Graybox Stabilization

Anish Arora* Murat Demirbas* Sandeep S. Kulkarni†

*Computer and Info. Science
The Ohio State University
Columbus, Ohio 43210 USA

†Computer Science and Eng.
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. Towards redressing 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-eventually specifications" as being amenable to design of graybox stabilization. We illustrate the design of graybox stabilization using timestamp-based distributed mutual exclusion as our example.

1 Introduction

Research in stabilization [8–11] 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, 12, 18] also assume implementation-specific details as input: [3, 12] assume the availability of the implementation invariant, [1] relies on the knowledge of the implementation actions, and [18] takes as input a "locally checkable" consistency predicate derived from implementation.

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 focus more on "what" as opposed to "how" and since implementations usually introduce new control and data details, specifications are typically more succinct than implementations, and thus, 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. As one piece of evidence, we offer the following quote due to Varghese [18] (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) . . .

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 grants from Microsoft Research and Michigan State University.

Secondly, in this paper, we introduce the concept of *everywhere specifications* (and the more general *everywhere-eventually specifications*), which are amenable to the design of graybox stabilization. Intuitively speaking, these specifications demand that their implementations always (respectively, eventually) satisfy them from every state. 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. Further, for effective design of stabilization in distributed systems, we identify the subclass of *local everywhere specifications* (resp., *local everywhere-eventually specifications*): these specifications are decomposable into parts each of which must be always (resp., eventually) satisfied by some system process from *all* of its states without relying on its environment (including other processes).

Thirdly, we illustrate the design of graybox stabilization in the context of timestamp-based distributed mutual exclusion (TME). Since TME itself is not an everywhere specification, we present a local everywhere-eventually specification *Lspec* that satisfies TME, 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 observe that Ricart-Agrawala ME and Lamport ME programs satisfy *Lspec*, and, hence, W adds stabilization to both of them without knowing how they are implemented.

Organization of the paper. In Section 2, we show that local everywhere-eventually specifications are amenable to design of graybox stabilization. In Section 3, we present our "local everywhere-eventually specification", *Lspec*, for TME. Then, in Section 4, we design the wrapper W for *Lspec*. In Section 5, we show that Ricart-Agrawala's TME program [14], and Lamport's TME program [13] satisfy *Lspec*, and hence W adds stabilization to both of them. We make concluding remarks in Section 6.

2 Graybox Design

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

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*.

Remark: Of course, implementations often use some components of states that are not used by specifications (intuitively, such components are "hidden" from specifications). In our set up, this is captured by allowing specifications to be independent of the values of such state components. For example, an implementation may use a set or a queue (cf. Ricart-Agrawala's and Lamport's programs in Section 5) whereas a specification does not contain any such variables. (End of Remark.)

Henceforth, let C be an implementation and A a specification.

Definition. C *implements* A, denoted $[C \subseteq A]_{init}$, 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 $[C \subseteq A]$, iff every computation of C is a computation of A.

Definition. C *everywhere-eventually implements* A, denoted $[C \subseteq \Sigma^* A]$, iff every computation of C can be written as an arbitrary finite prefix from the state space Σ followed by 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-eventually 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, 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, $[C \subseteq A]_{init}$. But "s*, $s2,\ s3,\ldots$" is a computation that is in A but not in C. Letting F denote a transient state corruption fault that yields s* upon starting from $s0$, it follows that although A is stabilizing to A if F occurs initially, C is not.

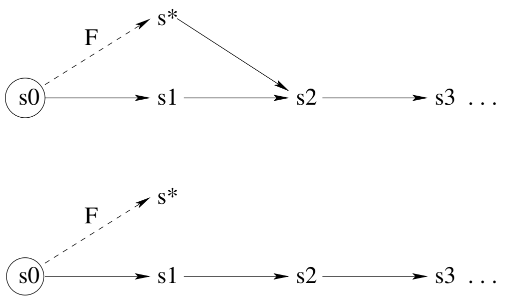

Figure 1. $[C \Rightarrow A]_{init}$

We are therefore led to considering the following class of specifications.

