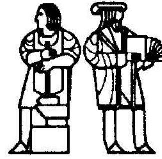
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-556

CONCURRENT TIMESTAMPING MADE SIMPLE

Rainer Gawlick

October 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Concurrent Timestamping Made Simple

by

Rainer Gawlick

B.A., Physics
University of California - Berkeley
(1989)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1992

© Massachusetts Institute of Technology 1992

Signature of Author______________________________
Department of Electrical Engineering and Computer Science
September 29, 1992

Certified by______________________________
Nancy A. Lynch
Professor of Computer Science
Thesis Supervisor

Certified by______________________________
Nir Shavit
Assistant Professor of Computer Science, Tel Aviv University
Thesis Supervisor

Accepted by______________________________
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

Concurrent Timestamping Made Simple

by

Rainer Gawlick

Submitted to the Department of Electrical Engineering and Computer Science

on September 29, 1992, in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

Abstract

Concurrent Timestamp Systems (CTSS) allow processes to temporally order concurrent events in an asynchronous shared memory system. Bounded memory constructions of a CTSS are extremely powerful tools for concurrency control, and are the basis for solutions to many coordination problems including mutual exclusion, randomized consensus, and multiwriter multireader atomic registers. Unfortunately, known bounded CTSS constructions seem to be complex from the algorithmic point of view. Because of the importance of bounded CTSS, the rather involved original construction by Dolev and Shavit was followed by a series of papers that tried to provide more easily verifiable CTSS constructions.

In this paper, we present what we believe is the simplest, most modular, and most easily proven bounded CTSS algorithm known to date. The algorithm is constructed and its correctness proven by carefully reasoned use of several tools. Our algorithm combines the labeling method of the Dolev-Shavit CTSS with the atomic snapshot algorithm proposed in Afek et. al, in a way that limits the number of interleavings that can occur. To facilitate our correctness proof, we introduce a specially tailored intermediate CTSS specification using unbounded label values taken from the positive reals. Our correctness proof first shows that the real-number based specification meets the CTSS axioms. Using the forward simulation techniques of the I/O Automata model, we then show that our bounded algorithm implements the real-number based specification. Finally, we prove that any CTSS that meets the CTSS axioms can be used to implement multireader multiwriter atomic registers and first-some-first-serve (*fcfs*) mutual exclusion.

Thesis Supervisor: Nancy A. Lynch

Title: Professor of Computer Science

Thesis Supervisor: Nir Shavit

Title: Assistant Professor of Computer Science, Tel Aviv University

Keywords: Timestamps, Concurrent Algorithms, Distributed Systems, I/O Automata

Contents

1 Introduction 5

2 I/O Automata Model 7

3 Concurrent Timestamp System 13

4 An Unbounded Concurrent Timestamp System 15

5 A Bounded Concurrent Timestamp System 23

6 Invariants 27

7 Simulation Proof 40

8 Applications 46

8.1 Multireader Multiwriter Atomic Registers . . . 47

8.2 Mutual Exclusion . . . 55

9 Formal Justification for Use of Snapshot 61

9.1 Theory . . . 62

9.2 Proof . . . 65

10 Discussion and Future Work 69

Acknowledgements

First, I would like to thank my advisor Nancy Lynch and my co–advisor Nir Shavit. Without Nancy's incredible ability to read, correct, and improve numerous drafts of long and intricate proofs, this thesis would never have been finished. Because of Nancy, I now know what it means to have a *formal proof.* Nir's Ph.D thesis provides an important basis for the research in this thesis. Furthermore, Nir's help and advice was instrumental in the early stages of the thesis work.

Thanks also go to my office mates, Roberto Segala and Jørgen Søgaard-Andersen, for many helpful discussions.

This work was supported in part by the Office of Naval Research under Contract N00014-91-J-1046, by the Defense Advanced Research Projects Agency under Contract N00014-89-J-1988, and by the National Science Foundation under Contract 89152206-CCR.

1 Introduction

The paradigm of concurrent timestamping is at the heart of solutions to some of the most fundamental problems in multiprocessor concurrency control. Examples of such problems include *fcfs* mutual exclusion [19], construction of a multireader multiwriter atomic register[34], and randomized consensus [8]. A simple bounded construction of a CTSS implies simple bounded solutions to most of these extensively researched problems.

A timestamp system is somewhat like a ticket machine at an ice cream parlor. People's requests to buy the ice cream are timestamped based on a numbered ticket (label) taken from the machine. Any person, in order to know in what order the requests will be served, can scan through all the labels and establish the total order among them. A *concurrent* timestamp system (CTSS) is a timestamp system in which any process can either take a new ticket or scan the existing tickets simultaneously with other processes. Furthermore, a CTSS is *waitfree*, which means that a process is guaranteed to finish any of the two above mentioned tasks in a finite number of steps, even if other processes experience stopping failures. Waitfree algorithms are highly suited for fault tolerant and realtime applications (see [16]).

Israeli and Li, in [17], were the first to isolate the notion of bounded timestamping (timestamping using bounded size memory) as an independent concept, developing an elegant theory of bounded *sequential* timestamp systems. Sequential timestamp systems prohibit concurrent operations. This work was continued in several interesting papers on sequential systems with weaker ordering requirements by Li and Vitanyi [26], Cori and Sopena [9] and Saks and Zaharoglou [35]. Dolev and Shavit [11] were the first to define and construct a bounded *concurrent* timestamp system. However, to quote [12]: "Their algorithm is ingenious but its proof is long and involved."

Because of the importance of the bounded concurrent timestamping problem, the original solution by Dolev and Shavit has been followed by a series of papers directed at providing a simpler bounded CTSS algorithm. Israeli and Pinchasov [18] have simplified the [11] algorithm and its proof by modifying the labeling scheme of [11], introducing a new label scanning method, and simplifying the ordering-of-events based formal proof [23] by reasoning about global states (However, it still takes over 40 pages...). Dwork and Waarts [12] have taken a totally different approach, by having their bounded construction simulate a new and simpler type of unbounded

CTSS construction in which processes choose from "local pools" of label values instead of a "global pool" as in [11, 18]. However, in order to bound the number of possible label values in the local pools, they are forced to introduce a form of amortized garbage collection. This greatly complicates their algorithm. (Their algorithm only has an informal operational proof.)

In this paper, we present a novel bounded algorithm that we believe is the simplest, most modular, and most easily proven CTSS algorithm known to date. Our basic approach is to decompose the problem into several distinct pieces.

- We base our algorithm on the atomic snapshot primitive introduced by Afek et. al [1] (we use it as a black box). This primitive is waitfree and allows a process to collect an "instantaneous" view of an array of shared registers. [1] gives an implementation of this primitive from atomic single writer multireader registers. By using a snapshot primitive, we limit the number of interleavings that can occur.

- The labeling operation, the operation of choosing a new label given a set of older ones, is very complex in all former algorithms. Based on the snapshot operation, we introduce a much simplified version of the labeling algorithm of [11].

- Proving that the bounded algorithm satisfies the CTSS specification has in the past led to long and involved inductive arguments. We overcome this problem by introducing a CTSS specification, that uses label values taken from the unbounded positive reals. Our correctness proof first shows that the real-number based specification meets the CTSS axioms of [11]. Using the forward simulation techniques of the I/O Automata model, we then show that our bounded algorithm implements the real-number based specification. (See [30] for references and a discussion of forward simulation techniques.)

The most efficient bounded CTSS implementations [12, 18] require $O(n)$ time per operation. Though one might think that a high price in complexity must be paid for our algorithm's modularity and ease of proof, this is not the case. The size of the labels is $O(n)$, and the time complexity of our algorithm is just that of the underlying atomic snapshot algorithm. The snapshot implementation of [3] requires $O(n\sqrt{n})$ single writer multireader register operations per snapshot operation. Hence the complexity of our algorithm is $O(n\sqrt{n})$ for each operation.

The final section of this paper considers some applications of the CTSS primitive. We present specific algorithms for *fcfs* mutual exclusion and multireader multiwriter atomic registers and prove that any CTSS can be used as a primitive in these algorithms.

2 I/O Automata Model

We present our algorithm in the context of the I/O Automata model. This model, introduced by Lynch and Tuttle [29], represents algorithms as *I/O Automata* which are characterized by *states*, *initial states*, a set of actions called an *action signature*, state transitions called *steps* and an equivalence relation on some of the actions of the action signature called a *partition*. For a I/O Automaton A its five components are denoted by $states(A)$, $start(A)$, $sig(A)$, $steps(A)$, and $part(A)$ respectively.

A step that results from an action is denoted by (s, π, s') where s is the original state, π is the action, and s' is the new state. If an action can be executed in a state s, it is said to be *enabled* in s. If an action is not enabled in state s, it is said to be *disabled* in s. Actions are classified into *external actions*, $ext(A)$, those visible to user of the algorithm, and *internal actions*, $int(A)$, which are not visible to the user. External actions are further classified into *input actions*, $in(A)$, which are under the control of the user of the algorithm, and *output actions*, $out(A)$, which are under the control of the algorithm. By definition input actions are enabled in all states. For an I/O Automaton A the tuple consisting of $in(A)$ and $out(A)$ is called A's *external action signature*, $exsig(A)$. We now give a more precise definition for some of the elements of an I/O Automaton. Specifically, for an I/O Automaton A, $sig(A) = (in(A), out(A), int(A))$. Furthermore, $part(A)$ defines an equivalence relation on the set of internal actions and output actions of A. Finally, we define $acts(A) = in(A) \cup out(A) \cup int(A)$.

An *execution* of an I/O Automaton is an alternating sequence of states and actions that could be produced if the algorithm is executed starting from an initial state. A state is called *reachable* is it is the final state of some execution. A *fair execution*, α, of infinite length is one in which for all $C \in part(A)$, if some action from C (not necessarily always the same action) is continuously enabled, α contains infinitely many actions from C. A fair execution of finite length is one in which for all $C \in part(A)$ no actions of C are enabled in the final state. A *schedule*, $sched(\alpha)$, is the projection of an execution α onto the actions of the I/O Automaton.

A *fair schedule*, *fairsched*(α), is the projection of a fair execution α on the actions of the I/O Automaton. A *behavior*, *beh*(α), is the projection of an execution α onto the external actions of the I/O Automaton. A *fair behavior*, *fairbeh*(α), is the projection of a fair execution α on the external actions of the I/O Automaton. The set of all possible behaviors of an I/O Automaton A is called *behs*(A). The set of all possible fair behaviors of an I/O Automaton A is called *fairbehs*(A).

In order to build complex I/O Automata from simple ones, the I/O Automata model defines the concept of *composition*. Composed I/O Automata interact using input and output actions that have the same name. Specifically, assume A and B are two composed I/O Automata. Let ACT be an output action of A and an input action of B. If A executes ACT this triggers the execution of ACT for B. In order to compose a set of I/O Automata, we must place certain restrictions on the action names the I/O Automata. Specifically, we require that none of the I/O Automata share any output actions, the internal actions of each I/O Automaton are not elements of the action sets of any other I/O Automaton, and no action can an element of the action sets of infinitely many I/O Automata (see [29] for a discussion of these restrictions). I/O Automata that satisfy these restrictions are said to be *strongly compatible*.

Definition 2.1 Let $I = \{1 \ldots n\}$. A *composition* $A = \prod_{i \in I} A_i$ of a countable collection of strongly compatible I/O Automata $\{A_1 \ldots A_n\}$ is the I/O Automaton defined as follows[1]:

- $sig(A) = \left(\bigcup_{i \in I} in(A_i) - \bigcup_{i \in I} out(A_i), \bigcup_{i \in I} out(A_i), \bigcup_{i \in I} int(A_i) \right)$,
- $states(A) = \prod_{i \in I} states(A_i)$,
- $start(A) = \prod_{i \in I} start(A_i)$,
- $steps(A)$ is the set of triples $(\bar{s_1}, \pi, \bar{s_2})$ such that for all i if $\pi \in acts(A)$, then $(\bar{s_1}[i], \pi, \bar{s_2}[i]) \in steps(A)$ and if $\pi \notin acts(A)$ then $\bar{s_1}[i] = \bar{s_2}[i]$.
- $part(A) = \cup_{i \in I} part(A_i)$,

■

[1]The $\prod$ symbol used to define $states(A)$ and $start(A)$ represents the normal Cartesian product. The notation $\bar{s}[i]$ denotes the ith component of the state vector $\bar{s}$.

We sometimes do not want the actions that constitute the interface between two composed I/O Automata to be visible to the environment. Therefore, the I/O Automata Model makes it possible to reclassify output actions to be internal actions. Such reclassified actions are said to be *hidden*.

The I/O Automata model represent a problem specification, P, as an external action signature, $exsig(P)$, along with set of allowable behaviors, $behs(P)$, on the actions in $exsig(P)$. An I/O Automaton A is said to *solve* a problem specification P if $exsig(A) = exsig(P)$ and $fairbehs(A) \subseteq behs(P)$. We say that an I/O Automaton A *implements* another I/O Automaton B if the $fairbehs(A) \subseteq fairbehs(B)$. Our correctness proof uses the following theorem on simulation proofs which is a restricted version of a theorem in [29].

Theorem 2.1 *Let A and B be I/O Automata with $sig(A) = sig(B)$, $part(A) = part(B)$, and R a relation over the states of A and B. Suppose:*

1. *If a is an initial state of A, then there exists an initial state b of B such that $(a, b) \in \mathrm{R}$.*
2. *Suppose a is a reachable state of A and b is a reachable state of B such that $(a, b) \in \mathrm{R}$. If (a, π, a') is a step of A then there exists a state b' of B such that (b, π, b') is a step of B and $(a', b') \in \mathrm{R}$.*
3. *If action π is enabled in state b of B and $(a, b) \in \mathrm{R}$ then action π is enabled in state a of A.*

Then $fairbehs(A) \subseteq fairbehs(B)$.

The I/O Automata model, while providing efficient techniques for reasoning about the correctness of algorithms, is much more general than the shared memory model [23] for which our timestamp algorithm is designed. Consequently, we introduce some added structure to the I/O Automata model. This section describes the basics needed to understand our correctness proof. Section 9 provides a more sophisticated development of shared memory concepts in the I/O Automata model. Some of the concepts in this section and most of the concepts in Section 9 are due to Goldman, Lynch and Yelick [15]. (See [28] for discussion of similar issues.)

We first introduce a type of interface which will be used to characterize the external action signature of I/O Automata and problem specifications for the shared memory model. The

interface captures the intuitive notion of a set of processes that perform operations on behalf of some user. Typically, any process might be able to perform several types of operations.

Definition 2.2 (operational interface) An *operational interface* is an external action signature S that partitions its actions into disjoint sets called *operation types*. The set of operation types of S is denoted by $ops(S)$. Each operation type consists of at least one input and one output action. ∎

As a short hand, we will sometime use the term *operation* instead of operation type. Notice that an operational interface only describes an external action signature. Hence an operational interface can be used to describe both I/O Automata and problem specifications. If we compose two I/O Automata which have an operational interface, the set of operation types of the composed I/O Automaton is the union of the sets of operation types of each of the constituent I/O Automata. Again, we must add some restrictions on a set of I/O Automata being composed. Assume that we wish to compose I/O Automaton A and I/O Automaton B. We require that each action in $acts(A) \cap acts(B)$ be an element of the same operation type in A and B. Furthermore, if one action of an operation type of A or B is in $acts(A) \cap acts(B)$ then all actions of that operation type are in $acts(A) \cap acts(B)$. An operation instance is defined as follows:

Definition 2.3 (operation instance) Let β be a behavior of an operational interface. Let a be an operation type of the operational interface. An *operation instance* is the occurrence of an input action of a and the first output action of a that follows the input action of a in the behavior β. ∎

We now introduce a set of notational conventions. Let S be an operational interface. For an operation type $a \in ops(S)$ we refer to the input actions of a by $\text{INVOKE}(a, v)$ and the output actions of a by $\text{RESPONSE}(a, r)$. The symbols v and r are syntactic placeholders for any arguments[2] that are used by this operation type. The I/O Automata and problem specifications that we consider typically allow several concurrent operations. We model concurrent operations with I/O Automata whose operational interfaces are structured as follows. Assume that A is an

[2]Formally, v and r are used to uniquely identify the actions of operation type a. Intuitively, v and r represent arguments. The arguments v and r are *syntactic* placeholders since the I/O Automata Model does not have the concept of an argument. Arguments are implemented by having a separate action for each possible argument value.

I/O Automaton with an operational interface that can handle up to n concurrent operations. Then for each $i \in \{1 \ldots n\}$ there exists a non empty set of operation types $S_i \subset ops(exsig(A))$. S_i and S_j are disjoint when $i \neq j$. For each operation type $a_i \in S_i$ we refer to the input actions of a_i by $\text{INVOKE}_i(a_i, v)$ and the output actions of a_i by $\text{RESPONSE}_i(a_i, r)$. Intuitively there is a process, p_i, associated with all actions whose names include the index i. For the remainder of the section, assume that all I/O Automata have an operational interface as described above.

We now define a set of concepts with which we can characterize the behaviors of I/O Automata and problem specifications that have operational interfaces. Let A be an I/O Automaton or a problem specification with an operational interface. If β is a behavior of A, then β_i is the projection of β onto the actions that have the index i as part of their name.

