Graybox Stabilization

Anish Arora[*] *Murat Demirbas*[*] *Sandeep S. Kulkarni*[†]

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

† Department of Computer
Science and Engineering
Michigan State University
East Lansing, Michigan 48824 USA

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 being "graybox" and identify "local everywhere specifications" as being amenable to design of graybox stabilization. We illustrate a method for designing graybox stabilization using timestamp-based distributed mutual exclusion as our example.

Keywords : Specification-based dependability, Stabilization, Dependability wrappers, Local everywhere specifications

---

0

Email: {anish,demirbas}@cis.ohio-state.edu, sandeep@cse.msu.edu ; Tel: +1-614-292-1836 ; Fax: +1-614-292-2911 ;
Web: http://www.cis.ohio-state.edu/{~anish,~demirbas}, http://www.cse.msu.edu/sandeep ;
This work was partially sponsored by NSA Grant MDA904-96-1-0111, NSF Grant NSF-CCR-9972368,
an Ameritech Faculty Fellowship, and a grant from Microsoft Research.

1 Introduction

Research in stabilization [6–9] 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 that system stabilization may 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, 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.

Contributions of the paper. To the best of our knowledge, this is the first time that system stabilization is shown to be provable without whitebox knowledge.

Secondly, in this paper, we introduce the concept of "*local everywhere specifications*", which are amenable to the design of graybox stabilization. Intuitively speaking, these specifications are decomposable into parts each of which must be satisfied by some system process from *all* of its states without relying on its environment (including other processes). By designing a system "wrapper" that achieves stabilization at the level of such a specification, it follows that every system implementation satisfying that specification achieves stabilization by using that wrapper.

Thirdly, we illustrate a method for designing graybox stabilization. Our method is based on the observation that system faults occur at two levels: (1) in the process, or (2) in the interface between processes. We deal with these two levels separately; in particular, we design level (1) wrappers, which handle "intra-process" consistency issues, separately from level (2) wrappers, which resolve the "inter-process" consistency issues.

Our illustration of the method uses timestamp-based distributed mutual exclusion (TME). We present

a local everywhere specification for TME, $Lspec$, and then design a wrapper W for $Lspec$ such that for any implementation that satisfies $Lspec$, wrapping that implementation with W yields stabilization of that implementation. By way of example, we show how Ricart-Agrawala ME and Lamport ME programs satisfy *Lspec*, and hence, W adds stabilization to both of them.

Organization of the paper. In Section 2.1, we show that local everywhere specifications are amenable to design of graybox stabilization. We present our method for designing graybox stabilization in Section 2.2. In Section 3, we present our "local everywhere specification", *Lspec*, for TME. Then, in Section 4, we use our method to design the wrapper W. In Section 5, we show that Ricart-Agrawala's TME program [11], and Lamport's TME program [10] satisfy *Lspec*, and hence W adds stabilization to both of them. We make concluding remarks in Section 6. (For reasons of space, we relegate all proofs to the Appendix.)

2 Graybox Design

In this section, after some preliminary definitions that express both specifications and implementations in uniform terms, we justify why "local everywhere specifications" are amenable to design of graybox stabilization and then present a design method.

Systems: Specifications and Implementations. Let Σ be a state space.

Definition. A *system* S is a set of (possibly infinite) sequences over Σ, with at least one sequence starting from every state in Σ, and a set of initial states chosen from Σ.

We refer to the state sequences of S as its *computations*. Intuitively, the requirement that S contain some computation starting from every Σ state captures that the computations of S are expressed fully, albeit in the absence of faults, S only exhibits computations that start from its initial states. Also, we refer to an abstract system as a *specification*, and to a concrete system as an *implementation*. Henceforth, let C be an implementation and A a specification.

Definition. C *implements* A, denoted $[C \Rightarrow A]_{init}$, iff every computation of C that starts from some initial state of C is a computation of A starting from some initial state of A.

Definition. C *everywhere implements* A, denoted $[C \Rightarrow A]$, iff every computation of C is a computation of A.

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.

Note that the definition of stabilization allows the possibility that A is stabilizing to A.

2.1 Graybox stabilization via local everywhere specifications

Given a specification A, the graybox approach is to design a wrapper W such that adding W to A yields a system that is stabilizing to A. Its goal is to ensure for any C, which implements A, that adding W to C would yield a system that also stabilizes to A. This goal is however not readily achieved for all specifications. In fact, even for specifications A where A is stabilizing to A we may observe:

C implements 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, $[C\ \Rightarrow\ A]_{init}$. 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. $[C\ \Rightarrow\ A]_{init}$

We are therefore led to considering the class of *everywhere specifications*. These specifications demand that their implementations satisfy them from every state in Σ. In other words, an everywhere specification A demands that its implementations C also satisfy $[C\ \Rightarrow\ A]$. We show that these specifications satisfy the goal of graybox design. First, observe that:

$[C\ \Rightarrow\ A]$ and A is stabilizing to A does imply that C is stabilizing to A.

Next, we prove the more general case: If adding a wrapper W to an everywhere specification A yields a system that is stabilizing to A, then adding W to any everywhere implementation C of A also yields a system that is stabilizing to A. Our proof formalizes "addition" of one system to another in terms of the operator $\Box$ (pronounced "box"). The definition of $\Box$ assumes that system computations are "fusion closed": if a system has computations $\langle \alpha, x, \gamma \rangle$ and $\langle \beta, x, \delta \rangle$ then it also has the computations $\langle \alpha, x, \delta \rangle$ and $\langle \beta, x, \gamma \rangle$, where α and β are finite prefixes of computations, γ and δ are suffixes of computations, and x is a state. (The assumption is reasonable since conventional specification and implementation languages are fusion closed. In the sequel, for instance, we use UNITY [3] as our language of specification description and guarded-commands [5] as our language of implementation description—both these languages are fusion-closed.) Now, $C\ \Box\ W$ is the system whose set of computations is the smallest fusion closed set that contains the

computations of C as well as the computations of W, and whose initial states are the common initial states of C and W. Thus, we have:

Lemma 0. $([C \Rightarrow A] \wedge [W' \Rightarrow W]) \Rightarrow [(C \,[]\, W') \Rightarrow (A \,[]\, W)]$ □

From the lemma, our goal follows trivially:

Theorem 1 (Stabilization via everywhere specifications).
If $[C \Rightarrow A]$, $A \,[]\, W$ is stabilizing to A, and $[W' \Rightarrow W]$ then $C \,[]\, W'$ is stabilizing to A. □

Recall that W' and W are designed based only on the knowledge of A and not of C in the graybox 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 graybox 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., $A = ([]\, i :: \ A_i)$ [1]. Hence, given a distributed implementation $C = ([]\, i :: \ C_i)$ it suffices to verify that $[C_i \Rightarrow A_i]$ for each process i. Verifying these "local implementations" is easier than verifying $[C \Rightarrow A]$ as the former depends only on the local state of each process and is independent of the environment of each process (including the other processes).

