admit multiple implementations and since system components are often reused, specification-based design of fault-tolerance offers the promise of reusability. Finally, for closed-source situations where exploiting a specification is warranted, specification-based approach allows the design of efficient fault-tolerance in contrast to a blackbox design.

Given a high-level system specification A, the specification-based approach is to design a tolerance wrapper W such that adding W to A yields a fault-tolerant system. The goal is to ensure that for any low-level refinement (implementation) C of A adding a low-level refinement W' of W would also yield a fault-tolerant system.

Note that since the refinements from A to C and W to W' can be done independently, specification-based design enables a posteriori or dynamic addition of fault-tolerance. That is, given a concrete implementation C, it is possible to add fault-tolerance to C as follows:

- First, design an abstract (high-level) tolerance wrapper W using solely an abstract specification A of C, and then
- add a concrete (low-level) refinement W' of W to C.

The goal of specification-based fault-tolerance is not readily achieved for all refinements. The refinements we need for achieving specification-based fault-tolerance should not only preserve fault-tolerance but also have nice composability features so that the refinements from A to C and W to W' can be done indepen-

- *M. Demirbas is with the Department of Computer Science and Engineering, University at Buffalo, SUNY, Buffalo, NY, 14260.*
  E-mail: demirbas@cse.buffalo.edu
- *A. Arora is with the Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, 43210.*
  E-mail: anish@cse.ohio-state.edu

dently. In this paper we present special classes of refinement, “everywhere refinements”, “local-everywhere refinements”, and “convergence refinements”, that enable specification-based design of stabilization. These refinements ensure that if A composed with W is fault-tolerant, then for any everywhere or convergence refinement C of A adding an everywhere or convergence refinement W' of W would also yield a fault-tolerant system. Using these refinements, in this paper, we present the design of Dijkstra’s 4-state and 3-state stabilizing token-ring systems as illustrations.

Outline of the rest of the paper. In Section 2, we give preliminaries. In Section 3 we show that everywhere and convergence refinements are stabilization preserving and in Section 4 that they are amenable for specification-based design of stabilization. In Section 5, using specification-based approach we present a derivation of Dijkstra’s 4-state stabilizing token-ring algorithm as a refinement of a simple, abstract token-ring algorithm. We present related work in Section 6 and make concluding remarks in Section 7. For reasons of space, we relegate most of the proofs and an illustration of automated synthesis of specification-based stabilization on the token-ring example to the supporting material associated with this article.

2 Preliminaries

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 convenience in our definitions and theorems, we pretend that the specification and the implementation use the same state space. In general, 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. We handle this 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. [1]All definitions and theorems in this paper are readily extended with respect to the definition of the abstraction function. Moreover, in our illustrations of specification-based design method in Sections 5 and 9, the concrete systems use a different state space than the abstract.

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* [2] 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.

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. Here, 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.

3 Stabilization Preserving Refinements

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.