Definition 2.4 (well-formed) Let A be an I/O Automaton or a problem specification with an operational interface. A behavior β of A is *well-formed* if, for all β_i, β_i consists of an alternating sequence of input and output actions, starting with an input action, such that each output action is immediately preceded by an input action of the same operation type. Specifically, if $a_i \in ops(exsig(A))$, each $\text{RESPONSE}_i(a_i, r)$ action is immediately preceded by an $\text{INVOKE}_i(a_i, v)$ action. ■

Definition 2.5 (well-formed-input) Let A be an I/O Automaton or a problem specification with an operational interface. A behavior β of A has a *well-formed-input* if, for all β_i, there exist no two consecutive input actions. ■

Definition 2.6 (well-formed-preserving) Let A be an I/O Automaton or a problem specification with an operational interface. Let β be a behavior of A. β is *well-formed-preserving* if, for all prefixes β' of β that have a well-formed-input, β' is well-formed. ■

We say that an I/O Automaton is well-formed-preserving if all of its behaviors are well-formed-preserving. Similarly, a problem specification is well-formed-preserving of all of its behaviors are well-formed-preserving. In addition to the safety properties described by the well-formedness concepts, we require the following liveness property.

Definition 2.7 (response-live) Let A be an I/O Automaton or a problem specification with an operational interface. Let β be a well-formed behavior of A. Then β is *response-live* if each $\text{INVOKE}_i(a_i, v)$ action is eventually followed by a $\text{RESPONSE}_i(a_i, r)$ action. ■

We say that an I/O Automaton is response-live if all of its fair behaviors are response-live. Similarly, a problem specification is response-live of all of its behaviors are response-live. We can now define the following partial order on the operation instances of any well-formed and response-live behavior.

Definition 2.8 ($\longrightarrow$ order) Let β be a well-formed and response-live behavior of an I/O Automaton or problem specification with an operational interface. Let a_i and b_j be any two operation instances[3] in β. In general a_i and b_j can be instances of the same operation type. We say that $a_i \longrightarrow b_j$ if and only if in the behavior β the $\text{RESPONSE}_i(a_i, r)$ action associated with a_i precedes the $\text{INVOKE}_j(b_j, v)$ action associated with b_j. ∎

The order $\longrightarrow$ is the same as the *precedes* relation of [22, 23]. Since β is a well-formed behavior, all operations with same index are totally ordered by $\longrightarrow$.

An important type of I/O Automaton is called an *atomic* I/O Automaton. Before defining an *atomic* I/O Automaton we introduce the notion of a *serial specification* [38].

Definition 2.9 (serial specification) A *serial specification* is a set of finite and/or infinite sequences of operations. ∎

Intuitively, a serial specification characterizes a behavior consisting of a set of sequentially executed operations.

Definition 2.10 (atomic I/O Automata) An I/O Automaton A is *atomic* for a serial specification S if A has an operational interface, is well-formed-preserving, and is response-live. Furthermore, for any behavior $\beta \in \mathit{fairbehs}(A)$ there exists a total order $\Longrightarrow$ on the operation instances in β such that:

1. $\Longrightarrow$ is consistent with $\longrightarrow$.

2. The sequence consisting of the operation instances in β ordered by $\Longrightarrow$ is in S.

∎

[3] We sometimes use the same name for operation instances and operation types. The meaning of a name will always be clear from context.

3 Concurrent Timestamp System

The following is a formal definition of a CTSS due to Dolev and Shavit [11]. It uses the axiomatic specification formalism of Lamport [22, 23].

A CTSS is a problem specification with an operational interface. A CTSS that permits n concurrent operations has $2n$ operation types, specifically LABEL$_i$ and SCAN$_i$ for $i \in \{1 \ldots n\}$. Each of these operation types consists of the following actions: LABEL$_i$ consists of the input action BEGINLABEL$_i(val_i)$ and the output action ENDLABEL$_i$. SCAN$_i$ consists of the input action BEGINSCAN$_i$ and the output action ENDSCAN$_i(\bar{o}, \bar{v})$. A LABEL$_i$ operation associates a value, val_i, taken from any domain, V, with a label. In order correctly handle initial conditions the value domain V must specify some initial value v_o. A SCAN$_i$ operation returns a pair $(\bar{o}, \bar{v})$, where $\bar{v} = (v_1 \ldots v_n)$ is an indexed set of values (one per process), and $\bar{o}$ is an *total order* on these indexes.

We now introduce some notation. In a particular behavior β, $L_i^{[k]}$ denotes the kth instance of a LABEL$_i$ operation, and $S_i^{[k]}$ denotes the kth instance of a SCAN$_i$ operation. Furthermore, $val_i^{[k]}$ denotes the value passed to operation $L_i^{[k]}$. (The superscript $[k]$ is used only for notation, and is not visible to the I/O Automaton). We call the superscript $[k]$ an *execution number.* The domain of execution numbers is $E = \{1, 2, \ldots\}$. Finally, we define a *choice function*, c, as a function mapping $\{1 \ldots n\} \times E \times \{1 \ldots n\}$ to $E \cup \{0\}$. Intuitively, the choice function provides a way to determine which operation wrote a value returned by a SCAN operation. Specifically, if $c(i, a, k) \neq 0$, the value v_k returned by operation $S_i^{[a]}$ was written by the operation $L_k^{[c(i,a,k)]}$. If $c(i, a, k) = 0$, then the value v_k returned by operation $S_i^{[a]}$ is the initial value v_o.

The set of behaviors of a CTSS, *behs*(CTSS), is defined as follows:

Definition 3.1 $\beta \in$ *behs*(CTSS) if and only if:

1. If β has a well-formed-input, then β is well-formed.
2. If β has a well-formed-input, then β is response-live.
3. If β is well-formed, then there exists a total order $\Longrightarrow$ on the set of all LABEL operations and a choice function c such that β, $\Longrightarrow$ and c satisfy axioms **P0–P4** given below.

■

Note: if β does not have a well-formed-input, then β can be arbitrary.

In order to handle initial conditions, we let $val_i^{[0]} = v_o$ for all i, where v_o is the initial value of the value domain V. Recall that execution numbers start with 1.

P0 *choice function:* For any value v_k in $\bar{v}$ of $S_i^{[a]}$, $v_k = val_k^{[c(i,a,k)]}$ where $val_k^{[0]} = v_o$.

P1 *ordering:* $\Longrightarrow$ is a total order on the set of all LABEL operation instances in β, such that:

- **a.** *precedence:* For any pair of LABEL operation instances $L_i^{[a]}$ and $L_j^{[b]}$ (where possibly i and j are the same index), if $L_i^{[a]} \longrightarrow L_j^{[b]}$, then $L_i^{[a]} \Longrightarrow L_j^{[b]}$.
- **b.** *consistency:* For any SCAN operation instance $S_i^{[a]}$ that returns $\bar{v}$ and $\bar{o}$, if $v_j, v_k \in \bar{v}$:

  $c(i,a,j) > 0$ and $c(i,a,k) > 0$: $j < k$ in $\bar{o}$ if and only if $L_j^{[c(i,a,j)]} \Longrightarrow L_k^{[c(i,a,k)]}$.

  $c(i,a,j) = 0$ and $c(i,a,k) = 0$: $j < k$ in $\bar{o}$ if and only if $j < k$.

  $c(i,a,j) = 0$ and $c(i,a,k) > 0$: $j < k$ in $\bar{o}$.

  $c(i,a,j) > 0$ and $c(i,a,k) = 0$: $k < j$ in $\bar{o}$.

The above property implies that there is a unique total ordering on LABEL operation instances of all processes, which is a serialization order (part a), and with which all SCAN operations are consistent (part b).

P2 *regularity*: Let $S_j^{[a]}$ be a SCAN operation instance. If $c(j,a,i) > 0$, then $S_j^{[a]} \not\longrightarrow L_i^{[c(j,a,i)]}$ and there is no $L_i^{[b]}$ such that $L_i^{[c(j,a,i)]} \longrightarrow L_i^{[b]} \longrightarrow S_j^{[a]}$. If $c(j,a,i) = 0$, then there exists no $L_i^{[b]}$ such that $L_i^{[b]} \longrightarrow S_j^{[a]}$.

Though a *regular* CTSS (having properties *P0-P2*) would suffice for some applications (for example Lamport's "Bakery Algorithm" [19]), a more powerful concurrent timestamp system is needed in applications such as the multireader multiwriter atomic register construction (see [24, 34]). To this end the following third and fourth axioms are added:

P3 *monotonicity*: Let $S_i^{[a]}$ return $v_k = val_k^{[c(i,a,k)]}$ and $S_j^{[b]}$ return $v_k = val_k^{[c(j,b,k)]}$ (where possibly $i = j$). Then, $S_i^{[a]} \longrightarrow S_j^{[b]}$ and $c(i,a,k) \neq c(j,b,k)$ imply $c(i,a,k) < c(j,b,k)$.

Note that $c(i,a,k) < c(j,b,k)$ implies that $L_k^{[c(i,a,k)]} \Longrightarrow L_k^{[c(j,b,k)]}$ when $c(i,a,k) > 0$ and $c(j,b,k) > 0$. Monotonicity is the property that in a unbounded real number CTSS can be

described by saying that the labels of any one process, as read by increasingly later SCAN operations, are "monotonically non-decreasing." It is important to note that *P3* does not imply that one can serialize all LABEL and SCAN operation instances. It does however imply the serializability of the SCAN operation instances of all processes relative to the LABEL operation instances of any *one* process [37]. *P4* [4] is an extension of part of the regularity property to the $\Longrightarrow$ order. The properties *P3* and *P4* together imply that all SCAN operations that consider only the "largest" value, where "largest" is based on the $\bar{o}$ ordering, can be serialized with respect to all LABEL operations.

P4 $\Longrightarrow$ *regularity*: Let $S_i^{[a]}$ be a SCAN operation instance. If $c(i,a,k) > 0$, then $S_i^{[a]} \longrightarrow L_j^{[b]}$ implies that $L_k^{[c(i,a,k)]} \Longrightarrow L_j^{[b]}$.

4 An Unbounded Concurrent Timestamp System

This section introduces a particular implementation of a concurrent time stamp system, UCTSS, that uses timestamps from $\Re^+$. UCTSS is introduced as an intermediary I/O Automaton whose purpose is to simplify the correctness proof of our bounded CTSS.

The code for the operations of UCTSS is presented in two forms. Figure 1 presents the code in the precondition-effect notation commonly used to describe I/O Automata[5]. Figure 2 uses psuedocode. We use the precondition-effect notation as the basis for the correctness proof and include the compact and intuitive psuedocode only for clarity.

The system models n processes indexed by $\{1 \ldots n\}$. Each process p_i in UCTSS can perform a SCAN$_i$ and LABEL$_i$ operation. A LABEL$_i$ operation allows process p_i to associate a label (timestamp) with a given value. A SCAN$_i$ operation allows process p_i to determine the order among values based on their associated labels. The function NEWLABEL$_i$, which is used by LABEL$_i$ is defined in Figure 3. A SNAP$_i$ operation, which is defined by Afek et al. in [1], atomically reads an array of single writer multireader registers. A UPDATE$_i$ operation, also defined by [1], writes a value to a single register in the array of single writer multireader registers

[4] A more powerful CTSS satisfying *P4* is needed in applications such as the multireader multiwriter atomic register construction of [24, 34]. *P4* is included in the journal version of [11], but is not included in the conference version of [11] or in [37].

[5] BCTSS is the name for our bounded CTSS implementation. The name is included in the caption since the code in the figure is shared by BCTSS and UCTSS. BCTSS is introduced in Section 5.

Shared State:

t_i: The current label associated with process p_i; initially 0.
v_i: The current value associated with process p_i; initially v_o.

Local State:

nt_i: The new label for p_i determined by function MAKELABEL_i; initially 0.
val_i: The new value for p_i passed to LABEL_i; initially v_o.
$\bar{t}_i$: An array of labels returned by SNAP_i; initially $(0 \ldots 0)$.
$\bar{v}_i$: An array of values returned by SNAP_i; initially $(v_o \ldots v_o)$.
$\bar{o}_i$: An array of process indexes ordered based on the $\ll$ order; initially $(1 \ldots n)$.
pc_i: The non-input action currently enabled; initially NIL.
op_i: The current operation; initially NIL.

```
SCAN_i:
  BEGINSCAN_i                   Eff:  op_i ← SCAN_i
                                      pc_i ← SNAP_i(t̄_i, v̄_i)

  SNAP_i(t̄_i, v̄_i)             Pre:  pc_i = SNAP_i(t̄_i, v̄_i)
                                Eff:  If op_i = SCAN_i then
                                        ō_i ← the sequence of indexes where
                                        j appears before k in o_i iff (t_j, j) ≪ (t_k, k)
                                        pc_i ← ENDSCAN_i(ō_i, v̄_i)
                                      If op_i = LABEL_i then
                                        nt_i ← NEWLABEL_i(t̄_i)
                                        pc_i ← UPDATE_i((t_i, v_i), (nt_i, val_i))

  ENDSCAN_i(ō_i, v̄_i)          Pre:  pc_i = ENDSCAN_i(ō_i, v̄_i)
                                Eff:  pc_i ← NIL

LABEL_i:
  BEGINLABEL_i                  Eff:  op_i ← LABEL_i
                                      pc_i ← SNAP_i(t̄_i, v̄_i)

  UPDATE_i((t_i, v_i), (nt_i, val_i))  Pre:  pc_i = UPDATE_i((t_i, v_i), (nt_i, val_i))
                                       Eff:  pc_i ← ENDLABEL_i

  ENDLABEL_i                    Pre:  pc_i = ENDLABEL_i
                                Eff:  pc_i ← NIL
```

Figure 1: Precondition-Effect code for UCTSS and BCTSS

```
SCAN_i
   SNAP_i(t̄_i, v̄_i)
   ō_i ← the sequence of indexes where j appears before k in o_i iff (t_j, j) ≪ (t_k, k)
   return (ō_i, v̄_i)

LABEL_i(val_i)
   SNAP_i(t̄_i, v̄_i)
   nt_i ← NEWLABEL_i(t̄_i)
   UPDATE_i((t_i, v_i), (nt_i, val_i))
```

Figure 2: Psuedocode for UCTSS and BCTSS

read by SNAP_i. SNAP_i and UPDATE_i are waitfree, therefore their use does not compromise the waitfree properties of our timestamp algorithm.

```
NEWLABEL_i(t̄_i)
   if i ≠ i_max
     then return (t_max + X) where X is nondeterministically selected from ℜ^{>0}
```

Figure 3: Code for NEWLABEL_i of UCTSS

The state of UCTSS is defined by the shared state and the local state of each of the n process. The shared and local state of each process, along with the initial values are defined in Figure 1. The state of UCTSS also has derived variables t_{max} and i_{max}. $t_{max} = \text{MAX}(t_1 \ldots t_n)$ and i_{max} is the largest process index i such that $t_i = t_{max}$.

In terms of the I/O Automata model, UCTSS is an I/O Automaton with an operational interface. UCTSS is a composition of n I/O Automata called $p_1, \ldots, p_n$. Each p_i is an I/O Automaton with an operational interface that consists of the operation types LABEL_i and SCAN_i. The LABEL_i operation type consists of the input action $\text{BEGINLABEL}_i(val_i)$ and the output action ENDLABEL_i. The operation type SCAN_i consists of the input action BEGINSCAN_i and the output action $\text{ENDSCAN}_i(\bar{o}_i, \bar{v}_i)$. The internal actions of p_i are $\text{SNAP}_i(\bar{t}_i, \bar{v}_i)$ and $\text{UPDATE}_i((t_i, v_i), (nt_i, val_i))$. The set *steps*($p_i$) is characterized by the *precondition* clause in each action. The set *part*(p_i) consists of a single equivalence classes C_i where the elements of C_i are the actions $\text{SNAP}_i(\bar{t}_i, \bar{v}_i)$, $\text{ENDSCAN}_i(\bar{o}_i, \bar{v}_i)$, $\text{UPDATE}_i((t_i, v_i), (nt_i, val_i))$, and ENDLABEL_i. The set *states*(p_i) is the set of all possible states of p_i where each state is defined by the values of the variables of the shared

and local state. The set *start*(p_i) is the set consisting of the state defined by the initial values of the variables of the shared and local state.

The shared state is accessed only using the atomic SNAP$_i$ and the UPDATE$_i$ actions. Since SNAP$_i$ and UPDATE$_i$ are atomic, each action of UCTSS is atomic. Notice that the SNAP$_i$ action makes references to the elements of the vector $\bar{t}_i$ indirectly through the use of i_{max} and t_{max} and in order to calculate $\bar{o}_i$. Since SNAP$_i$ is atomic, the labels in $\bar{t}_i$ are the same as the corresponding labels in the shared state. In other words, $t_{i_j} = t_j$ during the action. Consequently, we refer directly to the shared variables i_{max}, t_{max}, and t_i rather than their copies $i_{i_{max}}$, $t_{i_{max}}$, and t_{i_i} when analyzing the SNAP$_i$ action.

UCTSS uses labels that are non-negative real numbers. The ordering between labels is the usual $<$ order of $\Re^+$. The ordering $\ll$ used in the ORDER$_i$ action is a lexicographical order between label and process index pairs.