Let $A = ([]\, i :: \ A_i)$, $C = ([]\, i :: \ C_i)$, $W = ([]\, i :: \ W_i)$, and $W' = ([]\, i : \ W'_i)$.

Lemma 2. $(\forall i :: [C_i \Rightarrow A_i]) \Rightarrow [C \Rightarrow A]$ □

Lemma 3. $((\forall i :: [C_i \Rightarrow A_i]) \wedge (\forall i :: [W'_i \Rightarrow W_i])) \Rightarrow [(C \,[]\, W') \Rightarrow (A \,[]\, W)]$ □

From Lemma 3 and Theorem 1, we have

Theorem 4 (Stabilization via local everywhere specifications).
If $(\forall i :: [C_i \Rightarrow A_i])$, $(\forall i :: [W'_i \Rightarrow W_i])$, and $A \,[]\, W$ is stabilizing to A, then $C \,[]\, W'$ is stabilizing to A. □

Theorem 4 is the formal statement of the amenability of local everywhere specifications for graybox

[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 [5].

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.

2.2 Graybox Design Method

Although Theorem 4 clarifies the role of local everywhere specifications, it leaves open the question of how to design wrappers $(\Box\, i\ ::\ W_i)$ that render $(\Box\, i\ ::\ A_i)$ stabilizing. For instance, designing W_i for each i such that $(A_i\ \Box\ W_i)$ is stabilizing to A_i does not always imply that $(\Box\, i\ ::\ A_i\ \Box\ W_i)$ is stabilizing to $(\Box\, i\ ::\ A_i)$; even though each process i may be internally consistent due to W_i, the processes may be mutually inconsistent. Moreover, W_i that renders A_i stabilizing may interfere with the wrappers of other processes and hence with their stabilization.

A method for designing graybox stabilization may be based on the following observation: In any system that consists of multiple processes, faults occur at two levels: (1) internal to a process, or (2) in the interface between processes. We may deal with these two levels separately, as follows.

For level (1), we need to ensure that the process recovers to an internally consistent state. For this purpose, we design a level (1) wrapper that restores the process to an internally consistent state. Since the new state of the process may not be consistent with the states of the other processes (or may even drive other processes to internally inconsistent states), the level (1) wrapper may also choose to raise an exception to notify other processes (more precisely, their corresponding wrappers) about this state change. Processes can provide one or more exception handlers to deal with these exceptions if need be.

For level (2), which may arise even when two processes are both internally consistent (but they are mutually inconsistent), we design a separate wrapper to reestablish the mutual consistency. The level (2) dependability wrapper optimistically tries to deal with the interface faults (mutual inconsistencies) by assuming that the processes are in internally consistent states. Should the optimistic assumption of internal consistency be invalid, the wrappers that deal with level (1) will eventually resolve the internal inconsistencies.

3 Timestamp-Based Distributed Mutual Exclusion (TME)

Towards applying the graybox method in the context of TME, in this section, we begin by giving a specification of TME in Section 3.1 and then present a local everywhere specification, *Lspec*, for TME in Section 3.2.

3.1 TME problem

System model. The system model for TME problem is message passing; processes communicate solely via message passing on interprocess channels. Execution is asynchronous, i.e., every process executes at its own speed and messages in the channels are subject to arbitrary but finite transmission delays. We assume that the processes are connected.

Faults. The fault model for TME allows messages to be corrupted, lost, or duplicated at any time. Moreover, processes (respectively channels) can be improperly initialized, fail, recover, or their state could be transiently (and arbitrarily) corrupted at any time. Stabilization is desired notwithstanding the occurrence of any finite number of these faults.

TME specification. The specification of TME, TME_Spec, is standard. We express it here in the UNITY specification language [3]. Let p and q be predicates on program states. "$p\ unless\ q$" denotes that if p is $true$ at some point in the computation and q is not, in the next step p remains $true$ or q becomes $true$. "$stable(p)$" is defined as ($p\ unless$ *false*). "q is *invariant*" iff q holds in the initial states and $stable(q)$. "$p\ \mapsto\ q$" (pronounced p *leads to* q) means that if p is $true$ at some point, q will be true (at that point or a later point) in the computation. "$p\ \hookrightarrow\ q$" (pronounced p *leads to always* q) iff $(p\ \mapsto\ q)$ and $stable(q)$. For a detailed discussion of these temporal predicates, we refer the interested reader to [3].

TME_Spec = *ME1* $\wedge$ *ME2* $\wedge$ *ME3*, where *ME1*, *ME2*, and *ME3* are defined as follows.

- *(ME1)* Mutual Exclusion: $(\forall j, k :: e.j\ \wedge\ e.k\ \Rightarrow\ j = k)$
- *(ME2)* Starvation Freedom: $(\forall j :: h.j\ \mapsto\ e.j)$
- *(ME3)* First-Come First-Serve: $(\forall j, k : j \neq k : (h.j\ \wedge\ REQ_j\ \underline{hb}\ REQ_k)\ \mapsto\ ts.(e.j) < ts.(e.k))$

Following the standard terminology, we use $e.j$ (pronounced $eating.j$) to denote that process j is accessing the critical section (CS), and $h.j$ (pronounced $hungry.j$) to denote that j has requested for the critical section but has not yet been granted to access the critical section. We use $t.j$ (pronounced $thinking.j$) to denote that j is neither *eating* nor *hungry*. We use $ts.j$ to denote the timestamp of the most current event at j; for an event f_j, that occurred at j, $ts.f_j$ denotes the timestamp of f_j. REQ_j is a lower bound for the timestamp of the current "request" of j: If j has not issued a request for CS (i.e., $t.j$ holds) then $REQ_j = ts.j$, else REQ_j denotes the timestamp of the current request of j. $j.REQ_k$ denotes j's latest information about REQ_k, that is, $j.REQ_k$ denotes j's local copy of the timestamp of the last request of k.