1. Note that our abstraction function allows C to introduce irrelevant variables for implementing a feature that is orthogonal to the functionality of A (e.g., a graphical user interface at C). If only the irrelevant variables differ for two states c_1 and c_2 of C, then our abstraction functions will map c_1 and c_2 to correspond the same state in A.

```
 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 introduced in C create additional challenges for the fault-tolerance of C. That is,

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

For a more abstract 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 program 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.

$$\begin{tikzcd}
 & s* \arrow[drr] & & & \\
A{:}\ s0 \arrow[ur, dashed, "F"] \arrow[r] & s1 \arrow[r] & s2 \arrow[r] & s3 \ \ldots
\end{tikzcd}$$

$$\begin{tikzcd}
 & s* & & & \\
C{:}\ s0 \arrow[ur, dashed, "F"] \arrow[r] & s1 \arrow[r] & s2 \arrow[r] & s3 \ \ldots
\end{tikzcd}$$

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

We are therefore motivated to use suitable stabilization preserving refinements in order to enable a specification-based design of stabilization. Next, we present the stabilization preserving properties of everywhere and convergence refinements.

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

Theorem 0 follows immediately from the definitions of stabilization and everywhere refinement.

The requirements for everywhere refinements are sometimes too restrictive. 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. One such example is model refinements 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. $\Box$

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).

4 Specification-Based Design of Stabilization

Here we focus on the 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 everywhere or 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 ▯ (pronounced "box") which denotes the union of automata.

Next, we prove that everywhere and convergence refinements enable specification-based design of stabilization, respectively in Sections 4.1 and 4.2. In Section 4.3 we prove composition theorems about everywhere and convergence refinements. Due to reasons of space we relegate most of the proofs to an online supporting material section.

4.1 Everywhere refinements

Lemma 2 .
$([C \subseteq A] \wedge [W' \subseteq W]) \Rightarrow [(C \,\Box\, W') \subseteq (A \,\Box\, W)]$
From the lemma, our goal follows trivially:

Theorem 3 (Stabilization via everywhere refinements). If $[C \subseteq A]$, $A \,\Box\, W$ is stabilizing to A, and $[W' \subseteq W]$ then $C \,\Box\, W'$ is stabilizing to A. □

Recall that W' and W are designed based only on the knowledge of A and not of C in the specification-based design approach. This results in the reusability of the wrapper for any everywhere implementation of A.

We now focus our attention on distributed systems. The task of verifying everywhere implementation is difficult for distributed implementations, because global state is not available for instantaneous access, all possible interleavings of the steps of multiple processes have to be accounted for, and global invariants are hard to calculate. For effective specification-based design of stabilization of distributed systems, we therefore restrict our consideration to a subclass of everywhere specifications, namely *local everywhere specifications*.

A local everywhere specification A is one that is decomposable into local specifications, one for every process i; i.e., $A = (\Box\, i :: \; A_i)$. Hence, given a distributed implementation $C = (\Box\, i :: \; C_i)$ it suffices to verify that $[C_i \subseteq A_i]$ for each process i. Verifying these "local implementations" is easier than verifying $[C \subseteq A]$ as the former depends only on the local state of each process and is independent of the environment of each process, thereby avoids the necessity of reasoning about the states of other processes.
Let $A = (\Box\, i :: \; A_i)$, $C = (\Box\, i :: \; C_i)$, $W = (\Box\, i :: \; W_i)$, and $W' = (\Box\, i :: \; W'_i)$.
Lemma 4 . $(\forall i :: [C_i \subseteq A_i]) \Rightarrow [C \subseteq A]$
Lemma 5 . $((\forall i :: [C_i \subseteq A_i])$
$\wedge\ (\forall i :: [W'_i \subseteq W_i])) \Rightarrow [(C \,\Box\, W') \subseteq (A \,\Box\, W)]$
From Lemma 5 and Theorem 3 , we have
Theorem 6 (Stabilization via local everywhere refinements).
If $(\forall i :: [C_i \subseteq A_i])$, $(\forall i :: [W'_i \subseteq W_i])$, and $A \,\Box\, W$ is stabilizing to A, then $C \,\Box\, W'$ is stabilizing to A. □

Theorem 6 is the formal statement of the amenability of local everywhere specifications for specification-based design of stabilization. Again, it is tacit that W'_i and W_i are designed based only on the knowledge of A_i and not of C_i.

4.2 Convergence refinements

Lemma 7 If $[C \preceq A]$ and $(A \,\Box\, W)$ is stabilizing to A then $[(C \,\Box\, W) \preceq (A \,\Box\, W)]$.

Theorem 8 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 7 and Theorem 1 . □

Theorem 8 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 9 , we prove a more general result in Theorem 10 .