Definition 4.1 ($\ll$ order) $(\ell_i, i) \ll (\ell_j, j)$ iff $\ell_i < \ell_j$ or $\ell_i = \ell_j$ and $i < j$. ■

We now prove some characteristics of $\ll$ that will be used to prove that UCTSS solves CTSS. First consider the following notation: $t_i^{[a]}$ is the label written as a consequence of the $L_i^{[a]}$ operation. When $a = 0$, then $t_i^{[a]}$ is equal to the initial value for labels, which for UCTSS is 0. $L_i^{[a]}$(UPDATE) refers to the UPDATE$_i$ action executed as a consequence of the $L_i^{[a]}$ operation and $L_i^{[a]}$(SNAP) refers to the SNAP$_i$ action executed as a consequence of the $L_i^{[a]}$ operation. Similarly, $S_i^{[a]}$(SNAP) refers to the SNAP$_i$ action executed as a consequence of the $S_i^{[a]}$ operation. The SNAP and UPDATE actions model two atomic operations. In the usual model for atomic operations [23], each operation is separated into a *request* (input) action and a *response* (output) action, concurrent operations executions are allowed, and it is assumed that every request eventually terminates in a matching response, in such a way as to produce the *illusion* of instantaneous operations. Consequently, we model SNAP and UPDATE as single actions rather than separate input and output actions. We present a formal justification for treating SNAP and UPDATE operations as single actions rather than separate input and output actions in Section 9. Since SNAP and UPDATE are single actions, there exists a total order on all SNAP and UPDATE actions. We represent this order by $\Longrightarrow'$. If a SNAP action returns the set of values, $\bar{v}$, and labels, $\bar{t}$, then v_k and t_k are the value and label written by the UPDATE$_k$ action that immediately proceeds the SNAP action in the $\Longrightarrow'$ ordering. If a SNAP action is not proceeded by an UPDATE$_k$ action,

then v_k and t_k are equal to their initial values.

Lemma 4.1 *Consider any well-formed, response-live behavior β where $\beta \in$ fairbehs*(UCTSS). *For any i, a and* SNAP *operation $L_j^{[b]}$*(SNAP), *if either $a > 0$ and $L_i^{[a]}$*(UPDATE) $\Longrightarrow'$ $L_j^{[b]}$(SNAP) *in β, or $a = 0$ then:*

1. $(t_i^{[a]}, i) \ll (t_j^{[b]}, j)$ *when $i \neq j$.*

2. $(t_i^{[a]}, i) = (t_j^{[b]}, j)$ *or* $(t_i^{[a]}, i) \ll (t_j^{[b]}, j)$ *when $i = j$.*

Proof: Let $\overline{t_{max}}$ and $\overline{i_{max}}$ be the t_{max} and i_{max} used in NEWLABEL$_j$ for $L_j^{[b]}$. Since β is well-formed, each process must read its current label when determining its new label. This fact, along with the fact that X in NEWLABEL$_i$ is in $\Re^{>0}$, shows that the labels for all process are nondecreasing. In other words, a label for some process in a particular state of β is never larger than the label for the same process in a subsequent state of β. Thus $t_i^{[a]} \leq \overline{t_{max}}$ when $a = 0$. When $a > 0$, $L_i^{[a]}$(UPDATE) $\Longrightarrow'$ $L_j^{[b]}$(SNAP) shows that $t_i^{[a]} \leq \overline{t_{max}}$. Consider the following cases:

$j = \overline{i_{max}}$ and $i \neq j$: When $j = \overline{i_{max}}$, then $\overline{t_{max}} = t_j^{[b-1]}$. Recall that $t_i^{[a]} \leq \overline{t_{max}}$. Consider the cases $t_i^{[a]} = \overline{t_{max}}$ and $t_i^{[a]} < \overline{t_{max}}$ separately. When $t_i^{[a]} = \overline{t_{max}}$, then, since $\overline{t_{max}} = t_j^{[b-1]}$, $t_i^{[a]} = t_j^{[b-1]}$. Furthermore, since $i \neq j$ and $j = \overline{i_{max}}$, $i \neq \overline{i_{max}}$. Since $j = \overline{i_{max}}$, $i \neq \overline{i_{max}}$ and $t_i^{[a]} = t_j^{[b-1]}$, the definition of i_{max} shows that $i < j$. As a result of the action $L_j^{[b]}$, $t_j^{[b]} = \overline{t_{max}}$. Hence, $t_i^{[a]} = t_j^{[b]}$ and $i < j$ which implies that $(t_i^{[a]}, i) \ll (t_j^{[b]}, j)$. Now consider the case $t_i^{[a]} < \overline{t_{max}}$. As a result of the action $L_j^{[b]}$, $t_j^{[b]} = \overline{t_{max}}$. Hence $t_i^{[a]} < t_j^{[b]}$ which implies that $(t_i^{[a]}, i) \ll (t_j^{[b]}, j)$.

$j = \overline{i_{max}}$ and $i = j$: As a result of the action $L_j^{[b]}$ and the fact that $j = \overline{i_{max}}$, $t_j^{[b]} = \overline{t_{max}}$. Since $t_i^{[a]} \leq \overline{t_{max}}$, it must now be the case that $t_i^{[a]} \leq t_j^{[b]}$. This implies that $(t_i^{[a]}, i) = (t_j^{[b]}, j)$ or $(t_i^{[a]}, i) \ll (t_j^{[b]}, j)$.

$j \neq \overline{i_{max}}$: As a result of the action $L_j^{[b]}$ and the fact that $j \neq \overline{i_{max}}$, $\overline{t_{max}} < t_j^{[b]}$. Since $t_i^{[a]} \leq \overline{t_{max}}$, it must now be the case that $t_i^{[a]} < t_j^{[b]}$. This implies that $(t_i^{[a]}, i) \ll (t_j^{[b]}, j)$.

∎

Corollary 4.2 *Consider any well-formed, response-live behavior β where $\beta \in fairbehs(\text{UCTSS})$. For any two* LABEL *operations $L_i^{[a]}$ and $L_j^{[b]}$, if $L_i^{[a]} \longrightarrow L_j^{[b]}$ in β, then:*

1. $(t_i^{[a]}, i) \ll (t_j^{[b]}, j)$ *when* $i \neq j$.
2. $(t_i^{[a]}, i) = (t_j^{[b]}, j)$ *or* $(t_i^{[a]}, i) \ll (t_j^{[b]}, j)$ *when* $i = j$.

Proof: If $L_i^{[a]} \longrightarrow L_j^{[b]}$, then $L_i^{[a]}(\text{UPDATE}) \Longrightarrow' L_j^{[b]}(\text{SNAP})$. Now Lemma 4.1 proves the corollary. ∎

Consider any well-formed, response-live behavior β where $\beta \in fairbehs(\text{UCTSS})$. Define $\Longrightarrow'$, a total order on all the SNAP and UPDATE operations of β, as before. We now define a total order[6] $\Longrightarrow$ on the LABEL operations in β and a choice function c. Recall from Definition 2.8 that $\longrightarrow$ defines a partial order on the operation instances of a well-formed, response-live behavior.

Definition 4.2 ($\Longrightarrow$ order) $L_i^{[a]} \Longrightarrow L_j^{[b]}$ iff either $L_i^{[a]} \longrightarrow L_j^{[b]}$ or $(t_i^{[a]}, i) \ll (t_j^{[b]}, j)$. ∎

Definition 4.3 (choice function c) If $S_i^{[a]}$ returns $\bar{v}$ and $L_j^{[b]}(\text{UPDATE})$ is the UPDATE_j action that immediately proceeds $S_i^{[a]}(\text{SNAP})$ in $\Longrightarrow'$, then $c(i, a, j) = b$. If no such UPDATE_j action exists, then $c(i, a, j) = 0$. ∎

For the following lemmas assume that β is well-formed, response-live, $\beta \in fairbehs(\text{UCTSS})$, and $\longrightarrow$ is defined as in Definition 2.8. Furthermore, $\Longrightarrow$ and c are defined as in Definition 4.2 and Definition 4.2 respectively.

Lemma 4.3 *The order $\Longrightarrow$ is a total order on all* LABEL *operation instances in β.*

Proof: In order to simplify the notation in this proof, we write $L_i^{[a]} \ll L_j^{[b]}$ instead of $(t_i^{[a]}, i) \ll (t_j^{[b]}, j)$. Since $\longrightarrow$ is a partial order, it is irreflexive, antisymmetric, and transitive. By definition, $\ll$ is irreflexive, antisymmetric, and transitive.

irreflexive: This follows immediately from the fact that $\longrightarrow$ and $\ll$ are irreflexive.

antisymmetric: To reach a contradiction assume that $L_i^{[a]} \Longrightarrow L_j^{[b]}$ and $L_j^{[b]} \Longrightarrow L_i^{[a]}$. Since $\longrightarrow$ and $\ll$ are antisymmetric, we can assume without loss of generality that $L_i^{[a]} \longrightarrow L_j^{[b]}$ and

[6] Lemma 4.3 proves that $\Longrightarrow$ is a total order.

$L_j^{[b]} \ll L_i^{[a]}$. Using the fact that $L_i^{[a]} \longrightarrow L_j^{[b]}$ along with Corollary 4.2 we can conclude that $L_i^{[a]} \ll L_j^{[b]}$ or $L_i^{[a]} = L_j^{[b]}$. However, this contradicts the fact that $L_j^{[b]} \ll L_i^{[a]}$.

transitive: For a contradiction assume that $L_i^{[a]} \Longrightarrow L_j^{[b]}$ and $L_j^{[b]} \Longrightarrow L_k^{[c]}$ but $L_i^{[a]} \not\Longrightarrow L_k^{[c]}$. Consider the case where $L_i^{[a]} \longrightarrow L_j^{[b]}$ and $L_j^{[b]} \ll L_k^{[c]}$ but $L_i^{[a]} \not\longrightarrow L_k^{[c]}$ and $L_i^{[a]} \not\ll L_k^{[c]}$. Corollary 4.2 and the fact that $L_i^{[a]} \longrightarrow L_j^{[b]}$ imply that $L_i^{[a]} \ll L_j^{[b]}$ or $L_i^{[a]} = L_j^{[b]}$. This fact along with the fact that $L_j^{[b]} \ll L_k^{[c]}$ implies that $L_i^{[a]} \ll L_k^{[c]}$. This contradicts that earlier assumption that $L_i^{[a]} \not\ll L_k^{[c]}$. Since $\longrightarrow$ and $\ll$ are transitive, the only other case is $L_i^{[a]} \ll L_j^{[b]}$ and $L_j^{[b]} \longrightarrow L_k^{[c]}$ but $L_i^{[a]} \not\longrightarrow L_k^{[c]}$ and $L_i^{[a]} \not\ll L_k^{[c]}$. We use the same reasoning as in the previous case to show that this case also cannot arise.

total: Consider any two label operations $L_i^{[a]}$ and $L_j^{[b]}$. When $i \neq j$ then $L_i^{[a]}$ and $L_j^{[b]}$ are ordered by $\ll$. When $i = j$ then $L_i^{[a]}$ and $L_j^{[b]}$ are ordered by $\longrightarrow$.

Since $\Longrightarrow$ is irreflexive, antisymmetric, transitive and total, we can conclude that $\Longrightarrow$ is a *total order.* ∎

Lemma 4.4 *β using the order $\Longrightarrow$ and choice function c satisfies axiom* **P0**.

Proof: This follows immediate from the definition of c, the fact that β is well-formed, and the definition of the SNAP and UPDATE actions. ∎

Lemma 4.5 *β using the order $\Longrightarrow$ and choice function c satisfies axiom* **P1**.

Proof: In order to simplify the notation in this proof, we write $L_i^{[a]} \ll L_j^{[b]}$ instead of $(t_i^{[a]}, i) \ll (t_j^{[b]}, j)$. From Lemma 4.3 we know that $\Longrightarrow$ is a total order. Part a of **P1**, *precedence*, follows immediately from the definition of $\Longrightarrow$. For part b of **P1**, *consistency*, let $S_i^{[a]}$ return $\bar{o}_i$. There are four cases to consider:

$c(i,a,j) \neq 0$ and $c(i,a,k) \neq 0$: $\Rightarrow$ If $j < k$ in $\bar{o}_i$ then, by the definition of $\bar{o}_i$ in the SNAP$_i$ action, $L_j^{[c(i,a,j)]} \ll L_k^{[c(i,a,k)]}$. By definition of $\Longrightarrow$ this shows that $L_j^{[c(i,a,j)]} \Longrightarrow L_k^{[c(i,a,k)]}$.
$\Leftarrow$ If $L_j^{[c(i,a,j)]} \Longrightarrow L_k^{[c(i,a,k)]}$ then either $L_j^{[c(i,a,j)]} \longrightarrow L_k^{[c(i,a,k)]}$ or $L_j^{[c(i,a,j)]} \ll L_k^{[c(i,a,k)]}$. When $L_j^{[c(i,a,j)]} \longrightarrow L_k^{[c(i,a,k)]}$ Corollary 4.2 and the fact that $j \neq k$ show that $L_j^{[c(i,a,j)]} \ll L_k^{[c(i,a,k)]}$. Now $j < k$ in $\bar{o}_i$ since $L_j^{[c(i,a,j)]} \ll L_k^{[c(i,a,k)]}$.

$c(i,a,j) = 0$ and $c(i,a,k) = 0$: In this case the definition of c show that the t_j and t_k read by $S_i^{[a]}(\text{SNAP})$ are equal to their initial values, which are 0. Now the definition of $\bar{o}_i$ in the SNAP action shows that $j < k$ in $\bar{o}_i$ if and only if $j < k$.

$c(i,a,j) = 0$ and $c(i,a,k) \neq 0$: Lemma 4.1 shows that $(t_j^{[c(i,a,j)]}, j) \ll (t_k^{[c(i,a,k)]}, k)$. Now the definition of $\bar{o}_i$ in the SNAP action shows that $j < k$ in $\bar{o}_i$.

$c(i,a,j) \neq 0$ and $c(i,a,k) = 0$: Lemma 4.1 shows that $(t_k^{[c(i,a,k)]}, k) \ll (t_j^{[c(i,a,j)]}, j)$. Now the definition of $\bar{o}_i$ in the SNAP action shows that $k < j$ in $\bar{o}_i$.

■

Lemma 4.6 *β using the order $\Longrightarrow$ and choice function c satisfies axiom* **P2.**

Proof: Consider $S_j^{[a]}$ with $c(j,a,i) > 0$. By definition of c, $L_i^{[c(j,a,i)]}(\text{UPDATE}) \Longrightarrow' S_j^{[a]}(\text{SNAP})$. Hence $S_j^{[a]} \not\longrightarrow L_i^{[c(j,a,i)]}$. In order to prove that the second part of the axiom holds for β we assume that there exists $L_i^{[b]}$ such that $L_i^{[c(j,a,i)]} \longrightarrow L_i^{[b]} \longrightarrow S_j^{[a]}$. This implies that $L_i^{[c(j,a,i)]}(\text{UPDATE}) \Longrightarrow' L_i^{[b]}(\text{UPDATE}) \Longrightarrow' S_j^{[a]}(\text{SNAP})$, which directly contradicts the definition of c. Now consider $S_j^{[a]}$ where $c(j,a,i) = 0$. The definition of c shows that there exists no $L_i^{[b]}(\text{UPDATE})$ such that $L_i^{[b]}(\text{UPDATE}) \Longrightarrow' S_j^{[a]}(\text{SNAP})$. Consequently, there exists no $L_i^{[b]}$ such that $L_i^{[b]} \longrightarrow S_j^{[a]}$. ■

Lemma 4.7 *β using the order $\Longrightarrow$ and choice function c satisfies axiom* **P3.**

Proof: Consider $S_i^{[a]} \longrightarrow S_j^{[b]}$, where $c(i,a,k) > 0$. By definition of c, $L_k^{[c(i,a,k)]}(\text{UPDATE}) \Longrightarrow' S_i^{[a]}(\text{SNAP}) \Longrightarrow' S_j^{[b]}(\text{SNAP})$. Now the definition of c and the fact that $c(i,a,k) \neq c(j,b,k)$ imply that $c(i,a,k) < c(j,b,k)$. When $c(i,a,k) = 0$ the fact that $c(i,a,k) \neq c(j,b,k)$ immediately shows that $c(i,a,k) < c(j,b,k)$. ■

Lemma 4.8 *β using the order $\Longrightarrow$ and choice function c satisfies axiom* **P4.**

Proof: Since $S_i^{[a]} \longrightarrow L_j^{[b]}$, $S_i^{[a]}(\text{SNAP}) \Longrightarrow' L_j^{[b]}(\text{SNAP})$. Furthermore, the definition of c and the fact that $c(i,a,k) > 0$ imply that $L_k^{[c(i,a,k)]}(\text{UPDATE}) \Longrightarrow' S_i^{[a]}(\text{SNAP})$. Consequently, $L_k^{[c(i,a,k)]}(\text{UPDATE}) \Longrightarrow' L_j^{[b]}(\text{SNAP})$. Now Lemma 4.1 implies that $(t_k^{[c(i,a,k)]}, k) \ll (t_j^{[b]}, j)$. Therefore the definition of $\Longrightarrow$ implies that $L_k^{[c(i,a,k)]} \Longrightarrow L_j^{[b]}$. ■

Lemma 4.9 *If a behavior β, where $\beta \in$ fairbehs*(UCTSS), *has a well-formed-input, then β is well-formed and response-live.*

Proof: Notice by inspecting the *precondition* clauses in the code of Figure 1 that for any equivalence class C_i of *part*(UCTSS), there is always at most one action enabled. Furthermore each action remains enabled until it is executed. Consequently, the actions must be executed in the sequence in which they are enabled. Furthermore, in a fair execution each enabled action will eventually be executed.