Definition. Everywhere specifications are specifications that demand their implementations to everywhere implement them. That is, an everywhere specification A demands that its implementations C also satisfy $[C \subseteq A]$.

Definition. Everywhere-eventually specifications are specifications that demand their implementations to everywhere-eventually implement them. That is, an everywhere-eventually specification A demands that its implementations C also satisfy $[C \subseteq \Sigma^* A]$.

We show that these specifications satisfy the goals of graybox design. First, observe that:

$[C \subseteq A]$ 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 formulation of "addition" of one system to another in terms of the operator $[]$ (pronounced "box") only assumes that $[]$ is monotonic in both arguments with respect to everywhere implements. That is, for systems X, Y and Z:

Proposition 0.

$$[X \subseteq Y] \Rightarrow ([X \,[]\, Z \subseteq Y \,[]\, Z] \wedge [Z \,[]\, X \subseteq Z \,[]\, Y]) \qquad \Box$$

Using Proposition 0, we prove the following lemma.

Lemma 1.

$$([C \subseteq A] \;\wedge\; [W' \subseteq W]) \Rightarrow [(C \,[]\, W') \subseteq (A \,[]\, W)] \qquad \Box$$

From the lemma, our goal follows trivially:

Theorem 2.
(Stabilization via everywhere specifications)
If $[C \subseteq A]$, $A \,[]\, W$ is stabilizing to A, $[W' \subseteq W]$ then $C \,[]\, W'$ is stabilizing to A. $\Box$

Corollary 3.
If $[C \subseteq \Sigma^* A]$, $[\Sigma^* A \subseteq \Sigma^*(A \,[]\, W)]$, $A \,[]\, W$ is stabilizing to A, and $[W' \subseteq W]$ then $C \,[]\, W'$ is stabilizing to A. $\Box$

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(-eventually) 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., $A = (\cap\, i :: \; A_i)$ [1]. Hence, given a distributed implementation $C = (\cap\, i :: \; C_i)$ 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 (including the other processes).

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

Lemma 4. $(\forall i :: [C_i \subseteq A_i]) \Rightarrow [C \subseteq A]$ $\Box$

Lemma 5. $((\forall i :: [C_i \subseteq A_i]) \;\wedge\; (\forall i :: [W'_i \subseteq W_i])) \Rightarrow [(C \,[]\, W') \subseteq (A \,[]\, W)]$ $\Box$

From Lemma 5 and Theorems 2, we get the following.

[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 [7].

Theorem 6.
(Stabilization via local everywhere specifications)
If $(\forall i :: [C_i \subseteq A_i])$, $(\forall i :: [W'_i \subseteq W_i])$,
and $A \;[]\; W$ is stabilizing to A,
then $C \;[]\; W'$ is stabilizing to A. $\square$

Corollary 7.
If $(\forall i :: [C_i \subseteq \Sigma^* A_i])$, $(\forall i :: [W'_i \subseteq W_i])$, $[\Sigma^* A \subseteq \Sigma^*(A \;[]\; W)]$, $A \;[]\; W$ is stabilizing to A,
then $C \;[]\; W'$ is stabilizing to A. $\square$

Corollary 7 is the formal statement of the amenability of local everywhere-eventually specifications for graybox 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.

Although Corollary 7 clarifies the role of local everywhere-eventually specifications, it leaves open the question of how to design wrappers $(\cap\ i :: W_i)$ that render $(\cap\ i :: A_i)$ stabilizing. For instance, designing W_i for each i such that $(A_i \;[]\; W_i)$ is stabilizing to A_i does not always imply that $(\cap\ i :: A_i \;[]\; W_i)$ is stabilizing to $(\cap\ i :: A_i)$; 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. That is, we need to also design wrappers to resolve inter-process consistency issues.

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) are subject to transient failures and their state may 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 [5]. 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 [5].