The problem of designing graybox stabilization for TME. *TME_Spec* is by itself not suitable for graybox stabilization. For one, it is not a local specification. But perhaps even more importantly, it should not be viewed as an everywhere specification: for instance, requiring Mutual Exclusion *(ME1)* in all system

states is unreasonably restrictive, since it is very difficult (if not impossible) to find an implementation that everywhere implements *ME1*.

The problem therefore is to (i) find a local everywhere specification *Lspec* that implements *TME_Spec* from its initial states, and (ii) design a graybox wrapper W, such that for any implementation M that everywhere implements *Lspec*, $M \ [] \ W$ stabilizes to *Lspec*.

3.2 *Lspec*: A local everywhere specification for TME

Before we present *Lspec* we invite our readers to solve (i) by themselves to gain appreciation of the issues involved.

Our *Lspec* consists of three parts: *Client Spec*, *Program Spec*, *Environment Spec*, each of which must be everywhere implemented. We also specify *Init*, the initial states of *Lspec*.

Client Spec.

Structural Spec: $(\forall j :: (h.j \ \vee \ e.j \ \vee \ t.j) \ \wedge \ (h.j \ \Rightarrow \ \neg(e.j \ \vee \ t.j))$
$\wedge \ (e.j \ \Rightarrow \ \neg(h.j \ \vee \ t.j)) \ \wedge \ (t.j \ \Rightarrow \ \neg(h.j \ \vee \ e.j)))$

Flow Spec: $(\forall j :: (h.j \ unless \ e.j) \ \wedge \ (e.j \ unless \ t.j) \ \wedge \ (t.j \ unless \ h.j))$

CS Spec: $(\forall j :: e.j \ \mapsto \ \neg e.j)$

Program Spec.

Request Spec: $(\forall j :: (h.j \ \Rightarrow \ REQ_j = REQ'_j)$
$\wedge \ h.j \ \mapsto \ (\forall k : k \neq j : \text{send}(REQ_j, j, k)))$

Reply Spec: $(\forall j, k : j \neq k : (\ \text{received}(j.REQ_k) \ \wedge \ j.REQ_k \ \underline{lt} \ REQ_j)$
$\mapsto \ \text{send}(REQ_j, j, k))$

CS Entry Spec: $(\forall j :: (e.j \ \Rightarrow \ REQ_j = REQ'_j)$
$\wedge \ (h.j \ \wedge \ (\forall k : k \neq j : REQ_j \ \underline{lt} \ j.REQ_k)) \ \mapsto \ e.j \ )$

CS Release Spec: $(\forall j :: t.j \ \Rightarrow \ REQ_j = ts.j)$

Environment Spec.

Timestamp Spec: ts is from a total domain and $(\forall e, f :: e \ \underline{hb} \ f \ \Rightarrow \ ts.e < ts.f)$ [2]

Communication Spec: Channels are FIFO.

Init. $((\forall j :: t.j \ \wedge \ ts.j = 0 \ \wedge \ REQ_j = 0 \ \wedge \ (\forall k : k \neq j : j.REQ_k = 0))$
$\wedge$ (all channels are empty))

[2] Lamport's [10] happened-before relation, $\underline{hb}$, is the smallest transitive relation that satisfies $e \ \underline{hb} \ f$ for any two events e and f such that (1) e and f are events on the same process and e occurred before f, or (2) e is a send event in one process and f is the corresponding receive event in another process.

Intuitively speaking, *Structural Spec* asserts that for every process j, in any state exactly one of $h.j$, $e.j$, or $t.j$ holds. *Flow Spec* imposes an order on the satisfaction of $h.j$, $e.j$, and $t.j$; e.g., if $h.j$ holds in the current state then in the next state $t.j$ may not become $true$. *CS Spec* states that $e.j$ is transient; if $e.j$ holds in the current state then in some future state $\neg e.j$ holds. *Request Spec* ensures that if $h.j$ holds in the current state, the value of REQ_j is left unchanged (REQ'_j refers to the value of REQ_j in the preceding state), and eventually a *request* message with timestamp REQ_j will be send to all processes. *Reply Spec* guarantees that each time an earlier *request* is received from another process, a *reply* message will eventually be sent to that process. *CS Entry Spec* asserts that if $h.j$ holds in the current state, the value of REQ_j is preserved, and additionally if REQ_j is earlier than all of j's copy of the *requests* of the other processes then j eventually enters CS. *Release Spec* asserts that when $t.j$ holds REQ_j is always set to the timestamp of the most current event in j. *Timestamp Spec* states that the timestamp values should be totally ordered and satisfy the "happened-before", $\underline{hb}$, relation (i.e., ts values do not decrease over time). *Communication Spec* requires all the channels to be FIFO.

It is reasonable to demand that *Client* and *Program* specifications be implemented at each process from any state; in fact, in Section 5 we recall well-known implementations from the literature which everywhere implement these specifications. Likewise, the demand is reasonable for *Timestamp Spec*, since it admits local everywhere implementations, for example, logical clocks [10]. The "less-than" relation, $\underline{lt}$, induces a total order on the timestamps produced by logical clocks. Also, logical clocks satisfy $\underline{hb}$ relation. Formally speaking: $(\forall e_j, f_k :: lc.e_j \ \underline{lt} \ lc.f_k = lc.e_j < lc.f_k \ \ \vee \ \ (lc.e_j = lc.f_k \ \ \wedge \ \ j < k))$, $(\forall e, f :: e \ \underline{hb} \ f \ \ \Rightarrow \ lc.e \ \underline{lt} \ lc.f)$.

Theorem 5 (TME_Spec). Every system M that implements *Lspec* also implements *TME_Spec*.

$$(\forall M :: [M \ \Rightarrow \ \textit{Lspec}]_{init} \ \Rightarrow \ [M \ \Rightarrow \ \textit{TME_Spec}]_{init} \ ) \qquad \square$$

4 Graybox Stabilization Wrapper for TME

Based on *Lspec* described above, we now design a graybox wrapper that ensures stabilization for all everywhere implementations M of *Lspec*.