Now consider any fair execution that has a well-formed-input. The precondition-effects code in Figure 1 shows that the following sequence of actions is executed in response to a BEGINSCAN$_i$ input action: SNAP$_i(\bar{t}_i, \bar{v}_i)$ and ENDSCAN$_i(\bar{o}_i, \bar{v}_i)$. In response to a BEGINLABEL$_i(val_i)$ input action, the following sequence of actions is executed: SNAP$_i(\bar{t}_i, \bar{v}_i)$, UPDATE$_i((t_i, v_i), (nt_i, val_i))$, and ENDLABEL$_i$. Also, no actions of C_i are enabled between the execution of an ENDSCAN$_i(\bar{o}_i, \bar{v}_i)$ or ENDLABEL$_i$ action and the next execution of a BEGINSCAN$_i$ or BEGINLABEL$_i(val_i)$ action. Inspection of these action sequences and the definitions of well-formed-preserving and response-live, immediately shows that UCTSS is well-formed-preserving and response-live. ∎

We now have the necessary lemmas to show that UCTSS solves CTSS.

Lemma 4.10 UCTSS *solves* CTSS.

Proof: By inspection *exsig*(UCTSS) = *exsig*(CTSS). In order to show that *fairbehs*(UCTSS) $\subseteq$ *behs*(CTSS) we consider any behavior β such that $\beta \in$ *fairbehs*(UCTSS). If β does not have a well-formed-input, then $\beta \in$ *behs*(CTSS) trivially. So, assume that β has a well-formed-input. Now Lemma 4.9 shows that β is well formed. Define an order $\Longrightarrow$ and a choice function c as in Definition 4.2 and Definition 4.3 respectively. Now, Lemma 4.4, Lemma 4.5, Lemma 4.6, Lemma 4.7, and Lemma 4.8 show that β, $\Longrightarrow$ and c satisfy axioms **P0**–**P4**. Hence $\beta \in$ *behs*(CTSS). ∎

5 A Bounded Concurrent Timestamp System

In this section we present our bounded implementation of a concurrent timestamp system, BCTSS. BCTSS differs from UCTSS in three ways: the structure of the labels, the order between labels, and the manner in which NEWLABEL$_i$ determines new labels. In all other aspects BCTSS

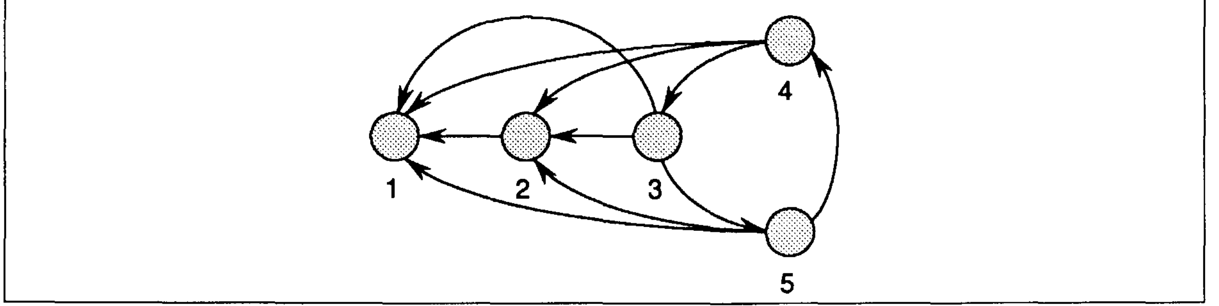


Figure 4: A graphical illustration of the $\prec_{\mathcal{A}}$ order between the elements of $\mathcal{A} = \{1 \ldots 5\}$

and UCTSS are identical. Recall that a label in UCTSS is an element of $\Re^+$. In BCTSS, labels are taken from a different domain. In order to construct the new domain we introduce the set $\mathcal{A} = \{1 \ldots 5\}$. We define the order $\prec_{\mathcal{A}}$ and the function NEXT on the elements of $\mathcal{A}$.

$$1 \prec_{\mathcal{A}} 2, 3, 4, 5; \quad 2 \prec_{\mathcal{A}} 3, 4, 5; \quad 3 \prec_{\mathcal{A}} 4; \quad 4 \prec_{\mathcal{A}} 5; \quad 5 \prec_{\mathcal{A}} 3.$$

The graph in Figure 4 represents $\prec_{\mathcal{A}}$, where $a \prec_{\mathcal{A}} b$ iff there is a directed edge from b to a.

$$\text{NEXT}(k) = \begin{cases} k+1 & \text{if } k \in \{1, 2, 3, 4\} \\ 3 & \text{if } k = 5 \end{cases}$$

A BCTSS label is an element of $\mathcal{A}^{n-1}$, where n is the number of processes in the system. We refer to elements of $\mathcal{A}^{n-1}$ using array notation. Specifically, the hth digit of label ℓ will be denoted by $\ell[h]$. Since we have redefined the label type, we must specify the order that is to be used between elements of $\mathcal{A}^{n-1}$ for the $\ll$ order in the SNAP_i action. The order between elements of $\mathcal{A}^{n-1}$ is represented by the symbol $\prec$ and will be a lexicographical order based on $\prec_{\mathcal{A}}$.

Definition 5.1 ($\prec$ order) $\ell_i \prec \ell_j$ iff there exists $h \in \{1 \ldots n-1\}$ such that $\ell_i[h'] = \ell_j[h']$ for all $h' < h$ and $\ell_i[h] \prec_{\mathcal{A}} \ell_j[h]$. ∎

Example 5.1 $4 \ldots 4.5.2 \prec 4 \ldots 4.3.1$

Lemma 5.1 *If ℓ_1 and ℓ_2 are elements of $\mathcal{A}^{n-1}$ then exactly one of the following is true: $\ell_1 \prec \ell_2$, $\ell_2 \prec \ell_1$, or $\ell_1 = \ell_2$.*

Proof: If $a, b \in \mathcal{A}$, then by definition of $\prec_{\mathcal{A}}$ exactly one of the following is true: $a \prec_{\mathcal{A}} b$, $b \prec_{\mathcal{A}} a$ or $a = b$. The lemma now follows since $\prec$ is a lexicographical order defined by $\prec_{\mathcal{A}}$. ∎

We define the following notation and functions for BCTSS labels:

Definition 5.2 ($\stackrel{h}{=}$ equivalence relation) For any $h \in \{0 \dots n-1\}$, $\ell_1 \stackrel{h}{=} \ell_2$ iff $\ell_1[h'] = \ell_2[h']$ for all $h' \leq h$. Note that $\ell_1 \stackrel{n-1}{=} \ell_2$ implies that $\ell_1 = \ell_2$. ∎

Definition 5.3 (NEXTLABEL) For any $h \in \{1 \dots n-1\}$, $\ell' = \text{NEXTLABEL}(\ell, h)$ iff $\ell' \stackrel{h-1}{=} \ell$, $\ell'[h] = \text{NEXT}(\ell[h])$ and $\ell'[h'] = 1$ for all $h' \in \{h+1 \dots n-1\}$. ∎

Definition 5.4 (CYCLE) For any $h \in \{1 \dots n-1\}$, $\ell' \in \text{CYCLE}(\ell, h)$ iff $\ell' \stackrel{h-1}{=} \ell$ and $\ell'[h] \in \{3, 4, 5\}$. ∎

Lemma 5.2 *A set $\mathcal{L}$ of labels is not totally ordered by $\prec$ iff there exist $\ell_1, \ell_2, \ell_3 \in \mathcal{L}$ and $h \in \{1 \dots n-1\}$ such that $\ell_1 \stackrel{h-1}{=} \ell_2 \stackrel{h-1}{=} \ell_3$ and $\{\ell_1[h], \ell_2[h], \ell_3[h]\} = \{3, 4, 5\}$.*

Proof: $\Rightarrow$ The $\prec$ ordering on $\mathcal{L}$ is irreflexive by definition and antisymmetric by Lemma 5.1. Therefore, it must be that transitivity does not hold. Specifically there exist $\ell_1, \ell_2, \ell_3 \in \mathcal{L}$ such that $\ell_1 \prec \ell_2 \prec \ell_3$, and $\ell_1 \not\prec \ell_3$. By Lemma 5.1 it cannot be that $\ell_1 = \ell_3$, therefore $\ell_3 \prec \ell_1$. Since $\prec$ is a lexicographical order, there must exist $h \in \{1 \dots n-1\}$ such that $\ell_1 \stackrel{h-1}{=} \ell_2 \stackrel{h-1}{=} \ell_3$ and $\ell_1[h] \prec_{\mathcal{A}} \ell_2[h] \prec_{\mathcal{A}} \ell_3[h]$ and $\ell_1[h] \not\prec_{\mathcal{A}} \ell_3[h]$. Now by definition of $\mathcal{A}$, $\{\ell_1[h], \ell_2[h], \ell_3[h]\} = \{3, 4, 5\}$.

$\Leftarrow$ By definition of $\mathcal{A}$ we can conclude without loss of generality that $\ell_1[h] \prec_{\mathcal{A}} \ell_2[h] \prec_{\mathcal{A}} \ell_3[h]$ and $\ell_1[h] \not\prec_{\mathcal{A}} \ell_3[h]$. Since $\ell_1 \stackrel{h-1}{=} \ell_2 \stackrel{h-1}{=} \ell_3$ and $\prec$ is a lexicographical order, $\ell_1 \prec \ell_2 \prec \ell_3$, and $\ell_1 \not\prec \ell_3$. Hence, ℓ_1, ℓ_2, and ℓ_3 are not totally ordered. ∎

We now define some functions on the states of BCTSS. In order to reason about the states of the system we introduce the notation $b.x$ to refer to the variable x in state b. For a state b and any label ℓ in state b:

Definition 5.5 (AGREE) For any $h \in \{0 \dots n-1\}$, $\text{AGREE}(b.\ell, h) = \{j \mid b.t_j \stackrel{h}{=} b.\ell\}$. ∎

Definition 5.6 (NUM) For any $h \in \{0 \dots n-1\}$, $\text{NUM}(b.\ell, h) = |\text{AGREE}(b.\ell, h)|$. ∎

Definition 5.7 (NUM_i) For any $h \in \{0 \dots n-1\}$, $\text{NUM}_i(b.\ell, h) = |\text{AGREE}(b.\ell, h) - \{i\}|$. ∎

Definition 5.8 (choice vector) A *choice vector* for state b is any vector $(b.\ell_1 \dots b.\ell_n)$ such that $b.\ell_i \in \{b.t_i, b.nt_i\}$ for each i. ∎

```
FULL_i(h), h ∈ {1...n−1}
  if NUM_i(t_max, h) ≥ n − h
    then return (true)
    else return (false)

NEWLABEL_i(t̄_i)
  if i ≠ i_max
    then h' ← minimum h ∈ {1...n−1} such that FULL_i(h) = true
         return (NEXTLABEL(t_max, h'))
```

Figure 5: Code for NEWLABEL$_i$ of BCTSS

Definition 5.9 (TOT) TOT(b) = *true* iff the set of values in every choice vector is totally ordered by $\prec$; otherwise TOT(b) = *false.* ■

Recall that the second difference between UCTSS and BCTSS is the $\ll$ order that is used in SNAP$_i$. We define $\ll$ for BCTSS lexicographically.

Definition 5.10 ($\ll$ order) $(\ell_i, i) \ll (\ell_j, j)$ iff either $\ell_i \prec \ell_j$ or $\ell_i = \ell_j$ and $i < j$. ■

In any state b in which TOT(b) = *true*, $\ll$ defines a total order.

We now define $b.t_{max}$ and $b.i_{max}$ for a state, b, in which TOT(b) = *true*. Consider the choice vector $(b.t_1 \ldots b.t_n)$. Since TOT(b) = *true*, there must exist $i \in \{1 \ldots n\}$ such that, for all $j \neq i$ and $j \in \{1 \ldots n\}$, $b.t_j \preceq b.t_i$. Let $b.t_{max} = b.t_i$. Let $b.i_{max}$ be the largest index j such that $b.t_j = b.t_{max}$.

The final difference between BCTSS and UCTSS is in the code for NEWLABEL$_i$. Recall that in UCTSS, NEWLABEL$_i$ nondeterministically picks a real number that is larger than t_{max}. In BCTSS, NEWLABEL$_i$ also picks the new label based on t_{max}. In states in which TOT(b) = *true*, $b.t_{max}$ and $b.i_{max}$ are defined. We let NEWLABEL$_i$ be a no–op for states in which TOT(b) = *false*. In Section 6 we will show that TOT(b) = *true* for all reachable states. When i_{max} is defined and $i \neq i_{max}$, NEWLABEL$_i$ finds the minimum h such that at least $n - h$ t-labels, excluding t_i, agree with the prefix of t_{max} up to and including the hth digit. Then the new label is the same as t_{max} for the first $h-1$ digits, it differs from t_{max} at the hth digit based on the function NEXT, and its remaining digits are equal to 1. The code for NEWLABEL$_i$ of BCTSS is given in Figure 5.

NEWLABEL_i finds the minimum integer h such that $\text{FULL}_i(h)$ returns *true*. We now show that such an h exists in $\{1 \ldots n-1\}$. The code that finds h is executed only when $i \neq i_{max}$. Notice that $\text{NUM}_i(t_{max}, n-1) \geq 1$ when $i \neq i_{max}$, hence $\text{FULL}_i(n-1) = true$.

The initial values for the labels in BCTSS are: $t_i = nt_i = 1^{n-1}$, $\bar{o_i} = (1 \ldots n)$, $\bar{v_i} = (v_o \ldots v_o)$, $\bar{t_i} = (1^{n-1} \ldots 1^{n-1})$, $v_i = val_i = v_o$, $op_i = \text{NIL}$, and $pc_i = \text{NIL}$.

6 Invariants

For use in the simulation proof we define the following invariants:

Theorem 6.1 *If b is a reachable state of* BCTSS *then, for all $i \in \{1 \ldots n\}$:*

I: $\text{TOT}(b) = \text{true}$.

II: *If $i = b.i_{max}$ then $b.t_i = b.nt_i$.*

III: *If $b.t_{max} \prec b.nt_i$ then there exists $h \in \{1 \ldots n-1\}$ such that $b.nt_i = \text{NEXTLABEL}(b.t_{max}, h)$.*

IV: *If $b.nt_i \preceq b.t_{max}$ then for any $h \in \{1 \ldots n-1\}$, if $b.t_i \stackrel{h}{=} b.t_{max}$ then $b.nt_i \stackrel{h}{=} b.t_{max}$.*

V: *For any $h \in \{1 \ldots n-1\}$, if $b.nt_i \in \text{CYCLE}(b.t_{max}, h)$ then $b.t_i \stackrel{h-1}{=} b.t_{max}$.*

VI: *For any $h \in \{1 \ldots n-1\}$,*

a: if $b.nt_i = \text{NEXTLABEL}(b.t_{max}, h)$ then $\text{NUM}_i(b.t_{max}, h-1) \geq n-h$.

b: if $b.t_{max}[h] \neq 1$ then $\text{NUM}(b.t_{max}, h-1) \geq n-h+1$.

■

I, II, and III are used in the simulation proof. We use an induction argument to show that all reachable states of BCTSS satisfy these invariants. The purpose of invariants IV - VI is to strengthen the induction hypothesis enough so that I can be proven. The only action that can cause invariant I to be violated is SNAP_i when $op_i = \text{LABEL}_i$. Specifically, we must show that the new nt_i picked by NEWLABEL_i does not introduce any cycles in the $\prec$ order of the t-labels and nt-labels. Since the NEWLABEL_i code can examine the all of the t-labels, the code can be written to avoid any cycles involving nt_i and the t-labels. However, the NEWLABEL_i code

cannot examine the local nt-labels of the other processes. In order to show that cycles that include nt_i and nt-labels are avoided, invariants IV and V are used to limit the possible values of the nt-labels based on the corresponding t-labels.

For example invariant IV implies that $nt_i \stackrel{h}{=} t_i$ when $t_i \stackrel{h}{=} t_{max}$ for all $nt_i \preceq t_{max}$. If nt_i is in the cycle at level h, in other words $nt_i[h] \in \{3,4,5\}$, then invariant V states that $nt_i \stackrel{h-1}{=} t_i$. Now assume that NEWLABEL$_i$ picks $nt_i =$ NEXTLABEL(t_{max}, h). Then the code for NEWLABEL$_i$, using the function FULL$_i$, limits the number number of t-labels that are $\stackrel{h-1}{=} t_{max}$ and consequently the number of t-labels that are $\stackrel{h-1}{=} nt_i$. Now invariant V can be used to limit the number of nt-labels that could, by being in the cycle at level h, cause a cycle to occur with the new nt_i.

Invariant III gives information about the structure of nt-labels that are $\succ t_{max}$. This information is used to determine how the new nt_i is ordered with respect to any nt-labels that are $\succ t_{max}$. Finally invariant VIb is used to prove invariant V, and invariant VIa is used to prove VIb. If a new label nt_i is picked in the cycle at level h then it must be that $t_{max}[h] \neq 1$; hence VIb applies. VIb says that NUM$(t_{max}, h-1) \geq n - h + 1$. The code for NEWLABEL$_i$ insures that NUM$_i(t_{max}, h-1) < n - h + 1$. Thus it must be the case that $t_i \stackrel{h-1}{=} t_{max}$. This is precisely what is required to prove invariant V.