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:
  $(\forall j,k : j \neq k : (h.j \;\wedge\; REQ_j \;\underline{hb}\; REQ_k) \;\mapsto\; ts.(e.j) < ts.(e.k))$ [2]

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. While designing graybox stabilization for TME, one possibility is to demand that TME_Spec be an everywhere(-eventually) specification. However this is unreasonable: 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(-eventually)

[2] Lamport's [13] 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.

implements *ME1*. The problem therefore is to find an everywhere(-eventually) specification that implies *TME_Spec*. Moreover, the specification should be suitable for distributed systems; i.e., it should be a local everywhere(-eventually) specification.

The problem now reduces to (i) find a local everywhere(-eventually) 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(-eventually) implements *Lspec*, $M \ [] \ W$ stabilizes to *Lspec*.

3.2 *Lspec*

Our *Lspec* for each j 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 of j.
Structural Spec:
$(h.j \not\equiv (e.j \ \vee \ t.j)) \ \wedge \ \neg(e.j \ \wedge \ t.j)$
Flow Spec:
$(h.j \ unless \ e.j) \ \wedge \ (e.j \ unless \ t.j)$
$\wedge \ (t.j \ unless \ h.j)$
CS Spec:
$e.j \ \mapsto \ \neg e.j$

Program Spec of j.
Request Spec:
$(h.j \ \Rightarrow \ REQ_j = REQ'_j)$
$\wedge \ h.j \ \mapsto \ (\forall k : k \neq j : \text{Sent}(REQ_j, j, k))$
Reply Spec:
$(\forall k : j \neq k :$
$(\text{Received}(j.REQ_k) \ \wedge \ j.REQ_k \ \underline{lt} \ REQ_j)$
$\mapsto \ \text{Sent}(REQ_j, j, k))$
CS Entry Spec:
$(e.j \ \Rightarrow \ REQ_j = REQ'_j) \ \wedge$
$(h.j \ \wedge \ (\forall k : k \neq j : REQ_j \ \underline{lt} \ j.REQ_k)) \ \mapsto \ e.j$
CS Release Spec:
$t.j \ \Rightarrow \ REQ_j = ts.j$

Environment Spec of j.
Timestamp Spec:
ts is from a total domain and
$(\forall e, f :: e \ \underline{hb} \ f \ \Rightarrow \ ts.e < ts.f)$
Communication Spec:
Channels are FIFO.

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

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 sent to all processes (i.e., the predicate "Sent(REQ_j, j, k))" will be truthified for all k). *Reply Spec* guarantees that each time an earlier *request* is received from another process (i.e., the predicate "(Received$(j.REQ_k) \ \wedge \ j.REQ_k \ \underline{lt} \ REQ_j)$" holds), 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.

Theorem 8 (TME_Spec). Every system M that implements *Lspec* also implements *TME_Spec*.
$(\forall M :: [M \subseteq \textit{Lspec}]_{init} \Rightarrow [M \subseteq \textit{TME\_Spec}]_{init})$ □

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 [13]. 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: $(\forall e_j, f_k :: lc.e_j \ \underline{lt} \ lc.f_k = lc.e_j < lc.f_k \ \vee \ (lc.e_j = lc.f_k \ \wedge \ j < k))$, and $(\forall e, f :: e \ \underline{hb} \ f \ \Rightarrow \ lc.e \ \underline{lt} \ lc.f)$.

4 Graybox Stabilization Wrapper for TME

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

However 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 dependability wrapper W which consists of a wrapper at each process j (i.e., $W = \cap\ j :: W_j$). In *Lspec* mutual inconsistencies between two processes j and k may arise only due to $j.REQ_k$ and $k.REQ_j$ variables; the only information that j and k have about each other's state are stored in $j.REQ_k$ and $k.REQ_j$. These mutual 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 11 that any system M that everywhere-eventually implements *Lspec* can be made stabilizing to *Lspec* by using W. Towards this end, we first prove in Lemma 9 that W does not interfere with *Lspec*, and subsequently in Lemma 10 that *Lspec* composed with W is stabilizing to *Lspec*.

Lemma 9 (Interference freedom). *Lspec* $\Box$ W implements *Lspec*. $\square$

Lemma 10 (Stabilization). *Lspec* $\Box$ W is stabilizing to *Lspec*. $\square$

Theorem 11 (Graybox stabilization). For any system M that everywhere-eventually implements *Lspec*, $(M \ \Box\ W)$ is stabilizing to *Lspec* (and, hence, to *TME_Spec*).