Intuitively speaking, *Lspec* (more specifically *Program Spec*) captures the internal consistency requirements for TME. Mutual exclusion requirement is observed locally at each process since *CS Entry Spec* requires that a total-ordering of requests is respected while accessing CS. *CS Entry Spec* further requires that the ordering be based on timestamp values of requests, hence, that first-come first-serve requirement is respected locally. Finally, *Request Spec*, *Reply Spec*, and *CS Release Spec* address starvation freedom: *Request Spec* requires the request to be sent to every process; *Reply Spec* requires a reply to be sent for earlier

requests; *CS Release Spec* in conjunction with *Reply Spec* states that a process not requesting for CS should not prevent the interested processes from entering CS. So, for any system M that everywhere implements *Lspec*, the internal consistency requirement of each process (in M) is satisfied at every state. Thus, we do not need to design level (1) wrappers for *Lspec*.

Level (2) wrappers are needed, however, since internal consistency of process states does not imply mutual consistency. For example, due to transient faults there might be more than one process accessing CS at the same time. Or, there may be deadlocks, as illustrated by the following scenario: Suppose processes j and k have both requested for CS. Due to transient faults (e.g., REQ_j and REQ_k are both dropped from the channels) j and k may have mutually inconsistent information: $j.REQ_k \ \underline{lt} \ REQ_j$ and $k.REQ_j \ \underline{lt} \ REQ_k$. Process j cannot enter CS because $j.REQ_k \ \underline{lt} \ REQ_j$. Likewise, k cannot enter CS. As far as the satisfaction of *Lspec* is concerned, j (respectively k) does not have to do anything more; j (resp. k) waits for k (resp. j) to respond to its request message. Therefore, the state of M has a deadlock.

In order to reestablish mutual consistency among the processes, we design a level (2) dependability wrapper W which consists of a wrapper at each process j (i.e., $W = \Box \ j :: \ W_j$). Observe from *Lspec* that mutual inconsistencies between processes j and k may arise only due to $j.REQ_k$ and $k.REQ_j$. These inconsistencies constitute a problem only when j or k is requesting CS (i.e., $h.j$ or $h.k$). Therefore, in order to reestablish mutual consistency between j and k it is sufficient to correct $j.REQ_k$ and $k.REQ_j$ when $h.j$ or $h.k$ holds. Thus, W_j is as follows.

$$W_j :: \ \ h.j \ \longrightarrow \ (\forall k : k \neq j : \ send(REQ_j, j, k))$$

W_j corrects $k.REQ_j$, for all k, by successively sending REQ_j to k as long as $h.j$ holds. $j.REQ_k$ is also corrected by W_j: After $k.REQ_j$ is corrected if $REQ_j \ \underline{lt} \ REQ_k$ holds then from *Reply Spec* it follows that $j.REQ_k$ is eventually set to REQ_k.

We can refine W_j as follows. Let X denote the set of processes k such that $j.REQ_k \ \underline{lt} \ REQ_j$. We require j to correct $k.REQ_j$ (and this in turn corrects $j.REQ_k$ as shown above) only for $k \in X$. For any k such that $k \notin X$ and $h.k$ holds, from *Request Spec* it follows that $j.REQ_k$ (and in turn $k.REQ_j$) will be corrected by W_k. Note that for any k such that $k \notin X$ and $\neg h.k$, there is no need to correct $j.REQ_k$ or $k.REQ_j$. Thus, our refined wrapper W_j is as follows.

$$W_j :: \ \ h.j \ \longrightarrow \ (\forall k : k \neq j \ \ \wedge \ \ j.REQ_k \ \underline{lt} \ REQ_j : \ send(REQ_j, j, k))$$

W_j is a graybox wrapper since it uses only the specification *Lspec* and does not depend on how *Lspec*

is implemented. Next, we prove in Theorem 8 that any system M that everywhere implements *Lspec* can be made stabilizing to *Lspec* by using W. Towards this end, we first prove in Lemma 6 that W does not interfere with *Lspec*, and subsequently in Lemma 7 that *Lspec* composed with W is stabilizing to *Lspec*.

Lemma 6 (Interference freedom). *Lspec* $[]$ W everywhere implements *Lspec*. □

Lemma 7 (Stabilization). *Lspec* $[]$ W is stabilizing to *Lspec*. □

Theorem 8 (Graybox stabilization). For any system M that everywhere implements *Lspec*, $(M \ [] \ W)$ is stabilizing to *Lspec* (and, hence, to *TME_Spec*).

$$(\forall M :: [M \ \Rightarrow \ \mathit{Lspec}] \ \Rightarrow \ (M \ [] \ W) \ \underline{\text{is stabilizing to}} \ \mathit{Lspec}\ )$$

Proof. Follows from Theorem 1 and Lemma 7. □

Implementation of W. It follows from Theorem 4 that any W_j' such that $[W_j' \ \Rightarrow \ W_j]$ is also a dependability wrapper for all M that everywhere implements *Lspec*. Thus, we can relax W_j by sending the request messages periodically instead of sending them successively. To this end, we employ a timeout mechanism at j; request messages are repeated only when timeouts occur.

$$W_j' :: \ (timer.j = 0 \ \wedge \ h.j) \ \longrightarrow \ (\forall k : k \neq j \ \wedge \ j.REQ_k \ \underline{lt} \ REQ_j : \ send(REQ_j, j, k)); \ timer.j = \Delta$$

The domain of $timer.j$ is from 0 to some natural number Δ. Note that the timeout mechanism is just an optimization and does not affect the correctness of the solution. In fact, W_j' is equivalent to W_j when $\Delta = 0$ (i.e., when timeout period is 0). The timeout mechanism can be employed to tune the wrapper to decrease the unnecessary repetitions of the request messages when the system is in the consistent states.

5 Reusability of the Wrapper for TME

In this section, we present two well-known everywhere implementations of *Lspec*, namely the mutual exclusion programs of Ricart-Agrawala [11] and Lamport [10]. It follows that the wrapper W renders both to be stabilizing tolerant to *Lspec*.

5.1 Ricart-Agrawala's Program, RA_ME

The idea of RA_ME is as follows. Whenever process j wants to enter the *critical section*, CS, it sends a timestamped *request* message to all the processes. k, upon receiving a *request* message from j, sends back

a *reply* message if k is not requesting or j's request has a lower timestamp than k's request. Otherwise, k defers the *reply* message. j enters CS only after it has received *reply* messages from all other processes. When j exits CS, it sends all the deferred *reply* messages.