The proof of Theorem 6.1 uses induction. It depends on a series of lemmas, one for the initial state and one for each action in the inductive step.

Lemma 6.2 *The initial state b of* BCTSS, *satisfies invariants I - VI.*

Proof: This follows from the fact that $b.t_i = b.nt_j = 1^{n-1}$ for all $i, j \in \{1 \ldots n\}$. ■

Lemma 6.3 *Let b be a state of* BCTSS *that satisfies I - VI. If (b, π, b') is a step of* BCTSS *where $\pi \in \{$BEGINSCAN$_k$, ENDSCAN$_k(\bar{o}_k, \bar{v}_k)$, BEGINLABEL$_k(val_k)$, ENDLABEL$_k\}$ for any k, then b' satisfies I - VI.*

Proof: None of the t-labels or nt-labels change as a result of π. This suffices to show that b' satisfies I - VI. ■

Lemma 6.4 *Let b be a state of* BCTSS *satisfying I - VI. If $(b,$ UPDATE$_k((t_k, v_k), (nt_k, val_k)), b')$ is a step of* BCTSS *for any k, then b' satisfies I - VI.*

Proof: The proof is divided into a series of claims. By invariant I for state b, $b.t_{max}$ and $b.i_{max}$ are defined. We split the argument into two cases: $k = b.i_{max}$ and $k \neq b.i_{max}$. Consider $k = b.i_{max}$ first.

Claim 6.4.1 *If $k = b.i_{max}$, then b' satisfies I - VI.*

Proof: By invariant II for state b, $b.t_k = b.nt_k$. Thus, none of the t-labels or nt-labels change for BCTSS. This suffices to show that b' satisfies I - VI. ■

So assume that $k \neq b.i_{max}$ for the remainder of the proof.

Claim 6.4.2 *If $k \neq b.i_{max}$ then I is true in b'.*

Proof: Assume for a contradiction that $\text{TOT}(b') = \textit{false}$. Since $\text{TOT}(b) = \textit{true}$ and t_k is the only label that changes, the choice vector whose values are not totally ordered must include $b'.t_k$. Now consider the same choice vector except that we substitute $b'.nt_k$ for $b'.t_k$. Since $b'.t_k = b'.nt_k$, this new choice vector's values are also not totally ordered. Since none of the labels in this new choice vector change as a result of the action, the same choice vector must not have had its values totally ordered in state b. However this contradicts the assumption that $\text{TOT}(b) = \textit{true}$. ■

Having proved invariant I we now know that $b'.i_{max}$ and $b'.t_{max}$ are defined. The proof for II - VI is subdivided into the following two cases: $b.nt_k \preceq b.t_{max}$ and $b.t_{max} \prec b.nt_k$. Assume first that $b.nt_k \preceq b.t_{max}$.

Claim 6.4.3 *If $k \neq b.i_{max}$ and $b.nt_k \preceq b.t_{max}$ then $b'.t_{max} = b.t_{max}$ and $b'.i_{max} = b.i_{max}$ or $b'.i_{max} = k$.*

Proof: Let $z = b.i_{max}$, then $b.t_z = b.t_{max}$ and $z \neq k$. We show first that $b'.t_i \preceq b.t_z$ for all i. First consider $i \neq k$. Since t_k is the only label that changes, $b'.t_i = b.t_i$. Therefore, the fact that $b.t_i \preceq b.t_z$ implies that $b'.t_i \preceq b.t_z$. Now let $i = k$. As a result of the action, $b'.t_i = b.nt_i$. By assumption $b.nt_i \preceq b.t_z$, so $b'.t_i \preceq b.t_z$. Since $z \neq k$, t_z does not change, so we can conclude that $b'.nt_i \preceq b'.t_z$ for all i. This implies that $b'.t_z = b'.t_{max}$. The following identity now establishes the first part of the claim: $b.t_{max} = b.t_z = b'.t_z = b'.t_{max}$.

Let $S = \{i | b.t_i = b.t_{max}\}$ and $S' = \{i | b'.t_i = b'.t_{max}\}$ Then, $b.i_{max} = \text{MAX}(S)$ and $b'.i_{max} = \text{MAX}(S')$. Since t_k is the only t-label that changes and $b'.t_{max} = b.t_{max}$, $S' = S$ or $S' = S - \{k\}$ or $S' = S \cup \{k\}$. When $S' = S$ then $\text{MAX}(S') = \text{MAX}(S)$. Let $z = b.i_{max}$. Since $k \neq b.i_{max}$, the definition of $b.i_{max}$ shows that $z \in S$ and $k < z$ when $k \in S$. Consequently, when $S' = S - \{k\}$ then $\text{MAX}(S') = \text{MAX}(S)$. Finally, when $S' = S \cup \{k\}$ then $\text{MAX}(S') = \text{MAX}(S)$ or $\text{MAX}(S') = k$. This shows that $b'.i_{max} = b.i_{max}$ or $b'.i_{max} = k$. ■

Claim 6.4.4 *If $k \neq b.i_{max}$ and $b.nt_k \preceq b.t_{max}$ then $\text{NUM}(b'.t_{max}, h) \geq \text{NUM}(b.t_{max}, h)$ and $\text{NUM}_i(b'.t_{max}, h) \geq \text{NUM}_i(b.t_{max}, h)$ for all i and h.*

Proof: The Claim follows immediately if we show that $\text{AGREE}(b'.t_{max}, h) \supseteq \text{AGREE}(b.t_{max}, h)$. Suppose $i \in \text{AGREE}(b.t_{max}, h)$. If $i \neq k$, then since t_i does not change and, by Claim 6.4.3, t_{max} does not change, $i \in \text{AGREE}(b'.t_{max}, h)$. Now consider $i = k$. By definition of AGREE, $b.t_i \stackrel{h}{=} b.t_{max}$. Since $b.nt_i \preceq b.t_{max}$, IV for state b implies that $b.nt_i \stackrel{h}{=} b.t_{max}$. As a result of the action $b'.t_i = b.nt_i$, so $b'.t_i \stackrel{h}{=} b.t_{max}$. This fact along with the fact that t_{max} does not change implies that $i \in \text{AGREE}(b'.t_{max}, h)$. ■

Claim 6.4.5 *If $k \neq b.i_{max}$ and $b.nt_k \preceq b.t_{max}$ then b' satisfies II - VI.*

Proof: We proceed with a case analysis. Consider any $i \in \{1 \ldots n\}$ and $h \in \{1 \ldots n-1\}$.

II: Suppose $i = b'.i_{max}$. By Lemma 6.4.3, $i = k$ or $i = b.i_{max}$. First consider $i = k$. As a direct consequence of the action, $b'.t_i = b'.nt_i$. Now consider $i = b'.i_{max}$ where $i \neq k$. In this case II holds for b' since t_i and nt_i do not change, and II holds for b.

III: III holds for b' since t_{max} and nt_i do not change, and III holds for b.

IV: First consider $i = k$. As a consequence of the action $b'.t_i = b'.nt_i$. Hence, $b'.t_i \stackrel{h}{=} b'.t_{max}$ implies that $b'.nt_i \stackrel{h}{=} b'.t_{max}$ for all h. Now consider $i \neq k$. Since IV holds in state b, and t_{max}, t_i and nt_i do not change, IV holds for state b'.

V: First consider $i = k$. $b'.nt_i \in \text{CYCLE}(b'.t_{max}, h)$ and the definition of CYCLE imply that $b'.nt_i \stackrel{h-1}{=} b'.t_{max}$. As a consequence of the action, $b'.t_i = b'.nt_i$. Hence, $b'.t_i \stackrel{h-1}{=} b'.t_{max}$. Now consider $i \neq k$. In this case V is true in b' since t_i, nt_i , and t_{max} do not change and V is true in b.

VI: Since nt_i and t_{max} do not change, $b'.nt_i = \text{NEXTLABEL}(b'.t_{max}, h)$ implies that $b.nt_i = \text{NEXTLABEL}(b.t_{max}, h)$, and $b'.t_{max}[h] \neq 1$ implies that $b.t_{max}[h] \neq 1$. By Claim 6.4.4, $\text{NUM}(b'.t_{max}, h) \geq \text{NUM}(b.t_{max}, h)$ and $\text{NUM}_i(b'.t_{max}, h) \geq \text{NUM}_i(b.t_{max}, h)$. Hence, VI holds for state b' since it holds for state b.

■

Claim 6.4.5 shows that II - VI hold when $b.nt_k \preceq b.t_{max}$. For the remainder of the proof assume that $b.t_{max} \prec b.nt_k$.

Claim 6.4.6 *If $k \neq b.i_{max}$ and $b.t_{max} \prec b.nt_k$ then $b'.t_{max} = b'.t_k$ and $b'.i_{max} = k$.*

Proof: We proceed by showing that $b'.t_i \prec b'.t_k$ for all $i \neq k$. From the definition of t_{max} and the assumption that $b.t_{max} \prec b.nt_k$, we know that $b.t_i \preceq b.t_{max} \prec b.nt_k$. Let $z = b.i_{max}$ then $b.t_z = b.t_{max}$ and $z \neq k$. Since $k \neq z$, $k \neq i$, and $b.t_z = b.t_{max}$, there exists a choice vector that includes the values $b.t_i, b.t_{max}$, and $b.nt_k$. Since $\text{TOT}(b) = true$, the values in this choice vector are totally ordered. Hence, $b.t_i \preceq b.t_{max} \prec b.nt_k$ implies that $b.t_i \prec b.nt_k$. As a result of the action $b.nt_k = b'.t_k$ and t_i does not change. Therefore, $b.t_i \prec b.nt_k$ implies that $b'.t_i \prec b'.t_k$. Hence $b'.t_{max} = b'.t_k$. Since k is the only process index for which $b'.t_{max} = b'.t_k$, $b'.i_{max} = k$. ■

The following Claim lists some of the properties of $b'.t_{max}$.

Claim 6.4.7 *If $k \neq b.i_{max}$ and $b.t_{max} \prec b.nt_k$ then there exists $h' \in \{1 \ldots n-1\}$ such that:*

1. $b'.t_{max} = b'.t_k = b'.nt_k = b.nt_k = \text{NEXTLABEL}(b.t_{max}, h')$.

2. $b'.t_{max}[h] = 1$ *for all* $h > h'$.

3. *For all* i, $b'.nt_i \stackrel{h'}{=} b'.t_{max}$ *implies that* $b'.nt_i = b'.t_{max}$.

4. *There exists no* $i \neq k$ *such that* $b'.t_i \stackrel{h'}{=} b'.t_{max}$.

5. $\text{NUM}(b'.t_{max}, h) \geq \text{NUM}(b.t_{max}, h)$ *and* $\text{NUM}_i(b'.t_{max}, h) \geq \text{NUM}_i(b.t_{max}, h)$ *for all* i *and all* $h < h'$.

Proof: By invariant III for state b and the assumption that $b.t_{max} \prec b.nt_k$, we conclude that $b.nt_k = \text{NEXTLABEL}(b.t_{max}, h')$ for $h' \in \{1 \ldots n-1\}$. Fix h'.

1: By Claim 6.4.6 $b'.t_{max} = b'.t_k$. The fact that $b'.t_k = b'.nt_k = b.nt_k$ is a direct consequence of the action $\text{UPDATE}_k((t_k, v_k), (nt_k, val_k))$. Finally, we have already shown that $b.nt_k = \text{NEXTLABEL}(b.t_{max}, h')$.

2: This follows directly from the definition of NEXTLABEL.

3: Suppose that $b'.nt_i \stackrel{h'}{=} b'.t_{max}$. First consider $i \neq k$. The fact that nt_i does not change and part 1 of the claim show that $b.nt_i = b'.nt_i \stackrel{h'}{=} b'.t_{max} = \text{NEXTLABEL}(b.t_{max}, h')$. Consequently, $b.nt_i \stackrel{h'}{=} \text{NEXTLABEL}(b.t_{max}, h')$. Now the definition of NEXTLABEL implies that $b.nt_i \stackrel{h'-1}{=} b.t_{max}$ and $b.nt_i[h'] = \text{NEXT}(b.t_{max}[h'])$. Thus $b.t_{max} \prec b.nt_i$. Now III for state b implies that $b.nt_i = \text{NEXTLABEL}(b.t_{max}, h)$ for some $h \in \{1 \ldots n-1\}$. Since $b.nt_i[h'] = \text{NEXT}(b.t_{max}[h'])$, $h = h'$. Hence, $b'.nt_i = b.nt_i = \text{NEXTLABEL}(b.t_{max}, h') = b'.t_{max}$. Now consider $i = k$. In this case $b'.t_{max} = b'.nt_k$ by part 1 of the claim.

4: We proceed by contradiction. Assume that there exists $i \neq k$ such that $b'.t_i \stackrel{h'}{=} b'.t_{max}$. Since t_i does not change as a result of the action, $b.t_i = b'.t_i \stackrel{h'}{=} b'.t_{max} = \text{NEXTLABEL}(b.t_{max}, h')$. Consequently, $b.t_i \stackrel{h'}{=} \text{NEXTLABEL}(b.t_{max}, h')$. Now the definition of NEXTLABEL implies that $b.t_i \stackrel{h'-1}{=} b.t_{max}$ and $b.t_i[h'] = \text{NEXT}(b.t_{max}[h'])$. Thus $b.t_{max} \prec b.t_i$. This contradicts the definition of $b.t_{max}$.

5: Let $h < h'$. Part 5 of the Claim follows immediately if we show that $\text{AGREE}(b'.t_{max}, h) \supseteq \text{AGREE}(b.t_{max}, h)$. Suppose $i \in \text{AGREE}(b.t_{max}, h)$. If $i \neq k$, then t_i does not change. By part 1 of claim and the definition of NEXTLABEL, $b'.t_{max} \stackrel{h}{=} b.t_{max}$. Now the definition of AGREE implies that $i \in \text{AGREE}(b'.t_{max}, h)$. Now consider $i = k$. Part 1 of the claim shows that $b'.t_i = b'.t_{max}$. Hence $i \in \text{AGREE}(b'.t_{max}, h)$. ■

The remainder of the proof is structured as a series of claims, one for each of the five remaining invariants. Fix h' to be the h' defined by Claim 6.4.7. Parts 1-5 of Claim 6.4.7 will be used throughout the remaining claims.

Claim 6.4.8 *If $k \neq b.i_{max}$ and $b.t_{max} \prec b.nt_k$ then II is true in b'.*

Proof: By Claim 6.4.6 $b'.i_{max} = k$. Part 1 of Claim 6.4.7 shows that $b'.t_k = b'.nt_k$. ■

Claim 6.4.9 *If $k \neq b.i_{max}$ and $b.t_{max} \prec b.nt_k$ then III is true in b'.*

Proof: Consider any i such that $b'.t_{max} \prec b'.nt_i$. By part 1 of Claim 6.4.7, $b'.t_{max} = b'.nt_k$ so $b'.t_{max} \prec b'.nt_i$ implies that $i \neq k$. Furthermore, nt_i does not change as a result of the action and part 1 of Claim 6.4.7 shows that $b'.t_{max} = b.nt_k$. Hence $b'.t_{max} \prec b'.nt_i$ implies that $b.nt_k \prec b.nt_i$. By assumption $b.t_{max} \prec b.nt_k$, so $b.t_{max} \prec b.nt_k \prec b.nt_i$. Now consider two cases, $i = b.i_{max}$ and $i \neq b.i_{max}$. When $i = b.i_{max}$, invariant II shows that $b.t_{max} = b.nt_i$. This implies that $b.nt_i \prec b.nt_k \prec b.nt_i$ which is impossible by Lemma 5.1. Therefore, it must be that $i \neq b.i_{max}$. Since $b.i_{max} \neq i$ and $b.i_{max} \neq k$ there must exist a choice vector that includes the values $b.t_{max}, b.nt_k$, and $b.nt_i$. Since $\text{TOT}(b) = true$, the values in this choice vector are totally ordered. Hence, $b.t_{max} \prec b.nt_k \prec b.nt_i$ implies that $b.t_{max} \prec b.nt_i$. Now III for state b and the fact that nt_i does not change show that $b'.nt_i = \text{NEXTLABEL}(b.t_{max}, h)$ for some $h \in \{1 \ldots n-1\}$. Since $b'.nt_i = \text{NEXTLABEL}(b.t_{max}, h)$, $b'.t_{max} = \text{NEXTLABEL}(b.t_{max}, h')$, and $b'.t_{max} \prec b'.nt_i$, it must be that $h < h'$. Hence $b'.nt_i = \text{NEXTLABEL}(b'.t_{max}, h)$, which directly implies that I holds for state b'. ■

Claim 6.4.10 *If $k \neq b.i_{max}$ and $b.t_{max} \prec b.nt_k$ then IV is true in b'.*

Proof: Let $b.'nt_i \preceq b'.t_{max}$. First consider $i = k$. By part 1 of Lemma 6.4.7, $b'.nt_k = b'.t_{max}$, which directly implies IV. Now consider $i \neq k$ and any h:

$h < h'$: Part 1 of Claim 6.4.7 and the definition of NEXTLABEL show that $b'.t_{max} \stackrel{h}{=} b.t_{max}$ when $h < h'$. Now consider two cases: $b.nt_i \preceq b.t_{max}$ and $b.nt_i \not\preceq b.t_{max}$. When $b.nt_i \preceq b.t_{max}$, IV for state b shows that $b.t_i \stackrel{h}{=} b.t_{max}$ implies that $b.nt_i \stackrel{h}{=} b.t_{max}$. Now IV is true in b' since t_i and nt_i do not change and $b'.t_{max} \stackrel{h}{=} b.t_{max}$. Now consider the case $b.nt_i \not\preceq b.t_{max}$. By Lemma 5.1, $b.t_{max} \prec b.nt_i$. Now III for state b shows that $b.nt_i = \text{NEXTLABEL}(b.t_{max}, h_i)$ for some $h_i \in \{1 \ldots n-1\}$. Furthermore, Since nt_i does not change, the assumption that $b'.nt_i \preceq b'.t_{max}$ implies that $b.nt_i \preceq b'.t_{max}$. Finally, part 1 of Claim 6.4.7 shows that $b'.t_{max} = \text{NEXTLABEL}(b.t_{max}, h')$. Using these facts and the definition of NEXTLABEL we can conclude that $h_i > h'$. Therefore, $b.nt_i \stackrel{h}{=} b'.t_{max}$. Since nt_i does not change, this implies that $b'.nt_i \stackrel{h}{=} b'.t_{max}$. This suffices to show that IV is true in b'.