Lemma 9 If $[W' \preceq W]$ and $(A \,\Box\, W)$ is stabilizing to A then $(A \,\Box\, W')$ is stabilizing to A.

Theorem 10 If $[C \preceq A]$ and $(A \,\Box\, W)$ is stabilizing to A, then $(\forall W' : [W' \preceq W] : (C \,\Box\, W')$ is stabilizing to $A)$.

Theorem 10 is the formal statement of the amenability of convergence refinements for specification-based design of stabilization: If W provides stabilization to A, then any convergence refinement W' of W provides stabilization to every convergence refinement C of A.

4.3 Compositionality of everywhere and convergence refinements

Here we prove composition theorems about everywhere and convergence refinements. In contrast to previous work on composition of fault-tolerance, these theorems are applicable for the general case of composition and are not limited to special cases such as layering composition [8].

Composition theorem for everywhere refinements. Composition theorem for everywhere refinements is simple:

Theorem 11 *(Composition of everywhere refinements).*

$$\begin{aligned} & [C \subseteq A] \;\wedge\; [D \subseteq B] \;\wedge\; [I' \subseteq I] \\ \Rightarrow\; & [C \,\Box\, D \,\Box\, I' \;\subseteq\; A \,\Box\, B \,\Box\, I] \end{aligned}$$

□

Everywhere refinements are very easy to compose and thus specification-based approach for composition of fault-tolerance readily applies for everywhere refinements.

Composition theorem for convergence refinements. Convergence refinements are weaker than everywhere refinements and thus they do not compose as cleanly as everywhere refinements.

(Example): $[C \preceq A] \Rightarrow [C \,\Box\, B \preceq A \,\Box\, B]$ is not a theorem for convergence refinements. Consider the following counterexample.

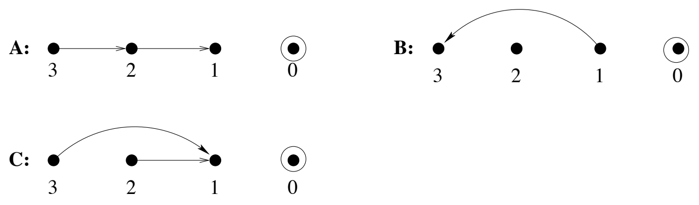


Fig. 2. $[C \preceq A]$

Let A, C, and B be as in Figure 2. State 0 is the initial state for A, C and B, thus $[C \subseteq A]_{init}$ and $[C \preceq A]$.

As seen in Figure 3, $a = 3{-}2{-}1{-}3{-}2{-}1{-}3{-}2{-}1...$ is a computation of $A \,\Box\, B$, and $c = 3{-}1{-}3{-}1{-}3{-}1...$

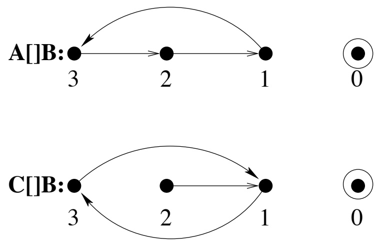


Fig. 3. $(C \;[]\; B)$ is not a convergence refinement of $(A \;[]\; B)$

is a computation of $C \;[]\; B$. However, c drops infinitely many states from a and thus $C \;[]\; B$ is not a convergence refinement of $A \;[]\; B$.
(End of example).

Therefore, in order to prove a composition theorem we need to prove loop-free behavior of the abstract composition outside its invariant.

Lemma 12 .

$$\begin{array}{ll} & [C \preceq A] \;\wedge\; (A \;[]\; B \;[]\; I) \textit{ is stabilizing to } A \\ \Rightarrow & [C \;[]\; B \;[]\; I \;\preceq\; A \;[]\; B \;[]\; I] \end{array}$$

Lemma 13 .

$$\begin{array}{ll} & [D \preceq B] \;\wedge\; (A \;[]\; B \;[]\; I) \textit{ is stabilizing to } A \\ \Rightarrow & [A \;[]\; D \;[]\; I \;\preceq\; A \;[]\; B \;[]\; I] \end{array}$$

Theorem 14 *(Composition of convergence refinements).*

$$\begin{array}{ll} & [C \preceq A] \;\wedge\; [D \preceq B] \;\wedge\; [I' \preceq I] \\ & \wedge\; (A \;[]\; B \;[]\; I) \textit{ is stabilizing to } A \\ \Rightarrow & [C \;[]\; D \;[]\; I' \;\preceq\; A \;[]\; B \;[]\; I] \end{array}$$

Proof: Follows from Lemma 12 , Lemma 13 (applied twice), and transitivity of $[\;\preceq\;]$. □

Our composition theorem is very general and is defined in an asymmetric manner. For B to be a stand-alone component rather than a tolerance wrapper, in the abstract one should also prove that $(A \;[]\; B \;[]\; I)$ is stabilizing to invariant of B. This way we ensure that starting from the initialized states, both component do useful work.

5 Application on Token-Ring Stabilization Problem

In Section 5.1, we present the abstract token-ring system and show how to achieve stabilization to the abstract. Then, we derive by way of convergence refinements of the abstract bidirectional token-ring system Dijkstra's 4-state system in Section 5.2.

5.1 Stabilizing the Abstract 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.

5.1.1 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.

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

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

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

$I1$ 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 $(\forall W :: (BTR \;[]\; W)$ is stabilizing to $BTR \;\Rightarrow\; (C \;[]\; W)$ is stabilizing to $BTR)$.

From Theorem 10 , it follows that any concrete system C that satisfies $[C \preceq BTR]$ is a solution to the BTR problem.

5.1.2 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:

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

$W2$ 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.

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

Theorem 15 $(BTR \;▯\; W1 \;▯\; W2)$ is stabilizing to BTR. □

5.2 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 $up.N = \mathit{false}$ and $up.0 = \mathit{true}$. The actions for BTR_4 follow from BTR via the mapping:

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