$(\forall M :: [M \subseteq \Sigma^* \ Lspec] \ \Rightarrow$
$\quad (M \ \Box\ W)$ is stabilizing to *Lspec* $)$

Proof. W satisfies the condition in Corollary 3 since W does not depend on any history information. The proof then follows from Corollary 3 and Lemma 10. $\square$

Implementation of W. It follows from Theorem 7 that any W'_j such that $[W'_j \subseteq W_j]$ 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.

$$\begin{aligned} W'_j :: \ \ & (timer.j = 0 \ \wedge \ h.j) \ \longrightarrow \\ & \quad (\forall k : k \neq j \ \wedge \ j.REQ_k \ \underline{lt}\ REQ_j : \\ & \qquad send(REQ_j, j, k)); \\ & \quad timer.j = \Delta \end{aligned}$$

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-eventually implementations of *Lspec*, namely the mutual exclusion programs of Ricart-Agrawala [14] and Lamport [13]. 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 $t.j$). The computations of RA_ME that start from the initial states satisfy *deferred_set.j* $= \{k \mid Received(j.REQ_k) \wedge REQ_j \ \underline{lt}\ j.REQ_k\}$. Since this condition does not necessarily hold at every state, RA_ME is not an everywhere implementation of *Lspec*. However RA_ME is an everywhere-eventually implementation since the last action periodically corrects the value of *deferred_set.j* and once *deferred_set.j* is corrected RA_ME correctly tracks *Lspec*.

In order to everywhere implement *Structural Spec*, we employ a variable called *state.j* over a domain of h, e, t. 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 given in Figure 2.

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 12. RA_ME everywhere-eventually implements *Lspec*. □

RA_ME

$t.j \ \wedge$ { *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*;
  else *deferred_set.j*:= *deferred_set.j* $\cup \{k\}$

$\neg e.j \ \wedge$ { *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*; *deferred_set.j*$:=\emptyset$;
  $(\forall k :: Received(j.REQ_k) :=$ *false* $)$

true $\longrightarrow$ *deferred_set.j* := { $k \mid Received(j.REQ_k)$
  $\wedge \ REQ_j \ \underline{lt}\ j.REQ_k$}

Figure 2. RA_ME

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 now describe Lamport's ME program (Lamport_ME) formally. 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. The computations of Lamport_ME that start from the initial states satisfy the consistency conditions for *request_queue.j* and *grant.j.k*. Since these conditions do not necessarily hold at every state, Lamport_ME is not an everywhere implementation of *Lspec*. However Lamport_ME is an everywhere-eventually implementation since the last action periodically corrects the values of *request_queue.j* and *grant.j.k*, and once they are corrected Lamport_ME correctly tracks *Lspec*.

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 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 in Figure 3.

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 13. Lamport_ME everywhere-eventually implements *Lspec*. □

Corollary 14. From Theorems 11, 12, and 13, 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, it may be argued that graybox stabilization is more reusable than stabilization that is particular to an implementation.

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*"* } $\longrightarrow$
  $j.REQ_k := REQ_k$; $Received(j.REQ_k) :=$ *true*;
  Insert($request\_queue.j$, $j.REQ_k$);
  send-reply(j , $lc.j$, k);
  $Received(j.REQ_k) :=$ *false*

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

$h.j \;\wedge\; (\forall k : k \neq j : grant.j.k)$
 $\wedge\; REQ_j =$ *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*);
  ($\forall k : k \neq j : j.REQ_k := \infty$);
  Dequeue($request\_queue.j$);
  ($\forall k : k \neq j$: *send-release*(j , REQ_j , k))

$\neg e.j \;\wedge\;$ { *receive-release for "*REQ_k*"* } $\longrightarrow$
  $j.REQ_k := REQ_k$; $grant.j.k$:= *true*;
  Dequeue($request\_queue.j$)

true $\longrightarrow$
  ($\forall k : k \neq j : grant.j.k :=$
    $(REQ_j \; \underline{lt} \; j.REQ_k) \;\wedge\; j.REQ_k \neq \infty)$;
  $request\_queue.j :=$
    Sort($\{REQ_j, (\forall k : k \neq j : j.REQ_k)\}$)

Figure 3. Lamport_ME

The graybox approach has received limited attention in the previous work on dependability. In particular, we can point to [4, 6, 19] which reason at a graybox level; [17] addresses specification-oriented integration of system modules for designing dependable systems; and [15] addresses the role of automated formal methods for specifications which involve dependability.