We now describe RA_ME formally. In RA_ME, j maintains a variable called *deferred_set.j* in addition to the variables in *Lspec* (i.e., REQ_j, $j.REQ_k$, $received(j.REQ_k)$, $h.j$, $e.j$, and $t.j$). We define *deferred_set.j* as $\{k \mid received(j.REQ_k) \ \wedge \ REQ_j \ \underline{lt} \ j.REQ_k\}$. In RA_ME, we define *deferred_set.j* inside an "always section" [3]; the value of *deferred_set.j* is *always* equal to the value of the right-hand side of the equality. Notice that whenever k is removed from *deferred_set.j*, in order to satisfy *Reply Spec*, a reply message should be sent to k.

In order to everywhere implement *Structural Spec*, we employ a variable called *state.j* over a domain of h, e, j. We assert (structurally) that $h.j \equiv$ (*state.j*$= h$), $e.j \equiv$ (*state.j*$= e$), and $t.j \equiv$ (*state.j*$= t$).

Initially, for all j, $REQ_j = 0$, $(\forall k :: j.REQ_k = 0)$, $t.j =$ *true*, $(\forall k :: received(j.REQ_k) =$*false*) and *deferred_set.j* is empty. RA_ME assumes FIFO channels, and that initially all the channels are empty. The resulting process actions for j are as follows.

RA_ME

always *deferred_set.j* = $\{ k \mid received(j.REQ_k) \ \wedge \ REQ_j \ \underline{lt} \ j.REQ_k\}$

$t.j \ \wedge \ \{$ *Request CS* $\} \longrightarrow$
$REQ_j := lc.j$; $h.j :=$ *true*;
$(\forall k : k \neq j :$ *send-request*$(j \ , REQ_j \ , k \ ))$

$\neg e.j \ \wedge \ \{$ *receive-request for* "REQ_k" $\} \longrightarrow$
$j.REQ_k := REQ_k$; $received(j.REQ_k) :=$ *true*;
if $(t.j)$ then $REQ_j := lc.j$;
if $(j.REQ_k \ \underline{lt} \ REQ_j)$
then *send-reply*$(j \ , REQ_j \ , k \ )$; $received(j.REQ_k) :=$ *false*

$\neg e.j \ \wedge \ \{$ *receive-reply for* "REQ_k" $\} \longrightarrow$
$j.REQ_k := REQ_k$

$h.j \ \wedge \ (\forall k : k \neq j : REQ_j \ \underline{lt} \ j.REQ_k) \longrightarrow$
$e.j :=$ *true*

$e.j \ \wedge \ \{$*Release CS* $\} \longrightarrow$ $(\forall k : k \in$ *deferred_set.j*: *send-reply*$(j \ , lc.j \ , k \ ))$;
$REQ_j := lc.j$; $t.j :=$ *true*; $(\forall k :: received(j.REQ_k) :=$ *false* $)$

Observe from RA_ME that *send-request* corresponds to the "send" in *Request Spec*, and *send-reply* corresponds to the "send" in *Reply Spec*. *Receive-request* corresponds to the "receive" in *Reply Spec*. *Receive-reply* also corresponds to the "receive" in *Reply Spec* (but this time no messages need to be sent since REQ_j is always less-than the reply from k).

Theorem 9. RA_ME everywhere implements *Lspec*.

$$[\text{ RA_ME } \Rightarrow \textit{ Lspec }]$$ □

5.2 Lamport's Program

In Lamport's program, every process j maintains a queue, $request_queue.j$, to store the existing CS requests ordered according to their timestamps. Whenever j wants to enter CS, it places its request timestamp into $request_queue.j$ and sends a timestamped *request* message to all the processes. k, upon receiving a *request* message from j, returns a timestamped *reply* message to j and places j's request into $request_queue.k$. j enters CS only after it has received *reply* messages from all other processes and j's request is at the head of $request_queue.j$. When j exits CS, it sends a timestamped *release* message to all processes. When k receives a *release* message from j, it removes j's request from $request_queue.k$.

We make two modifications to Lamport's program so that it implements *Lspec* from any state. The first modification is that we assume that the *Insert* primitive that is used for placing the requests into $request_queue$ also ensures that each process has at most one request in the queue. This enables the correction of an old and possibly incorrect request of j when a new request from j is received. The second modification is that after j has received *Reply* messages from all other processes, j can enter the CS if j's request is equal to *or less than* the request at the head of $request_queue.j$. This ensures that *CS Entry Spec* is satisfied in any state. In the appendix we give an implementation of Lamport's program, namely Lamport_ME.

Theorem 10. Lamport_ME everywhere implements *Lspec*.

$$[\text{ Lamport_ME } \Rightarrow \textit{ Lspec }]$$ □

Corollary 11. From Theorems 8, 9, and 10, it follows that W renders RA_ME and Lamport_ME stabilizing tolerant to *Lspec* (and, hence, to *TME_Spec*). □

6 Concluding Remarks

In this paper, we investigated the graybox design of system stabilization, which uses only the system specification, towards overcoming drawbacks of the traditional whitebox approach, which uses the system

implementation as well. The graybox 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, graybox stabilization is more reusable than stabilization that is particular to an implementation.

The graybox approach has received limited attention in the previous work on dependability. In particular, we can point to [1,4,14] which reason at a graybox level; [13] addresses specification-oriented integration of system modules for designing dependable systems; and [12] addresses the role of automated formal methods for specifications which involve dependability.

Our endorsement of everywhere specifications that are "local" should not be confused with the clever stabilization technique that endorses "local checking and correction" (see, e.g. [2]). (A distributed system is locally checkable if whenever it is in a bad state, some link subsystem [i.e. two neighboring nodes and the channel between them] is also in a bad state. A distributed system is locally correctable if it is corrected to a good state by each link subsystem correcting itself to a good state.) Local checkability and correctability are requirements of implementations, and are thus geared to whitebox reasoning. By way of contrast, our requirement is at the level of a specification. Moreover, a local everywhere specification can accommodate implementations that may or may not be locally checkable or correctable.

Although we have limited our discussion of the graybox approach to the property of stabilization, the approach is applicable for the design of other dependability properties, for example, masking fault-tolerance and fail-safe fault-tolerance. (A system is masking fault-tolerant iff its computations in the presence of the faults implement the specification. A component is fail-safe fault-tolerant iff its computations in the presence of faults implement the "safety" part [but not necessarily the "liveness" part] of its specification.) Our observation that graybox stabilization is not readily achieved for all specifications is likewise true for graybox masking and graybox fail-safe. Moreover, our observation that local everywhere specifications are amenable to graybox stabilization is also true for graybox masking and graybox fail-safe.