$h \geq h'$: Part 4 of Claim 6.4.7 shows that $b'.t_i \stackrel{h}{\neq} b'.t_{max}$. Hence, IV is vacuously true in b'.

■

Claim 6.4.11 *If $k \neq b.i_{max}$ and $b.t_{max} \prec b.nt_k$ then V is true in b'.*

Proof: Suppose $b'.nt_i \in \text{CYCLE}(b'.t_{max}, h)$ for some i and h. The definition of CYCLE implies that $b'.nt_i \overset{h-1}{=} b'.t_{max}$. We consider two cases:

$h \leq h'$: First consider $i \neq k$. Part 1 of Claim 6.4.7 and the definition of NEXTLABEL show that $b'.t_{max} \overset{h-1}{=} b.t_{max}$. Thus, V is true in b' since t_i and nt_i do not change, $\text{CYCLE}(b'.t_{max}, h)$ depends only on $b'.t_{max}[1 \ldots h-1]$, and V is true in b. Now let $i = k$. In this case, part 1 of Claim 6.4.7 shows that $b'.t_i = b'.t_{max}$. This suffices to show V.

$h > h'$: Since $b'.nt_i \overset{h-1}{=} b'.t_{max}$ and $h > h'$, it follows that $b'.nt_i \overset{h'}{=} b'.t_{max}$. Thus part 3 of Claim 6.4.7 implies that $b'.nt_i = b'.t_{max}$. By part 2 of Claim 6.4.7, $b'.t_{max}[h] = 1$. Thus $b'.nt_i[h] = 1$, which implies that $b'.nt_i \notin \text{CYCLE}(b'.t_{max}, h)$. This contradicts our original assumption that $b'.nt_i \in \text{CYCLE}(b'.t_{max}, h)$. Therefore this case cannot arise.

■

Claim 6.4.12 *If $k \neq b.i_{max}$ and $b.t_{max} \prec b.nt_k$ then VIb is true in b'.*

Proof: Assume that $b'.t_{max}[h] \neq 1$. We proceed with a case analysis:

$h < h'$: Part 1 of Claim 6.4.7 and the definition of NEXTLABEL show that $b'.t_{max} \overset{h}{=} b.t_{max}$. Thus $b'.t_{max}[h] \neq 1$ implies that $b.t_{max}[h] \neq 1$. Since $b.t_{max}[h] \neq 1$ and VIb is true for b, $\text{NUM}(b.t_{max}, h-1) \geq n - h + 1$. By part 5 of Claim 6.4.7 $\text{NUM}(b'.t_{max}, h-1) \geq \text{NUM}(b.t_{max}, h-1)$. Thus, $\text{NUM}(b'.t_{max}, h-1) \geq n - h + 1$ which implies that VIb is true for b'.

$h = h'$ and $b.t_{max}[h] \neq 1$: Since $b.t_{max}[h] \neq 1$ and VIb is true for b, $\text{NUM}(b.t_{max}, h-1) \geq n-h+1$. By part 5 of Claim 6.4.7 $\text{NUM}(b'.t_{max}, h-1) \geq \text{NUM}(b.t_{max}, h-1)$. Thus, $\text{NUM}(b'.t_{max}, h-1) \geq n - h + 1$ which implies that VIb is true for b'.

$h = h'$ and $b.t_{max}[h] = 1$: Part 1 of Claim 6.4.7 and the fact that $h' = h$ imply that $b.nt_k = \text{NEXTLABEL}(b.t_{max}, h)$. Since $b.nt_k = \text{NEXTLABEL}(b.t_{max}, h)$ and VIa is true for state b, $\text{NUM}_k(b.t_{max}, h-1) \geq n - h$. By part 5 of Claim 6.4.7 $\text{NUM}_k(b'.t_{max}, h-1) \geq \text{NUM}_k(b.t_{max}, h-1)$. Thus, $\text{NUM}_k(b'.t_{max}, h-1) \geq n - h$. Since $b'.t_{max} = b'.t_k$, $k \in$

$\text{AGREE}(b'.t_{max}, h)$. Therefore $\text{NUM}(b'.t_{max}, h-1) > \text{NUM}_k(b'.t_{max}, h-1) \geq n-h$. Thus, $\text{NUM}(b'.t_{max}, h-1) \geq n-h+1$, which implies that VIb is true for b'.

$h > h'$: Part 2 of Claim 6.4.7 and the fact that $h > h'$ imply that $b'.t_{max}[h] = 1$. This contradicts the assumption that $b'.t_{max}[h] \neq 1$. Therefore, this case cannot arise.

■

Claim 6.4.13 *If $k \neq b.i_{max}$ and $b.t_{max} \prec b.nt_k$ then VIa is true in b'.*

Proof: Let $b'.nt_i = \text{NEXTLABEL}(b'.t_{max}, h)$ for some h and i. We proceed with a case analysis:

$h < h'$: Part 1 of Claim 6.4.7 and the definition of NEXTLABEL show that $b'.t_{max} \stackrel{h}{=} b.t_{max}$. Now the fact that nt_i does not change and the fact that $b'.nt_i = \text{NEXTLABEL}(b'.t_{max}, h)$ imply that $b.nt_i = \text{NEXTLABEL}(b.t_{max}, h)$. Since $b.nt_i = \text{NEXTLABEL}(b.t_{max}, h)$ and VIa is true in state b, $\text{NUM}_i(b.t_{max}, h-1) \geq n-h$. Part 5 of Claim 6.4.7 shows that $\text{NUM}_i(b'.t_{max}, h-1) \geq \text{NUM}_i(b.t_{max}, h-1)$. Therefore, $\text{NUM}_i(b'.t_{max}, h-1) \geq n-h$ which implies that VIa is true for b'.

$h = h'$[7]: Using part 1 of Claim 6.4.7 and the definition of NEXTLABEL we can conclude that $b'.t_{max}[h] = \text{NEXT}(b.t_{max}[h])$. There exists no $z \in \mathcal{A}$ such that $\text{NEXT}(z) = 1$. Hence $b'.t_{max}[h] \neq 1$. Claim 6.4.13 implies that VIb holds for state b'. Since $b'.t_{max}[h] \neq 1$, VIb for state b' implies that $\text{NUM}(b'.t_{max}, h-1) \geq n-h+1$. Thus $\text{NUM}_i(b'.t_{max}, h-1) \geq n-h$ and VIa is true in state b'.

$h > h'$: The fact that $b'.nt_i = \text{NEXTLABEL}(b'.t_{max}, h)$ and the definition of NEXTLABEL imply that $b'.nt_i \stackrel{h-1}{=} b'.t_{max}$. Now part 3 of Claim 6.4.7 and the fact that $h > h'$ imply that $b'.nt_i = b'.t_{max}$. Thus $b'.nt_i \neq \text{NEXTLABEL}(b'.t_{max}, h)$ which contradicts our assumption that $b'.nt_i = \text{NEXTLABEL}(b'.t_{max}, h)$. Therefore, this case cannot arise.

■

We now complete the proof of the lemma. To show that b' satisfies I - VI we consider two cases: $k = b.i_{max}$ and $k \neq b.i_{max}$. Claim 6.4.1 shows that b' satisfies I - VI when $k = b.i_{max}$.

[7] Actually, this case cannot arise. However, the argument that proves that the case cannot arise is more complicated that the argument that proves that VIa is satisfied if the case does arise.

When $k \neq b.i_{max}$ Claim 6.4.2 shows that invariant I holds in state b'. The proof for invariants II - VI is subdivided into two cases: $b.nt_k \preceq b.t_{max}$ and $b.t_{max} \prec b.nt_k$. Claim 6.4.5 shows that II - VI hold when $b.nt_k \preceq b.t_{max}$. Claim 6.4.8, Claim 6.4.9, Claim 6.4.10, Claim 6.4.11, Claim 6.4.12 and Claim 6.4.13 each consider one of the invariants to show that II - VI hold when $b.t_{max} \prec b.nt_k$. ■

Lemma 6.5 *Let b be a state of* BCTSS *that satisfies I - VI. If $(b, \text{SNAP}_k(\bar{t_k}, \bar{v_k}), b')$ is a step of* BCTSS *for any k, then b' satisfies I - VI.*

Proof: Note that none of the t-labels or nt-labels change when $op_k = \text{SCAN}_k$. Therefore, assume that $op_k = \text{LABEL}_k$. The proof is divided into s series of claims. First consider the case where $k = b.i_{max}$.

Claim 6.5.14 *If $k = b.i_{max}$ then b' satisfies I - VI.*

Proof: The definition of $\text{SNAP}_k(\bar{t_k}, \bar{v_k})$ for BCTSS shows that no labels change. This suffices to show that b' satisfies I - VI. ■

So assume that $k \neq b.i_{max}$ for the remainder of the proof of the lemma. By definition of NEWLABEL_k, $b'.nt_k = \text{NEXTLABEL}(b.t_{max}, h')$ for some $h' \in \{1 \ldots n-1\}$. Fix h'. Note, by definition of NEXTLABEL, $b.t_{max} \prec b'.nt_k$.

Claim 6.5.15 *If $k \neq b.i_{max}$ then $\text{NUM}_k(b.t_{max}, h') = \text{NUM}_k(b.t_{max}, h'-1) = n - h'$.*

Proof: By definition of NEWLABEL_k, $\text{FULL}_k(h')$ returns *true* in state b, so $\text{NUM}_k(b.t_{max}, h') \geq n - h'$. Moreover, $\text{FULL}_k(h'-1)$ returns *false* in state b, therefore $\text{NUM}_k(b.t_{max}, h'-1) < n-(h'-1)$. But by definition, $\text{NUM}_k(b.t_{max}, h'-1) \geq \text{NUM}_k(b.t_{max}, h')$ so $\text{NUM}_k(b.t_{max}, h'-1) = \text{NUM}_k(b.t_{max}, h') = n - h'$. ■

Claim 6.5.16 *If $k \neq b.i_{max}$ then I is true in b'.*

Proof: For a contradiction assume that $\text{TOT}(b') = \mathit{false}$. Then there must exist a choice vector C whose values are not totally ordered. By Lemma 5.2, there exists $b'.\ell_i, b'.\ell_j, b'.\ell_z \in C$ such that $b'.\ell_i \overset{h-1}{=} b'.\ell_j \overset{h-1}{=} b'.\ell_z$ and $\{b'.\ell_i[h], b'.\ell_j[h], b'.\ell_z[h]\} = \{3,4,5\}$ for some $h \in \{1 \ldots n-1\}$.

Since $b'.\ell_i, b'.\ell_j$ and $b'.\ell_z$ are elements of a choice vector, $b'.\ell_i \in \{b'.t_i,\ b'.nt_i\}$, $b'.\ell_j \in \{b'.t_j,\ b'.nt_j\}$, $b'.\ell_z \in \{b'.t_z,\ b'.nt_z\}$ and $i \neq z, j \neq z, j \neq i$. By I for state b, $\text{TOT}(b) = \textit{true}$. Therefore the values of C for state b must be totally ordered. The only label that changes as a result of the action is nt_k. Consequently, we can assume without loss of generality that $b'.\ell_z = b'.nt_k$ and $z = k$. Furthermore, since $i \neq k$ and $j \neq k$, ℓ_i and ℓ_j do not change as a result of the action. Thus, $b.\ell_i = b'.\ell_i$ and $b.\ell_j = b'.\ell_j$. Now we can conclude that:

$$b.\ell_i \overset{h-1}{=} b.\ell_j \overset{h-1}{=} b'.nt_k \quad \text{and} \quad \{b.\ell_i[h], b.\ell_j[h], b'.nt_k[h]\} = \{3,4,5\}. \tag{1}$$

Recall that $b'.nt_k = \text{NEXTLABEL}(b.t_{max}, h')$. We will now show that $h = h'$. Let $z = b.i_{max}$, then $b.t_z = b.t_{max}$. Since $k \neq b.i_{max}$, $k \neq z$. The definition of NEXTLABEL implies that $b.t_z \overset{h'-1}{=} b'.nt_k$. For a contradiction assume that $h < h'$. Now substitute $b.t_z$ for $b'.nt_k$ in Equation 1 to conclude that $b.\ell_i \overset{h-1}{=} b.\ell_j \overset{h-1}{=} b.t_z$ and $\{b.\ell_i[h], b.\ell_j[h], b.t_z[h]\} = \{3,4,5\}$. By Lemma 5.2 any set of labels containing $b.\ell_i, b.\ell_j$, and $b.t_z$ is not totally ordered. We now show that $i \neq z$ and $j \neq z$ since this will allow us to conclude that there exists a choice vector that includes $b.\ell_i, b.\ell_j$, and $b.t_z$. Since $\{b.\ell_i[h], b.\ell_j[h], b.t_z[h]\} = \{3,4,5\}$, and $b.\ell_i \in \{b.t_i, b.nt_i\}$ either $b.t_i[h] \neq b.t_z[h]$ or $b.nt_i[h] \neq b.t_z[h]$. If $i = z$ the former is clearly impossible and the later is impossible since $b.nt_z = b.t_z$ by invariant II. Thus $i \neq z$. The same argument shows that $j \neq z$. Now we have a choice vector for state b whose values are not totally ordered. The existence of such a choice vector contradicts invariant I for state b. Thus $h \not< h'$. The definition of NEXTLABEL implies that $b'.nt_k[h''] = 1$ for all $h'' > h'$. Since $b'.nt_k[h] \in \{3,4,5\}$, $h \not> h'$. Now $h \not< h'$ and $h \not> h'$ so $h = h'$.

We now construct a set of labels which is not totally ordered and which includes $b.t_{max}$ and $b'.nt_k$. First show that $b.t_{max}[h'] \in \{3,4,5\}$. Since $b'.nt_k[h'] \in \{3,4,5\}$, the definition of NEXTLABEL implies that $b.t_{max}[h'] \in \{2,3,4,5\}$. We proceed by showing that $b.t_{max}[h'] \neq 2$. In order to reach a contradiction we assume that $b.t_{max}[h'] = 2$. Since $b.t_{max} \overset{h'-1}{=} b'.nt_k$ and $b'.nt_k \overset{h'-1}{=} b.\ell_i$, $b.t_{max} \overset{h'-1}{=} b.\ell_i$. Furthermore, $b.t_{max}[h'] = 2$ and $b.\ell_i[h'] \in \{3,4,5\}$ thus $b.t_{max}[h'] \prec_{\mathcal{A}} b.\ell_i[h']$. Consequently, $b.t_{max} \prec b.\ell_i$. We consider the cases $b.\ell_i = b.t_i$ and $b.\ell_i = b.nt_i$ separately. When $b.\ell_i = b.t_i$, $b.t_{max} \prec b.t_i$, which contradicts the definition of $b.t_{max}$. Thus, this case cannot arise. When $b.\ell_i = b.nt_i$, $b.t_{max} \prec b.nt_i$. Now invariant III and the definition of NEXTLABEL imply that $b.nt_i[h'] = b.t_{max}[h']$ or $b.nt_i[h'] = \text{NEXT}(b.t_{max}[h'])$ or $b.nt_i[h'] = 1$. Thus, when $b.t_{max}[h'] = 2$, $b.nt_i[h'] \notin \{4,5\}$. Therefore we can conclude that $b.\ell_i[h'] \notin$

$\{4,5\}$ when $b.t_{max}[h'] = 2$. Using the same argument we can show that $b.\ell_j[h'] \notin \{4,5\}$ when $b.t_{max}[h'] = 2$. This contradicts Equation 1 according to which $\{b.\ell_i[h'], b.\ell_j[h'], b'.nt_k[h']\} = \{3,4,5\}$. Thus $b.t_{max}[h'] \neq 2$ and $b.t_{max}[h'] \in \{3,4,5\}$.