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

5.2.1 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 $W1'$:

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

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.

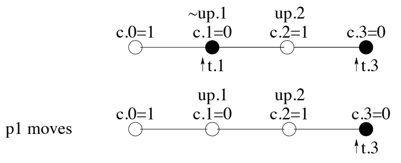
$W2$ states that if a process j has ${\uparrow}t.j$ and ${\downarrow}t.j$ it will drop both of them. $W2'$ is also trivial since using the mapping we get $({\uparrow}t.j \;\wedge\; {\downarrow}t.j \equiv \mathit{false})$. That is, in BTR_4 j cannot possess ${\uparrow}t.j$ and ${\downarrow}t.j$ at the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

5.2.2 Refinement of BTR_4

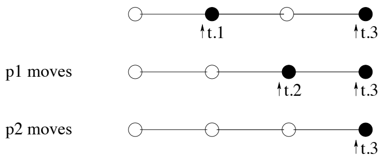
The concrete execution model does not allow writing to the states of the neighboring processes, thus, the actions of BTR_4 are too coarse grained for the concrete execution model. We refine BTR_4 into $C1$ by commenting ("//") out the clauses in BTR_4 that violate the restrictions of the concrete execution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

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

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 16 $[C1 \;\preceq\; BTR]$.

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 17 $C1 \;\square\; W1' \;\square\; W2'$ is stabilizing to BTR.
Proof. Since $[W1' \subseteq W1]$ and $[W2' \subseteq W2]$, we have $[W1' \;\square\; W2' \subseteq W1 \;\square\; W2]$. The result then follows from Theorem 10 , Lemma 16 , and Theorem 15 . $\square$

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

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

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

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

6 Related Work

Research in stabilization [8], [10], [11], [12] has traditionally relied on the availability of a complete system implementation. The standard approach to reasoning uses knowledge of all implementation variables and actions to exhibit an "invariant" condition such that if the system is properly initialized then the invariant is always satisfied and if the system is placed in an arbitrary state then continued execution of the system eventually reaches a state from where the invariant is always satisfied. Likewise, the generic methods for designing stabilization [1], [3], [13], [20] also assume implementation-specific details as input: [3], [13] assume the availability of the implementation invariant, [1] relies on the knowledge of the implementation actions, and [20] takes as input a "locally checkable" consistency predicate derived from implementation. To the best of our knowledge, our work [2] is the first time that system stabilization is shown to be provable without whitebox knowledge. As one piece of evidence, we offer the following quote due to Varghese [20] (parenthetical comments are ours):

> In fact, the only method we know to *prove* a behavior stabilization result (i.e., stabilization with respect to system specification) is to first prove a corresponding execution stabilization result (i.e., stabilization with respect to system implementation) ...

There has been limited work on blackbox addition of stabilization. Checkpointing and recovery based approaches may be considered as an example of this category. As another example, Awerbuch and Varghese [4] presented a method that transforms any synchronous protocol into a self-stabilizing version for dynamic asynchronous networks. The method adds stabilization in a blackbox manner by re-executing the non-stabilizing algorithm until stabilization is observed in the dynamic network environment.

Next, we discuss related work on fault-tolerance preserving refinements.

Method by Z. Liu and M. Joseph. Liu and Joseph [17] 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 design the tolerance based on the concrete program, while we design our wrappers based on the abstract program.

Method by L. Lamport and S. Merz. In [14], Lamport and Merz claim that there is no need for a special technique for formal specification and verification of fault-tolerance systems, and that refinement of fault-tolerance programs could be achieved using temporal logic of actions (TLA) and a hierarchical proof method. Towards this end, they show how a message-passing Byzantine agreement program (of [15]) can be derived from its high-level specification. The authors claim that little ingenuity is required for proofs of refinements since a hierarchical proof strategy is adopted. However, a considerable amount of ingenuity is still required for coming up with the refinement programs in the first place.

Fault-tolerance preserving atomicity refinements. Fault-tolerance preserving refinements have been studied in the context of atomicity refinement in [5], [19]. The refinements presented in those work are instances of everywhere refinements. In our work we study fault-tolerance preserving refinements in the more general context of computation-model refinement (not just atomicity refinement), and we also present a more general type of fault-tolerance preserving refinement, convergence refinement.