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(-eventually) specifications are only a sufficient condition for graybox design of dependability properties. Experience [16, 20] in fact confirms that there are practical systems where local everywhere(-eventually) specifications are not necessary. 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.

Acknowledgments

We thank Ted Herman and an anonymous referee for insightful comments which helped to improve the paper.

References

[1] Y. Afek and S. Dolev. Local stabilizer. *PODC97 Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, page 287, 1997.

[2] A. Arora, M. Demirbas, and S. S. Kulkarni. Graybox stabilization. Technical Report OSU-CISRC-1/01-TR01, The Ohio State University, Department of Computer and Information Science, 2001. ftp://ftp.cis.ohio-state.edu/pub/tech-report/.

[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] 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.

[5] K. M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley Publishing Company, 1988.

[6] 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.

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

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

[9] 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.

[10] 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.

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

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

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

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

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

[16] A. Singhai, S.-B. Lim, and S.R. Radia. The SunSCALR framework for internet servers. *Proceedings of the 28th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-28)*, pages 108–117, 1998.

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

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

[19] 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.

[20] Y.-M. Wang, W. Russell, A. Arora, J. Xu, and R. Jagannathan. Towards dependable home networking: An experience report. *International Conference on Dependable Systems and Networks*, 2000.

Appendix A1

Lemma 1.
$([C \subseteq A] \wedge [W' \subseteq W]) \Rightarrow [(C \,\square\, W') \subseteq (A \,\square\, W)]$

Proof.

$$
\begin{array}{ll}
 & [C \subseteq A] \;\wedge\; [W' \subseteq W] \\
\Rightarrow & \{\text{ Proposition 0 applied twice }\} \\
 & [(C \,\square\, W) \subseteq (A \,\square\, W)] \;\wedge \\
 & [(C \,\square\, W') \subseteq (C \,\square\, W)] \\
= & \{\text{ transitivity of } [\ \subseteq\ ] \} \\
 & [(C \,\square\, W') \subseteq (A \,\square\, W)]
\end{array}
$$

$\square$

Lemma 4. Given that $A = (\cap\, i ::\ A_i)$, and $C = (\cap\, i ::\ C_i)$,

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

Proof.

$$
\begin{array}{ll}
 & (\forall i :: [C_i \ \subseteq \ A_i]) \\
\Rightarrow & \{ \text{ monotonicity of } \cap \text{ w.r.t. } [\ \subseteq\ ] \ \} \\
 & [(\cap\, i ::\ C_i) \ \subseteq \ (\cap\, i ::\ A_i)] \\
= & \{ \text{ premise } \} \\
 & [C \ \subseteq \ A]
\end{array}
$$

□

Lemma 5. Given that $W = (\cap\, i ::\ W_i)$, $W' = (\cap\, i ::\ W'_i)$, $A = (\cap\, i ::\ A_i)$, and $C = (\cap\, i ::\ C_i)$,

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

Proof.

$$
\begin{array}{ll}
 & ((\forall i :: [C_i \ \subseteq \ A_i]) \ \wedge \ (\forall i :: [W'_i \ \subseteq \ W_i])) \\
= & \{ \text{ Lemma 4, twice } \} \\
 & ([C \ \subseteq \ A] \ \wedge \ [W' \ \subseteq \ W]) \\
= & \{ \text{ Lemma 1 } \} \\
 & [(C \ \Box \ W') \ \subseteq \ (A \ \Box \ W)]
\end{array}
$$

□

Theorem 8 (TME_Spec). Every system that implements *Lspec* also implements *TME_Spec*.