Of course, local everywhere specifications may be too severe a demand for the design of all dependability properties, and so an interesting direction for further research is to identify other relevant classes of specifications that are amenable to graybox design of other dependability properties. Another direction we are pursuing is automatic synthesis of graybox dependability.

References

[1] 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.

[2] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[3] K. M. Chandy and J. Misra. *Parallel program design*. Addison-Wesley Publishing Company, 1988.

[4] M. Demirbas. Resettable vector clocks: A case study in designing graybox fault-tolerance. Master's thesis, Technical report OSU-CISRC-4/00-TR11, Ohio State University, February 2000.

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

[6] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[7] 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.

[8] 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.

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

[10] L. Lamport. Time, clocks, and the ordering of events in a disributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[11] G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1991.

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

[13] N. Suri, S. Ghosh, and T. Marlowe. A framework for dependability driven sw integration. *IEEE DCS*, pages 405–416, 1998.

[14] K. P. Vo, Y. M. Wang, P. E. Chung, and Y. Huang. Xept: A software instrumentation method for exception handling. *Proc. Int. Symp. on Software Reliability Engineering (ISSRE)*, November 1997.

Appendix A1

In this appendix, we provide the proofs for the lemmas and theorems that appear in the paper. In order to prove Theorem 5, we present additional theorems A.1 through A.7 inside the proof for Theorem 5.

Lemma 0. $([C \Rightarrow A] \wedge [W' \Rightarrow W]) \Rightarrow [(C \,[]\, W') \Rightarrow (A \,[]\, W)]$

Proof.

$$
\begin{array}{ll}
 & [C \Rightarrow A] \wedge [W' \Rightarrow W] \\
\Rightarrow & \{ \text{ monotonicity of } [] \text{ (w.r.t. } [\Rightarrow]\text{), twice } \}
\end{array}
$$

$$
\begin{array}{ll}
 & [(C \ \Box \ W) \ \Rightarrow \ (A \ \Box \ W)] \ \wedge \ [(C \ \Box \ W') \ \Rightarrow \ (C \ \Box \ W)] \\
= & \{ \text{ transitivity of } [\ \Rightarrow \ ] \ \} \\
 & [(C \ \Box \ W') \ \Rightarrow \ (A \ \Box \ W)]
\end{array}
$$

□

Lemma 2. Given that $A = (\Box \, i :: \ A_i)$, and $C = (\Box \, i :: \ C_i)$,

$$(\forall i :: [C_i \ \Rightarrow \ A_i]) \ \Rightarrow \ [C \ \Rightarrow \ A]$$

Proof.

$$
\begin{array}{ll}
 & (\forall i :: [C_i \ \Rightarrow \ A_i]) \\
\Rightarrow & \{ \text{ Lemma 0 } \} \\
 & [(\Box \, i :: \ C_i) \ \Rightarrow \ (\Box \, i :: \ A_i)] \\
= & \{ \text{ premise } \} \\
 & [C \ \Rightarrow \ A]
\end{array}
$$

□

Lemma 3. Given that $W = (\Box \, i :: \ W_i)$, $W' = (\Box \, i :: \ W'_i)$, $A = (\Box \, i :: \ A_i)$, and $C = (\Box \, i :: \ C_i)$,

$$((\forall i :: [C_i \ \Rightarrow \ A_i]) \ \wedge \ (\forall i :: [W'_i \ \Rightarrow \ W_i])) \ \Rightarrow \ [(C \ \Box \ W') \ \Rightarrow \ (A \ \Box \ W)]$$

Proof.

$$
\begin{array}{ll}
 & ((\forall i :: [C_i \ \Rightarrow \ A_i]) \ \wedge \ (\forall i :: [W'_i \ \Rightarrow \ W_i])) \\
= & \{ \text{ Lemma 2, twice } \} \\
 & ([C \ \Rightarrow \ A] \ \wedge \ [W' \ \Rightarrow \ W]) \\
= & \{ \text{ Lemma 0 } \} \\
 & [(C \ \Box \ W') \ \Rightarrow \ (A \ \Box \ W)]
\end{array}
$$

□

Theorem 5 (TME_Spec). Every system that implements *Lspec* also implements *TME_Spec*.

$$(\forall M :: [M \ \Rightarrow \ \mathit{Lspec}]_{init} \ \Rightarrow \ [M \ \Rightarrow \ \mathit{TME_Spec}]_{init} \ )$$

Proof. In order to prove Theorem 5, we first give some definitions, and identify an invariant, I, for *Lspec*. Based on this invariant, we prove Theorems A.1 through A.7 whereby the proof of Theorem 5 follows.

We say that $earlier.(j,k)$ holds if $REQ_j \ \underline{lt} \ REQ_k$ (i.e., $(\forall j,k :: earlier.(j,k) = REQ_j \ \underline{lt} \ REQ_k)$), and $earliest.j$ holds if j is *earlier* than all the other processes (i.e., $(\forall j :: earliest.j = (\forall k : k \neq j : earlier.(j,k)))$). We use $rank.j$ to denote the number of processes k such that $(j.REQ_k \ \underline{lt} \ REQ_j)$ holds (i.e., $(\forall j :: rank.j = (\sum_k : (k \neq j) \ \wedge \ (j.REQ_k \ \underline{lt} \ REQ_j) : 1))$).

$$\boxed{(I) \equiv (\forall j,k : j \neq k : j.REQ_k = REQ_k \ \ \vee \ \ j.REQ_k \ \underline{lt} \ REQ_k)}$$

Theorem A.1 (Invariant of *Lspec*). Every program that implements *Lspec* satisfies I.

$$(\forall M :: [M \ \Rightarrow \ \mathit{Lspec}]_{init} \ \Rightarrow \ [M \ \Rightarrow \ I]_{init} \ )$$

Proof.

$$
\begin{array}{ll}
 & [M \ \Rightarrow \ I]_{init} \\
\Leftarrow & \{ \ I \text{ holds for } \mathit{Init}, \ I \text{ is stable in } \mathit{Lspec} \ \} \\
 & [M \ \Rightarrow \ \mathit{Lspec}]_{init}
\end{array}
$$

□

Lemma A.2. $(\forall M :: [M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\Rightarrow\; [M \;\Rightarrow\; (\forall j,k : j \neq k : earlier.(j,k) \;\Longleftrightarrow\; \neg earlier.(k,j))]_{init})$

Proof.