McGuire and Gouda [18] 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. Their framework cannot handle arbitrary state corruptions we considered here, and only allows the following faults: message loss, message ordering, and message duplication. Another self-stabilization preserving compiler is presented in Dolev et. al. [9]. Given a self-stabilizing program written in Abstract State Machine (ASM) language their compiler produces machine code that eventually has the same input-output relation as the original ASM program. Neither of these compilers produce everywhere or convergence refinements, and therefore are not amenable for the compositional refinement approach we proposed.

Semantics of fault-tolerance preserving refinements. Leal [16] 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.

7 Concluding remarks

In this paper, we investigated the specification-based design of system stabilization, which uses only the system specification, towards overcoming drawbacks of the tradional whitebox approach, which uses the system implementation as well. The specification-based design approach offers the potential of adding stabilization in a scalable manner, since specifications grow more slowly than implementations. It also offers the potential of component reuse. Component technologies typically separate the notion of specification (variously called interface or type) from that of implementation. Since reuse occurs more often at the specification level than the implementation level, specification-based design of stabilization is more reusable than stabilization that is particular to an implementation.

General applicability of the method. Although we have limited our discussion of the specification-based design approach to the property of stabilization, the approach is applicable for the design of other dependability properties, for example, masking fault-tolerance, fail-safe fault-tolerance, super-stabilization, and loop-freedom. Our observation that specification-based design of stabilization is not readily achieved for all refinements is likewise true for specification-based design of masking and fail-safe fault-tolerance. Moreover, our observation that local everywhere specifications are amenable to specification-based design of stabilization is also true for specification-based design of masking and fail-safe fault-tolerance.

Our method applies for a rich class of programs whose invariants allow decentralization of the invariant. In fact most programs we are familiar with satisfies this decentralization of the invariant, where it is possible to rewrite a global invariant as a composition of local invariants [8], [20].

Our method also benefits from tool support for finding abstraction function. Given C, the tool presented in [6] automatically produces A such that C is an everywhere refinement of A. By adopting this abstraction function tool, we give a toy example of automated synthesis of specification-based tolerance in the online supporting material section of this paper.

References

[1] Y. Afek and S. Dolev. Local stabilizer. *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 287, 1997.

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

[3] A. Arora, M. G. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. *Journal of High Speed Networks*, 5(3):293–306, 1996.

[4] B. Awerbuch and G. Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols (extended abstract). *FOCS*, pages 258–267, 1991.

[5] 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.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

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

[8] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[9] S. Dolev, Y. Haviv, and M. Sagiv. Self-stabilization preserving compiler. In *International Conference on Self-Stabilizing Systems*, pages 81–95, 2005.

[10] M. Flatebo, A. K. Datta, and S. Ghosh. *Readings in Distributed Computer Systems*, chapter 2: Self-stabilization in distributed systems. IEEE Computer Society Press, 1994.

[11] M. G. Gouda. The triumph and tribulation of system stabilization. *Invited Lecture, Proceedings of 9th International Workshop on Distributed Algorithms, Springer-Verlag*, 972:1–18, November 1995.

[12] T. Herman. Self-stabilization bibliography: Access guide. Chicago Journal of Theoretical Computer Science, Working Paper WP-1, initiated November 1996.

[13] S. Katz and K. Perry. Self-stabilizing extensions for message passing systems. *Distributed Computing*, 7:17–26, 1993.

[14] L. Lamport and S. Merz. Specifying and verifying fault-tolerant systems. *Third Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 863*, pages 41–76, 1994.

[15] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 1982.

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

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

[18] T. M. McGuire. *Correct implementation of network protocols*. PhD thesis, University at Texas at Austin, 2004.

[19] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *13th International Symposium on Distributed Computing (DISC)*, 1999.

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