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

Proof. In order to prove Theorem 8, we identify an invariant, I, for *Lspec*.

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

We prove Theorem 8 based on this invariant. For reasons of space we relegate the proof to [2]. □

Lemma 9 (Interference freedom). *Lspec* $\Box$ W implements *Lspec*.

$$[(\ \mathit{Lspec} \ \Box \ W) \ \subseteq \ \mathit{Lspec}]_{init}$$

Proof.

$$
\begin{array}{ll}
 & true \\
\Rightarrow & \{ \ (h.j \ \Rightarrow \ REQ_j = REQ'_j) \text{ in } W, W, \mathit{Request\ Spec} \ \} \\
 & [W \ \subseteq \ \mathit{Request\ Spec}]_{init} \\
\Rightarrow & \{ \text{ Proposition 0 } \} \\
 & [(\mathit{Lspec} \ \Box \ W) \ \subseteq \ (\mathit{Lspec} \ \Box \ \mathit{Request\ Spec})]_{init} \\
= & \{ \mathit{Request\ Spec} \in \mathit{Lspec} \} \\
 & [(\ \mathit{Lspec} \ \Box \ W) \ \subseteq \ \mathit{Lspec}]_{init}
\end{array}
$$

□

Lemma 10. *Lspec* $\Box$ W is stabilizing to *Lspec*.

Proof.

$$
\begin{array}{ll}
 & true \\
\Rightarrow & \{ \ \mathit{Request\ Spec}, W, \mathit{Reply\ Spec}, \mathit{Release\ Spec}, \\
 & \quad \mathit{Timestamp\ Spec}, \text{ channels flushed}, \\
 & \quad \mathit{Communication\ Spec}, stable(I) \text{ in } (\mathit{Lspec} \ \Box \ W) \\
 & \quad \text{(follows from Lemma 9, Theorem 8) } \} \\
 & [(\ \mathit{Lspec} \ \Box \ W) \ \subseteq \ true \hookrightarrow I] \\
= & \{ \text{ Lemma 9 } \} \\
 & [(\ \mathit{Lspec} \ \Box \ W) \ \subseteq \ true \hookrightarrow I] \\
 & \quad \wedge \ [(\ \mathit{Lspec} \ \Box \ W) \ \subseteq \ \mathit{Lspec}]_{init} \\
= & \{ \text{ From proof of Theorem 8, } W \text{ does not depend on} \\
 & \quad \text{history information, definition of stabilization } \} \\
 & (\ \mathit{Lspec} \ \Box \ W) \text{ is stabilizing to } \mathit{Lspec}
\end{array}
$$

□

Theorem 12. RA_ME everywhere-eventually implements *Lspec*.

$$[\ \text{RA\_ME} \ \subseteq \ \Sigma^* \ \mathit{Lspec}\ ]$$

Proof .
Structural Spec: At any time $state.j$ denotes exactly one of $h.j$, $e.j$, or $t.j$.
Flow Spec: $h.j$, $e.j$, $t.j$ 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: follows from *Request CS* action and that REQ_j is not changed until $t.j$ holds.
Reply Spec: *receive-request for* "REQ_k" and *Release CS* action and correction of *deferred_set.j* by the last action.
CS Entry Spec: *Grant CS* action.
CS Release Spec: *Release CS* action. □

Theorem 13. Lamport_ME everywhere-eventually implements *Lspec*.

$$[\ \text{Lamport\_ME} \ \subseteq \ \Sigma^* \ \mathit{Lspec}\ ]$$

Proof . The proof is the same as that in Theorem 12, except that for the proof of *CS Entry Spec* we use the correction of *request_queue.j* and *grant.j.k* by the last action of Lamport_ME.
□

Appendix A2 : Symbols and Operators

Symbol	Used as
A, *Lspec*	Abstract system, specification
C	Concrete system
W	Wrapper
$[C \ \subseteq \ A]_{init}$	C implements A
$[C \ \subseteq \ A]$	C everywhere implements A
$[C \ \subseteq \ \Sigma^* A]$	C everywhere-eventually implements A

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

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

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