Since $\{b.\ell_i[h'], b.\ell_j[h'], b'.nt_k[h']\} = \{3,4,5\}$, using the definition of $\prec_{\mathcal{A}}$, we can assume without loss of generality that:

$$b.\ell_i[h'] \prec_{\mathcal{A}} b.\ell_j[h'] \prec_{\mathcal{A}} b'.nt_k[h'] \quad \text{and} \quad b.\ell_i[h'] \not\prec_{\mathcal{A}} b'.nt_k[h']. \tag{2}$$

Recall that $z = b.i_{max}$, $b.t_z = b.t_{max}$, $b.t_z \overset{h'-1}{=} b'.nt_k$, and $b.t_z[h'] \prec_{\mathcal{A}} b'.nt_k[h']$. Hence, we can replace $b.\ell_j$ by $b.t_{max}$ in Equation 1 and Equation 2 which yields the following:

$$b.\ell_i \overset{h'}{=} b.t_{max} \overset{h'}{=} b'.nt_k \quad \text{and} \quad \{b.\ell_i[h], b.t_{max}[h], b'.nt_k[h]\} = \{3,4,5\}, \tag{3}$$

$$b.\ell_i[h'] \prec_{\mathcal{A}} b.t_{max}[h'] \prec_{\mathcal{A}} b'.nt_k[h'] \quad \text{and} \quad b.\ell_i[h'] \not\prec_{\mathcal{A}} b'.nt_k[h']. \tag{4}$$

Consequently,

$$b.\ell_i \prec b.t_{max} \prec b'.nt_k \quad \text{and} \quad b.\ell_i \not\prec b'.nt_k, \tag{5}$$

$$\{b.\ell_i, b.t_{max}, b'.nt_k\} \subseteq \text{CYCLE}(b.t_{max}, h'). \tag{6}$$

Consider the cases $b.\ell_i = b.nt_i$ and $b.\ell_i = b.t_i$ separately:

$b.nt_i$: Since $b.nt_i \in \text{CYCLE}(b.t_{max}, h')$, V for state b shows that $b.t_i \overset{h'-1}{=} b.t_{max}$. By Claim 6.5.15 $\text{NUM}_k(b.t_{max}, h'-1) = \text{NUM}_k(b.t_{max}, h')$. Therefore, since $i \neq k$, $b.t_i \overset{h'-1}{=} b.t_{max}$ implies that $b.t_i \overset{h'}{=} b.t_{max}$. Now, from IV for state b and the fact that $b.nt_i \prec b.t_{max}$, it follows that $b.nt_i \overset{h'}{=} b.t_{max}$, a contradiction to Equation 4 according to which $b.nt_i[h'] \prec_{\mathcal{A}} b.t_{max}[h']$.

$b.t_i$: By Claim 6.5.15, $\text{NUM}_k(b.t_{max}, h'-1) = \text{NUM}_k(b.t_{max}, h')$. Therefore, since $i \neq k$, $b.t_i \overset{h'-1}{=} b.t_{max}$ implies that $b.t_i \overset{h'}{=} b.t_{max}$. Now, $b.t_i \overset{h'}{=} b.t_{max}$ contradicts Equation 4 according to which $b.t_i[h'] \prec_{\mathcal{A}} b.t_{max}[h']$.

We have reached a contradiction in each case. Consequently, there exists no choice vector such that its values are not totally ordered. Hence, $\text{TOT}(b') = \mathit{true}$. ■

Claim 6.5.17 *If $k \neq b.i_{max}$ then II - VI are true in b'.*

Proof: VIb holds in state b' since it holds in state b and no t-labels change. Now consider II - VIa. If $i \neq k$, then the definition of $\text{SNAP}_k(\bar{t_k}, \bar{v_k})$ shows that neither t_i, nt_i, t_{max}, nor $\text{NUM}_i(t_{max}, h)$ change. Therefore, II - VIa are true in state b' since II - VIa are true in state b. So assume that $i = k$. In this case $b'.nt_i = \text{NEXTLABEL}(b.t_{max}, h')$ and $b'.t_{max} \prec b'.nt_i$. Consider II - VIa separately:

II: Since $k \neq b.i_{max}$, $i \neq b.i_{max}$. Furthermore, $b.i_{max} = b'.i_{max}$ thus $i \neq b'.i_{max}$. Now II is vacuously true in state b'.

III: Since $b'.t_{max} = b.t_{max}$, and $b'.nt_i = \text{NEXTLABEL}(b.t_{max}, h')$, $b'.nt_i = \text{NEXTLABEL}(b'.t_{max}, h')$.

IV: Since $b'.t_{max} = b.t_{max} \prec b'.nt_i$ IV is vacuously true in b'.

V: Suppose that $b'.nt_i \in \text{CYCLE}(b'.t_{max}, h)$ where $h \in \{1 \ldots n-1\}$. The definition of CYCLE now implies that $b'.nt_i[h] \in \{3, 4, 5\}$. Recall that $b'.nt_i = \text{NEXTLABEL}(b.t_{max}, h')$. The definition of NEXTLABEL implies that $b'.nt_i[h''] = 1$ for all $h'' > h'$. Since $b'.nt_i[h] \in \{3, 4, 5\}$, we can conclude that $h \leq h'$. We consider the two cases $h = h'$ and $h < h'$ separately.

First consider the case $h = h'$. Since $\text{NEXT}(1) \notin \{3, 4, 5\}$, and $\text{NEXT}(b.t_{max}[h]) = b'.nt_i[h] \in \{3, 4, 5\}$, $b.t_{max}[h] \neq 1$. Now VIb for state b shows that $\text{NUM}(b.t_{max}, h-1) \geq n-h+1$. Furthermore, Claim 6.5.15 and the fact that $i = k$ show that $\text{NUM}_i(b.t_{max}, h-1) < n-h+1$. Since $\text{NUM}(b.t_{max}, h-1) \geq n-h+1$ and $\text{NUM}_i(b.t_{max}, h-1) < n-h+1$, $k \in \text{AGREE}(b.t_{max}, h-1)$. Thus $b.t_i \overset{h-1}{=} b.t_{max}$. Since t_i and t_{max} do not change, $b'.t_i \overset{h-1}{=} b'.t_{max}$.

Now consider the case $h < h'$. The fact that $b'.nt_i = \text{NEXTLABEL}(b.t_{max}, h')$ and the definition of NEXTLABEL imply that $b.t_{max}[h] = b'.nt_i[h]$. Therefore, $b.t_{max}[h] \neq 1$ since $b.t_{max}[h] = b'.nt_i[h] \in \{3, 4, 5\}$. Now VIb for state b shows that $\text{NUM}(b.t_{max}, h-1) \geq n-h+1$. The definition of NEWLABEL_i and the fact that $i = k$ show that $\text{FULL}_i(h-1)$ returns *false*, which implies that $\text{NUM}_i(b.t_{max}, h-1) < n-h+1$. Since $\text{NUM}(b.t_{max}, h-1) \geq n-h+1$ and $\text{NUM}_i(b.t_{max}, h-1) < n-h+1$, $i \in \text{AGREE}(b.t_{max}, h-1)$. Thus $b.t_i \overset{h-1}{=} b.t_{max}$. Since t_i and t_{max} do not change, $b'.t_i \overset{h-1}{=} b'.t_{max}$.

VIa: Since $b'.t_{max} = b.t_{max}$ and $b'.nt_i = \text{NEXTLABEL}(b.t_{max}, h')$, we conclude that $b'.nt_i = \text{NEXTLABEL}(b'.t_{max}, h')$. Now, Claim 6.5.15 implies that $\text{NUM}_i(b'.t_{max}, h' - 1) = n - h'$.

■

We can now complete the proof of the lemma. Claim 6.5.14 shows that I - VI hold for b' when $k = b.i_{max}$. When $k \neq b.i_{max}$, Claim 6.5.16 shows that I holds in b' and Claim 6.5.17 shows that II - VI hold for b'. ■

Proof: (For Theorem 6.1) We proceed by induction on the length of the execution ending in the reachable state b. The base case is established by Lemma 6.2. The induction step is a case analysis based on the action π, where (b', π, b'') is a step in the execution. If $\pi \in \{\text{BEGINSCAN}_k, \text{ENDSCAN}_k(\bar{o_k}, \bar{v_k}), \text{BEGINLABEL}_k(val_k), \text{ENDLABEL}_k\}$, the induction step follows from Lemma 6.3. If $\pi = \text{UPDATE}_k((t_k, v_k), (nt_k, val_k))$, the induction step follows from Lemma 6.4. If $\pi = \text{SNAP}_k(\bar{t_k}, \bar{v_k})$, the induction step follows from Lemma 6.5. ■

7 Simulation Proof

In this section we prove that BCTSS solves CTSS. Specifically, we use Theorem 2.1 to show that *fairbehs*(BCTSS) $\subseteq$ *fairbehs*(UCTSS). This implies that BCTSS implements UCTSS. Recall that we have already shown that UCTSS solves CTSS. In order to use Theorem 2.1, we define the relation R between the states of BCTSS and the states of UCTSS as follows:

Definition 7.1 (relation R) If b is a state of BCTSS and u is a state of UCTSS then $(b, u) \in$ R iff for all $i, j \in \{1 \ldots n\}$, $i \neq j$:

1. $b.\bar{o_i} = u.\bar{o_i}$.
2. $b.t_j \prec b.t_i$ iff $u.t_j < u.t_i$,
   $b.nt_j \prec b.t_i$ iff $u.nt_j < u.t_i$,
   $b.t_j \prec b.nt_i$ iff $u.t_j < u.nt_i$,
   $b.nt_j \prec b.nt_i$ iff $u.nt_j < u.nt_i$.
3. $b.v_i = u.v_i$.

4. $b.val_i = u.val_i$.

5. $b.\bar{v}_i = u.\bar{v}_i$.

6. $b.op_i = u.op_i$.

7. $b.pc_i = u.pc_i$.

■

Parts 1 and 5 ensure that a process p_i returns the same response to a SCAN$_i$ request in BCTSS and in UCTSS. Recall that $\bar{o}_i$ contains the order of the labels that was last observed by p_i. Part 2 states that the $\prec$ ordering of any choice vector from BCTSS is the same as the $<$ ordering of the corresponding labels from UCTSS. Notice that part 2 gives no information about the relation between t_i and nt_i. Parts 3 and 5 ensure that BCTSS and UCTSS associate values with labels in the same manner. Part 6 ensures that UCTSS and BCTSS will execute the same part of the SNAP$_i$ action code. Finally, part 7 ensures that UCTSS and BCTSS will be able to execute the corresponding action during each state transition.

The following lemma proves that the first of the three assumptions required by Theorem 2.1 is true.

Lemma 7.1 *For the initial state b of* BCTSS*, there exists an initial state u of* UCTSS *such that* $(b, u) \in$ R.

Proof: In the initial states b of BCTSS and u of UCTSS, $\bar{o}_i = (1 \ldots n)$ for all $i \in \{1 \ldots n\}$. Hence part 1 of R is satisfied. Part 2 is satisfied since $t_i = nt_j$ for all $i, j \in \{1 \ldots n\}$ in both BCTSS and UCTSS. Parts $3 - 5$ are satisfied since $\bar{v}_i = (0 \ldots 0)$ and $v_i = val_i = 0$ for all $i \in \{1 \ldots n\}$ in both BCTSS and UCTSS. Parts 6 and 7 of R is satisfied for the initial states since $op_i = pc_i =$ NIL in both systems. ■

The following lemma shows that the mapping R is preserved by all of the actions of BCTSS. This lemma proves that the second of the three assumptions required by Theorem 2.1 is true.

Lemma 7.2 *Let b be a reachable state of* BCTSS *and u be a reachable state of* UCTSS *such that* $(b, u) \in$ R*. If (b, π, b') is a step of* BCTSS *then, there exists u' such that (u, π, u') is a step of* UCTSS *and* $(b', u') \in$ R.

Proof: We proceed by case analysis on π.

Case $\pi \in \{\text{BEGINSCAN}_k, \text{ENDSCAN}_k(\bar{o_k}, \bar{v_k}), \text{ENDLABEL}_k\}$:

Since $(b,u) \in \text{R}$, we can conclude that $b.pc_k = u.pc_k$, $b.\bar{o_k} = u.\bar{o_k}$, and $b.\bar{v_k} = u.\bar{v_k}$. Hence, π is enabled in u. Let u' be the unique state of UCTSS such that (u, π, u') is a step of UCTSS. In both BCTSS and UCTSS only op_k and pc_k change as a result of π. Inspection of the code in Figure 1 shows that $b'.op_k = u'.op_k$ and $b'.pc_k = u'.pc_k$. This suffices to shows that $(b', u') \in \text{R}$.

Case: $\pi = \text{BEGINLABEL}_k(val_k)$:

Since $\text{BEGINLABEL}_k(val_k)$ is an input action, it is clearly enabled in state u. Let u' be the unique state of UCTSS such that (u, π, u') is a step of UCTSS. Only val_k, op_k, and pc_k change as a result of the action. By definition of the action $b'.val_k = u'.val_k$. Furthermore $b'.op_k = u'.op_k = \text{LABEL}_k$ and $b'.pc_k = u'.pc_k = \text{SNAP}_k(\bar{t_k}, \bar{v_k})$. This suffices to shows that $(b', u') \in \text{R}$.

Case $\pi = \text{SNAP}_k(\bar{t_k}, \bar{v_k})$ when $b.op_k = \text{SCAN}_k$:

Since $(b,u) \in \text{R}$, $b.pc_k = u.pc_k$. Hence, π is enabled in u. Furthermore $u.op_k = b.op_k = \text{SCAN}_k$. Let u' be the unique state such that (u, π, u') is a step of UCTSS.

$\text{SNAP}_k(\bar{t_k}, \bar{v_k})$, when $op_k = \text{SCAN}_k$, determines $\bar{o_k}$ based on the $\ll$ ordering. Recall that $\ll$ is a lexicographical order defined by the order between the t-labels, using $\prec$ for BCTSS and $<$ for UCTSS, and the order between the process indices. By assumption $(b,u) \in \text{R}$. This implies that $b.t_i \prec b.t_j$ iff $u.t_i < u.t_j$ for all $i, j \in \{1 \ldots n\}$; thus $\text{SNAP}_k(\bar{t_k}, \bar{v_k})$ will produce the same ordering for BCTSS and UCTSS. Hence $b'.\bar{o_k} = u'.\bar{o_k}$. Furthermore, part 3 of R implies that $b'.\bar{v_k} = u'.\bar{v_k}$. Figure 1 shows $b'.pc_k = u'.pc_k = \text{ENDSCAN}_k(\bar{o_k}, \bar{v_k})$. Only $\bar{o_k}$, $\bar{v_k}$, and pc_k change as a result of the action and thus we can conclude that $(b', u') \in \text{R}$.

Case $\pi = \text{SNAP}_k(\bar{t_k}, \bar{v_k})$ when $b.op_k = \text{LABEL}_k$:

Since $(b,u) \in \text{R}$, $b.pc_k = u.pc_k$. Hence, π is enabled in u. Furthermore $u.op_k = b.op_k = \text{LABEL}_k$. There are two case: $k = b.i_{max}$ and $k \neq b.i_{max}$.

We first consider the case $k = b.i_{max}$. Since $(b,u) \in \text{R}$, part 2 of R implies that $b.i_{max} = u.i_{max}$. Hence, $k = u.i_{max}$. Let u' be the unique state such that (u, π, u') is a step of UCTSS. Now the definition of NEWLABEL_k for BCTSS and UCTSS shows that only pc_k changes for both

BCTSS and UCTSS. Figure 1 shows $b'.pc_k = u'.pc_k = \text{UPDATE}_k((t_k, v_k), (nt_k, val_k))$. This suffices to show that $(b', u') \in \text{R}$.

So assume that $k \neq b.i_{max}$ for the remainder of the proof of this case. Since $(b, u) \in \text{R}$, part 2 of R implies that $b.i_{max} = u.i_{max}$. Hence, $k \neq u.i_{max}$. In this case there are many states u' such that (u, π, u') is a step of UCTSS; these states differ only by the value of $u'.nt_k$. We now define a particular value $u'.nt_k$ and hence a particular state u'.