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; earlier.(j,k)$

$\equiv$ { definition of $earlier$, *Timestamp Spec*: $\underline{lt}$ imposes a total order on $(\forall j :: REQ_j)$ }

$\neg earlier.(k,j)$

□

Lemma A.3. $(\forall M :: [M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\Rightarrow\; [M \;\Rightarrow\; ((\exists j :: earliest.j) \;\wedge\; (\forall j,k :: (earliest.j \;\wedge\; earliest.k) \;\Rightarrow\; j = k))]_{init})$

Proof. Follows from the fact that $\underline{lt}$ imposes a total order on $(\forall j :: REQ_j)$, the definition of $earliest$, and Lemma A.2. □

Theorem A.4. Every program that implements *Lspec* also implements *ME1*.

$$(\forall M :: [M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\Rightarrow\; [M \;\Rightarrow\; (\forall j,k :: (e.j \;\wedge\; e.k) \;\Rightarrow\; j = k)]_{init})$$

Proof.

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; e.j \;\wedge\; e.k$

$\Rightarrow$ { I, *CS Entry Spec* }

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; earliest.j \;\wedge\; earliest.k$

$\Rightarrow$ { Lemma A.3 }

$j = k$

□

Lemma A.5. $(\forall M :: [M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\Rightarrow\; [M \models (\forall j :: (h.j \;\wedge\; rank.j \geq 0) \;\mapsto\; (h.j \;\wedge\; rank.j = 0))]_{init})$

Proof.

$[M \;\Rightarrow\; \mathit{Lspec}]_{init}$

$=$ { I, definition of rank, *Request Spec* and *Timestamp Spec* }

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; (\forall j,m : m \in N : (h.j \;\wedge\; rank.j = m)\; unless\; (h.j \;\wedge\; rank.j < m))$

$\Rightarrow$ { Lemma A.3, *Request Spec*, *Reply Spec*, *CS Entry Spec*, *CS Release Spec*, and again *Reply Spec* }

$(\forall j,m : m \in N : ((h.j \;\wedge\; rank.j = m)\; unless\; (h.j \;\wedge\; rank.j < m))$

$\qquad\qquad \wedge\; ((h.j \;\wedge\; rank.j = m) \;\mapsto\; (h.j \;\wedge\; rank.j < m)))$

$\Rightarrow$ { induction on m }

$(\forall j :: (h.j \;\wedge\; rank.j \geq 0) \;\mapsto\; (h.j \;\wedge\; rank.j = 0))$

□

Theorem A.6. Every program that implements *Lspec* also implements *ME2*.

$$(\forall M :: [M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\Rightarrow\; [M \;\Rightarrow\; (\forall j :: h.j \;\mapsto\; e.j)]_{init})$$

Proof.

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; h.j$

$\Rightarrow$ { defn. of rank }

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; (h.j \;\wedge\; rank.j \geq 0)$

$\mapsto$ { Lemma A.5 }

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; (h.j \;\wedge\; rank.j = 0)$

$\Rightarrow$ { defn. of rank }

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; (h.j \;\wedge\; (\forall k : k \neq j : REQ_j \;\underline{lt}\; j.REQ_k))$

$\mapsto$ { *CS Entry Spec* }

$e.j$

□

Theorem A.7. Every program that implements *Lspec* also *ME3*.

$(\forall M :: [M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\Rightarrow\; [M \;\Rightarrow\; (\forall j, k : j \neq k : (h.j \;\wedge\; REQ_j \;\underline{hb}\; REQ_k) \;\mapsto\; ts.(e.j) < t.s(e.k))]_{init})$

Proof.

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; h.j \;\wedge\; REQ_j \;\underline{hb}\; REQ_k$

$=$ { defn. of earlier }

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; h.j \;\wedge\; earlier.(j, k)$

$\mapsto$ { Theorem A.6, I, *Request Spec*, and *CS Entry Spec* }

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; e.j \;\wedge\; earlier.(j, k)$

$\Rightarrow$ { defn. earliest }

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; e.j \;\wedge\; \neg earliest.k$

$\Rightarrow$ { I, *CS Entry Spec* }

$[M \;\Rightarrow\; \mathit{Lspec}]_{init} \;\wedge\; e.j \;\wedge\; \neg e.k$

$\Rightarrow$ { *Reply Spec, Timestamp Spec* }

$ts.(e.j) < ts.(e.k)$

□

Proof of Theorem 5 (continued). Proof of Theorem 5 follows from Theorems A.4, A.6, and A.7. □

Lemma 6 (Interference freedom). *Lspec* $[]$ W everywhere implements *Lspec*.

$$[(\mathit{Lspec} \;[]\; W) \;\Rightarrow\; \mathit{Lspec}]$$

Proof.

$true$

$=$ { $true \equiv [X \;\Rightarrow\; X]$ }

$[\mathit{Lspec} \;\Rightarrow\; \mathit{Lspec}]$

$\Rightarrow$ { $(h.j \;\Rightarrow\; REQ_j = REQ'_j)$ in W, W, *Request Spec* }

$[\mathit{Lspec} \;\Rightarrow\; \mathit{Lspec}] \;\wedge\; [W \;\Rightarrow\; \mathit{Request\ Spec}]$

$\Rightarrow$ { Lemma 0 }

$[(\mathit{Lspec} \;[]\; W) \;\Rightarrow\; (\mathit{Lspec} \;[]\; \mathit{Request\ Spec})]$

$=$ { *Lspec* is fusion-closed }

$[(\mathit{Lspec} \;[]\; W) \;\Rightarrow\; \mathit{Lspec}]$

□

Lemma 7. *Lspec* $[]$ W is stabilizing to *Lspec*.

$$(\mathit{Lspec} \;[]\; W) \;\underline{\text{is stabilizing to}}\; \mathit{Lspec}$$

Proof.

$true$

$\Rightarrow$ { *Request Spec*, W, *Reply Spec*, *Release Spec*, *Timestamp Spec*, channels flushed,
Communication Spec, $stable(I)$ in (*Lspec* [] W) (follows from Lemma 6, Theorem A.1) }
$[(\ Lspec\ [] \ W)\ \Rightarrow\ true\ \hookrightarrow\ I]$

$=$ { Lemma 6 }
$[(\ Lspec\ [] \ W)\ \Rightarrow\ true\ \hookrightarrow\ I]\ \wedge\ [(\ Lspec\ [] \ W)\ \Rightarrow\ Lspec]$

$=$ { Theorem A.1, fusion-closure, definition of stabilization }
(*Lspec* [] W) is stabilizing to *Lspec*

□

Theorem 9. RA_ME everywhere implements *Lspec*.

$$[\ \text{RA_ME}\ \Rightarrow\ Lspec\ ]$$

Proof .

Structural Spec: At any time $state.j$ denotes exactly one of $h.j$, $e.j$, or $t.j$.

Flow Spec: Program text; $h.j,\ e.j,\ t.j$ are modified only by *Request CS*, *Grant CS*, or *Release CS*.

CS Spec: Client assumption.

Timestamp Spec: RA_ME uses logical clocks.

Communication Spec: RA_ME assumes FIFO channels.

Request Spec: Program text; *Request CS* action and REQ_j is not changed until $t.j$ holds.

Reply Spec: Program text; *receive-request for* "REQ_k", *Release CS* action.

CS Entry Spec: Program text; *Grant CS* action.

CS Release Spec: Program text; *Release CS* action. □

Lamport's ME Program, Lamport_ME

In Lamport_ME, j maintains two variables, namely *request_queue.j* and *grant.j.k*, in addition to the variables in *Lspec* (i.e., REQ_j, $j.REQ_k$, $received(j.REQ_k)$, $h.j$, $e.j$, and $t.j$). *request_queue.j* is a queue that stores the existing CS requests that j is aware of. That is, $REQ_k \in$ *request_queue.j* iff j has received k's request message and since then has not received a release message from k. *grant.j.k* is a boolean that denotes whether j has received a reply to its request message from k. In Lamport_ME, a process, upon receiving a request message, sends back a reply immediately (cf. *receive-request* action). Thus, $received(j.REQ_k)$ is set to *true* at the beginning of *receive-request* action and set back to *false* at the end of that action.

We do not explicitly specify how $j.REQ_k$ should be modified by Lamport_ME. Instead, however, we define $j.REQ_k$ in terms of *grant.j.k*, *request_queue.j*, and REQ_j as follows:

$$REQ_j\ \underline{lt}\ j.REQ_k \quad \equiv \quad \textit{grant.j.k}\ \wedge\ (REQ_k \text{ is not ahead of } REQ_j \text{ in } \textit{request_queue.j})$$

We use "*Insert* ($request_queue.j$, REQ_k)" to place REQ_k into $request_queue.j$, "*Head* ($request_queue.j$)" to access the item at the head of $request_queue.j$, and "*Dequeue* ($request_queue.j$)" to remove the item at the head of $request_queue.j$. Initially, for all j, $REQ_j = 0$, $t.j$, ($\forall k ::$ *grant.j.k = false*), and $request_queue.j$ is empty. Lamport_ME assumes FIFO channels, and that initially all the channels are empty. The resulting process actions for j are given as follows.

Lamport_ME

$t.j \ \wedge \ \{$ *Request CS* $\} \longrightarrow$ $REQ_j := lc.j$; $h.j$:=*true*;
Insert($request_queue.j$, REQ_j);
($\forall k : k \neq j :$ *send-request*(j , REQ_j , k))

$\neg e.j \ \wedge \ \{$ *receive-request for* "REQ_k" $\} \longrightarrow$
Insert($request_queue.j$, REQ_k);
send-reply(j , $lc.j$, k)

$\neg e.j \ \wedge \ \{$ *receive-reply for* "$lc.k$" $\} \longrightarrow$
if ($REQ_j \ \underline{lt} \ lc.k$) then $grant.j.k$:= *true*

$h.j \ \wedge \ (\forall k : k \neq j : grant.j.k)$
$\wedge \ (REQ_j =$ *Head*($request_queue.j$) $\vee \ REQ_j \ \underline{lt}$ *Head*($request_queue.j$)) $\longrightarrow$
$e.j$:= *true*

$e.j \ \wedge \ \{$*Release CS* $\} \longrightarrow$ $REQ_j := lc.j$; $t.j$:=*true*;
($\forall k : k \neq j : grant.j.k :=$ *false*);
Dequeue($request_queue.j$);
($\forall k : k \neq j$: *send-release*(j , REQ_j , k))

$\neg e.j \ \wedge \ \{$ *receive-release for* "REQ_k" $\} \longrightarrow$
Dequeue($request_queue.j$)

Observe from Lamport_ME that *send-request* corresponds to the "send" in *Request Spec*, and *send-reply*, *send-release* correspond to the "send" in *Reply Spec*. *Receive-request* corresponds to the "receive" in *Reply Spec*. *Receive-reply* and *receive-release* also correspond to the "receive" in *Reply Spec* (but this time no messages need to be sent since REQ_j is always less-than the reply/release from k).

Theorem 10. Lamport_ME everywhere implements *Lspec*.

$$[\text{ Lamport_ME } \Rightarrow \ \textit{Lspec} \ ]$$

Proof . The proof is the same as that in Theorem 9, except that for the proof of *CS Entry Spec* we use the definition of $(REQ_j \ \underline{lt} \ j.REQ_k)$ given above. □

Appendix A2 : Symbols and Operators

Symbol	Used as
A, Lspec, TME_Spec	Abstract system, specification
C	Concrete system, implementation
W	Wrapper

Operators	Explanation
$\Box$	Box operator (cf. Section 2.1)
$unless$	'Unless' operator (cf. Section 3.1)
$stable$	$stable(p) = p\ unless\ false$
$\mapsto$	'Leads to' operator (cf. Section 3.1)
$\hookrightarrow$	'Leads to always' operator (cf. Section 3.1)

Propositional connectives (in decreasing order of precedence)	
$\neg$	Negation
$\wedge, \vee$	Conjunction, Disjunction
$\Rightarrow, \Leftarrow$	Implication, Follows from
$\equiv, \not\equiv$	Equivalence, Inequivalence

First order quantifiers	
$\forall, \exists$	Universal (pronounced *for all*), Existential (pronounced *there exists*)