Define $S = \{i | \, i \neq k \text{ and } b.t_{max} \prec b.nt_i\}$. Let $z = b.i_{max}$, then $b.t_z = b.t_{max}$. Invariant II shows that $b.nt_z = b.t_z$. Hence, $b.nt_z = b.t_{max}$. This implies that $z \notin S$. Thus, $b.i_{max} \notin S$. For all $i \in S$, III for state b shows that $b.nt_i = \text{NEXTLABEL}(b.t_{max}, h_i)$ for some $h_i \in \{1 \ldots n-1\}$. Furthermore, the definition of NEWLABEL_k implies that $b'.nt_k = \text{NEXTLABEL}(b.t_{max}, h_k)$ for some $h_k \in \{1 \ldots n-1\}$. Define:

$$S_1 = \{i | \, i \in S, h_i > h_k\}, \quad S_2 = \{i | \, i \in S, h_i = h_k\} \quad \text{and} \quad S_3 = \{i | \, i \in S, h_i < h_k\}. \tag{7}$$

Note that:

$$S_1 \cap S_2 = S_2 \cap S_3 = S_1 \cap S_3 = \emptyset \quad \text{and} \quad S_1 \cup S_2 \cup S_3 = S. \tag{8}$$

Since $\prec$ is a lexicographical order, the order between any two labels in BCTSS is determined by the first digit at which they differ. Therefore, for any $i_1 \in S_1$, $i_2 \in S_2$, and $i_3 \in S_3$, it is the case that:

$$b.t_{max} \prec b.nt_{i_1} \prec b.nt_{i_2} = b'.nt_k \prec b.nt_{i_3}. \tag{9}$$

Recall $z = b.i_{max}$. Thus, $b.t_z \prec b.nt_{i_1} \prec b.nt_{i_2} = b'.nt_k \prec b.nt_{i_3}$. Since $z \notin S$ and $(b, u) \in \text{R}$, part 2 of R now shows that $u.t_z < u.nt_{i_1} < u.nt_{i_2} < u.nt_{i_3}$. Since $b.i_{max} = u.i_{max}$, $z = u.i_{max}$ and $u.t_z = u.t_{max}$. This shows that:

$$u.t_{max} < u.nt_{i_1} < u.nt_{i_2} < u.nt_{i_3}. \tag{10}$$

We use the following rules for picking $u'.nt_k$. If $S_2 \neq \emptyset$, then $u'.nt_k = u.nt_i$ for any $i \in S_2$. If on the other hand $S_2 = \emptyset$, define $u.nt_{max}$ and $u.nt_{min}$ as follows: $u.nt_{max} = \max(u.nt_i | \, i \in S_1)$ if $S_1 \neq \emptyset$, otherwise $u.nt_{max} = u.t_{max}$. $u.nt_{min} = \min(u.nt_i | \, i \in S_3)$ if $S_3 \neq \emptyset$, otherwise $u.nt_{min} = \infty$. Choose any $u'.nt_k$ such that $u.nt_{max} < u'.nt_k < u.nt_{min}$. For any $i_1 \in S_1, i_2 \in S_2$, and $i_3 \in S_3$, the two rules and Equation 10 imply that:

$$u.t_{max} < u.nt_{i_1} < u.nt_{i_2} = u'.nt_k < u.nt_{i_3}. \tag{11}$$

With both rules for choosing $u'.nt_k$, $u.t_{max} < u'.nt_k$. Hence, there exists an $X \in \Re^{>0}$ such that $u'.nt_k = u.t_{max} + X$.

We now show that $(b', u') \in$ R. Only nt_k and pc_k change as a result of the action. Figure 1 shows $b'.pc_k = u'.pc_k = \text{UPDATE}_k((t_k, v_k), (nt_k, val_k))$. Consequently, $(b', u') \in$ R if we can show that part 2 of R holds for states b' and u'. For part 2 of the relation there are four cases to consider. All other cases do not involve $b'.nt_k$. Let $i \in \{1 \ldots n\}$ and $i \neq k$:

1. $b'.nt_k \prec b'.t_i$ iff $u'.nt_k < u'.t_i$,

   $b'.t_i \prec b'.nt_k$ iff $u'.t_i < u'.nt_k$:

   Since no t-labels change, $b'.t_{max} = b.t_{max}$ and $b'.i_{max} = b.i_{max}$. Recall that $k \neq b.i_{max}$, hence $b'.nt_k = \text{NEXTLABEL}(b.t_{max}, h_k)$ and $b'.t_{max} = b.t_{max} \prec b'.nt_k$ as a result of the action. Furthermore, $b'.t_i = b.t_i$. Therefore, $b'.t_i \preceq b'.t_{max} \prec b'.nt_k$. Let $z = b'.i_{max}$. In this case $z \neq k$ and $b'.t_z = b'.t_{max}$. Since $i \neq k$, $z \neq k$ and $b'.t_z = b'.t_{max}$, there exists a choice vector that includes $b'.t_i, b'.t_{max}$, and $b'.nt_k$. By invariant I the values of this choice vector are totally ordered by $\prec$. Therefore, $b'.t_i \preceq b'.t_{max} \prec b'.nt_k$ implies that $b'.t_i \prec b'.nt_k$.

   Similarly, since $k \neq u.i_{max}$, $u'.t_{max} = u.t_{max} < u'.nt_k$ as a result of the action. Furthermore, $u'.t_i = u.t_i$. Therefore $u'.t_i \leq u'.t_{max} < u'.nt_k$. This implies that $u'.t_i < u'.nt_k$.

2. $b'.nt_i \prec b'.nt_k$ iff $u'.nt_i < u'.nt_k$,

   $b'.nt_k \prec b'.nt_i$ iff $u'.nt_k < u'.nt_i$:

   We can divide the nt-labels of UCTSS into two disjoint sets: Recall that $S = \{j \mid j \neq k$ and $b.t_{max} \prec b.nt_j\}$. Define $T = \{j \mid j \neq k$ and $b.t_{max} \succeq b.nt_j\}$. Similarly, define $S_u = \{j \mid j \neq k$ and $u.t_{max} < u.nt_j\}$. Define $T_u = \{j \mid j \neq k$ and $u.t_{max} \geq u.nt_j\}$. By part 2 of R and the fact that $(b, u) \in$ R, $S = S_u$ and $T = T_u$. Consider $i \in T$ and $i \in S$ separately.

   Suppose $i \in T$. Since $i \neq k$, $b'.nt_i = b.nt_i$. Therefore $b'.nt_i \preceq b'.t_{max} \prec b'.nt_k$. Let $z = b'.i_{max}$. In this case $z \neq k$ and $b'.t_z = b'.t_{max}$. Since $i \neq k$, $z \neq k$ and $b'.t_z = b'.t_{max}$, there exists a choice vector that includes $b'.nt_i, b'.t_{max}$, and $b'.nt_k$. By invariant I the values of this choice vector are totally ordered by $\prec$. Therefore, $b'.nt_i \preceq b'.t_{max} \prec b'.nt_k$ implies that $b'.nt_i \prec b'.nt_k$. Similarly, $u'.nt_i = u.nt_i$, since $i \neq k$. Therefore, $u'.nt_i \leq u'.t_{max} < u'.nt_k$. This implies that $u'.nt_i < u'.nt_k$.

Now suppose $i \in S$. Consider any $i_1 \in S_1$, $i_2 \in S_2$, and $i_3 \in S_3$ where S_1, S_2, S_3 are defined by Equation 7. Since $k \notin S$, $b'.nt_j = b.nt_j$ and $u'.nt_j = u.nt_j$ for all $j \in S$. Consequently Equation 9 and Equation 11 show that $b.t_{max} \prec b'.nt_{i_1} \prec b'.nt_{i_2} = b'.nt_k \prec b'.nt_{i_3}$ and $u.t_{max} < u'.nt_{i_1} < u'.nt_{i_2} = u'.nt_k < u'.nt_{i_3}$. Using these facts we now consider the following cases: $i \in S_1$, $i \in S_2$, and $i \in S_3$. If $i \in S_1$, then $b'.nt_i \prec b'.nt_k$ and $u'.nt_i < u'.nt_k$. If $i \in S_2$, then $b'.nt_i = b'.nt_k$ and $u'.nt_i = u'.nt_k$. If $i \in S_3$, then $b'.nt_k \prec b'.nt_i$ and $u'.nt_k < u'.nt_i$.

Case $\pi = \text{UPDATE}_k((t_k, v_k), (nt_k, val_k))$:

Since $(b, u) \in \text{R}$, $b.pc_k = u.pc_k$. Hence, π is enabled in u. Let u' be the unique state such that (u, π, u') is a step of UCTSS.

Only v_k, t_k and pc_k change as a result of the action. Since $(b, u) \in \text{R}$, part 4 of R shows that $b.val_k = u.val_k$. Thus, $b'.v_k = u'.v_k$. Figure 1 shows $b'.pc_k = u'.pc_k = \text{ENDLABEL}_k$. Consequently, $(b', u') \in \text{R}$ if we can show that part 2 of R holds for states b' and u'. For part 2 of R there are four cases to consider. All other cases are immediate since they do not involve t_k, and since t_k is the only label that changes as a result of the action. Let $i \in \{1 \ldots n\}$ and $i \neq k$:

1. $b'.t_k \prec b'.t_i$ iff $u'.t_k < u'.t_i$:

   Since $(b, u) \in \text{R}$ and t_k is the only label that changes, $b.nt_k \prec b'.t_i$ iff $u.nt_k < u'.t_i$. As a result of the action, $b'.t_k = b.nt_k$ and $u'.t_k = u.nt_k$. Hence $b'.t_k \prec b'.t_i$ iff $u'.t_k < u'.t_i$.

2. $b'.t_i \prec b'.t_k$ iff $u'.t_i < u'.t_k$,
   $b'.nt_i \prec b'.t_k$ iff $u'.nt_i < u'.t_k$,
   $b'.t_k \prec b'.nt_i$ iff $u'.t_k < u'.nt_i$:

   For all three statements, the reasoning is similar to that of case 1.

■

We can now conclude that BCTSS correctly implements the properties of CTSS.

Theorem 7.3 BCTSS *solves* CTSS.

Proof: By definition of BCTSS and UCTSS, *sig*(BCTSS) = *sig*(UCTSS) and *part*(BCTSS) = *part*(UCTSS). Lemma 7.1, and Lemma 7.2 show that BCTSS and UCTSS satisfy the first two conditions of Theorem 2.1. For the third condition note that action π is enabled in UCTSS if and only if π is enabled in BCTSS. Consequently, Theorem 2.1 shows that *fairbehs*(BCTSS) $\subseteq$ *fairbehs*(UCTSS). Thus BCTSS implements UCTSS. Since UCTSS solves CTSS, BCTSS solves CTSS. ■

8 Applications

This section discusses two applications of a CTSS in the area of waitfree algorithms. Specifically, we discuss multireader multiwriter atomic registers and *first-come-first-serve* (*fcfs*) mutual exclusion[8]. Both of these problems are solved by very simple algorithms based on a CTSS. Using our bounded CTSS, these problems have a simple bounded solution. For both problems we present an algorithm based on a CTSS along with a correctness proof for the algorithm. In the correctness proof, we assume nothing about the CTSS except that it satisfies the CTSS specification of Section 3.

l–exclusion (see [13, 14]) and randomized consensus (see [4, 8, 27, 2]) are also important problems that have simple CTSS based solutions. l–exclusion seeks to limit the number of processes concurrently executing a section of code called the *critical section* to l. Mutual exclusion is the same as l–exclusion when $l = 1$. Randomized consensus provides a random algorithm by which a set of asynchronous processes can agree on a common value. A consensus algorithm is consider *valid* if all processes agree on value a whenever a was the input originally given to all processes. Finally a consensus algorithm must guarantee that each process will terminate in a finite number of steps with probability 1 even if other processes exhibit stopping failures. Shavit [37] presents an algorithm based on a CTSS along with a correctness proof for both the l–exclusion and randomized consensus problems. In the correctness proofs, he assumes nothing about the CTSS except that it satisfies axioms **P0**-**P3** of the CTSS specification of Section 3.

[8]The algorithms for *fcfs* mutual exclusion and multireader multiwriter registers presented in this paper are based on similar algorithms presented in [37]. We discuss the algorithms since [37] does not prove their correctness.

8.1 Multireader Multiwriter Atomic Registers

This section presents a simple bounded algorithm for solving the famous problem of constructing a multireader multiwriter atomic register, MRMW, from single writer multireader atomic registers (see [33, 17, 36]). Informally, the read and write operations of a multireader multiwriter atomic register are separated into a *request* (input) action and a *response* (output) action, concurrent operations executions are allowed, and every request eventually terminates in a matching response, in such a way as to produce the *illusion* of instantaneous operations.

The algorithm in Figure 6 is a version (due to Li and Vitanyi [25]) of the elegant and simple unbounded Vitanyi-Awerbuch algorithm [34]. The original solution is based on an unbounded construction that behaves in a manner similar a CTSS. We replace this construction by the LABEL and SCAN operations of the CTSS specification[9].

The code for the operations of MRMW is presented in two forms. Figure 7 presents the code in the precondition-effect notation commonly used to describe I/O Automata. Figure 6 uses psuedocode. We use the precondition-effect notation as the basis for the correctness proof and include the compact and intuitive psuedocode only for clarity. The only shared variables of MRMW are those of the CTSS. The local variables $\bar{o}_i$ and $\bar{v}_i$ contain the results of the SCAN_i operation. Recall that the nth process index in the array $\bar{o}_i$ contains the process index of the process currently associated with the "largest" label in the $\Longrightarrow$ ordering of LABEL operations.

```
READ_i
    SCAN_i(t̄_i, v̄_i)
    return (v_{i_max}) where max = o_{i_n}

WRITE_i(val_i)
    LABEL_i(val_i)
```

Figure 6: Psuedocode for MRMW.

In terms of the I/O Automata model, MRMW is an I/O Automaton with an operational interface. MRMW is the composition of n I/O Automata $\{p_1 \dots p_n\}$ and any I/O Automaton solving CTSS for n concurrent operations. The actions BEGINSCAN_i, $\text{ENDSCAN}_i(\bar{o}_i, \bar{v}_i)$,

[9][37] erroneously claims that the Vitanyi-Awerbuch algorithm [34] can be implement using a CTSS that only satisfies axioms **P0–P3**.

Shared State:
The shared state of the CTSS with initial values given by Figure 1.
Local State:
The local state of the CTSS with initial values given by Figure 1.
val_i: The value written by WRITE_i; initially v_o.
$v_{i_{max}}$: The value returned by READ_i; initially v_o.
$\bar{v}_i$: An array of values returned by SCAN_i; initially $(v_o \ldots v_o)$.
$\bar{o}_i$: An array of process indexes returned by SCAN_i; initially $(1 \ldots n)$.

READ_i:	BEGINREAD_i	*Eff:* $pc_i \leftarrow \text{BEGINSCAN}_i$
	BEGINSCAN_i	*Pre:* $pc_i = \text{BEGINSCAN}_i$ *Eff:* $pc_i \leftarrow \text{NIL}$
	$\text{ENDSCAN}_i(\bar{o}_i, \bar{v}_i)$	*Eff:* $pc_i \leftarrow \text{ENDREAD}_i(v_{i_{max}})$ where $max = o_{i_n}$
	$\text{ENDREAD}_i(v_{i_{max}})$	*Pre:* $pc_i = \text{ENDREAD}_i(v_{i_{max}})$ *Eff:* $pc_i \leftarrow \text{NIL}$
WRITE_i:	$\text{BEGINWRITE}_i(val_i)$	*Eff:* $pc_i \leftarrow \text{BEGINLABEL}_i(val_i)$
	$\text{BEGINLABEL}_i(val_i)$	*Pre:* $pc_i = \text{BEGINLABEL}_i(val_i)$ *Eff:* $pc_i \leftarrow \text{NIL}$
	ENDLABEL_i	*Eff:* $pc_i \leftarrow \text{ENDWRITE}_i$
	ENDWRITE_i	*Pre:* $pc_i = \text{ENDWRITE}_i$ *Eff:* $pc_i \leftarrow \text{NIL}_i$

Figure 7: Precondition-Effect code for MRMW.

BEGINLABEL$_i(val_i)$, and ENDLABEL$_i$ are the means by which p_i and the I/O Automaton solving CTSS communicate. These actions are hidden in MRMW. Each p_i is an I/O Automaton with an operational interface. The operation types of p_i are READ$_i$, WRITE$_i$, SCAN$_i$, and LABEL$_i$. The operation type READ$_i$ consists of the input action BEGINREAD$_i$ and the output action ENDREAD$_i(v_{i_{max}})$. The operation type WRITE$_i$ consists of the input action BEGINWRITE$_i(val_i)$ and the output action ENDWRITE$_i$. The operation type SCAN$_i$ consists of the output action BEGINSCAN$_i$ and the input action ENDSCAN$_i(\bar{o}_i, \bar{v}_i)$. The operation type LABEL$_i$ consists of the output action BEGINLABEL$_i(val_i)$ and the input action ENDLABEL$_i$. There are no internal actions for p_i. The set *states*(p_i) is the set of all possible states of p_i where each state is defined by the values of the variables of the shared and local state. The set *starts*(p_i) is the set consisting of the state defined by the initial values of the variables of the shared and local state. The set *steps*(p_i) is characterized by the *precondition* clause in each action. The set *part*(p_i) consists of the equivalence class C_i where C_i consists of BEGINSCAN$_i$, ENDREAD$_i(v_{i_{max}})$, BEGINLABEL$_i(val_i)$, and ENDWRITE$_i$.

We introduce the following notation: In any schedule β, where $beh(\beta) \in fairbehs$(MRMW) and $beh(\beta)$ is well-formed and response-live, denote the ath execution of WRITE$_i$ by $W_i^{[a]}$ and the ath execution of READ$_i$ by $R_i^{[a]}$. Since each WRITE operation results in exactly one LABEL operation and each READ operation results in exactly one SCAN operation, $L_i^{[a]}$ and $S_i^{[a]}$ are the the LABEL$_i$ operation of $W_i^{[a]}$ and the SCAN$_i$ operation of $R_i^{[a]}$ respectively. Define $x(i,a) = o_{i_n}$ for operation $R_i^{[a]}$. Intuitively, $x(i,a)$ is the index of the process that wrote the value returned by $R_i^{[a]}$. Let c be a choice function for β as characterized by **P0**–**P4** of Section 3. Define $r(i,a) = c(i,a,x(i,a))$ for operation $R_i^{[a]}$. Intuitively, $r(i,a)$ is the execution number of the WRITE operation that wrote the value returned by $R_i^{[a]}$. Since MRMW has an operational interface and $beh(\beta)$ is well-formed and response-live, Definition 2.8 gives a partial order $\longrightarrow$ on all READ and WRITE operations of β. By inspection of the code in Figure 7, the projection of β onto the actions in *exsig*(CTSS), β_c, yields a well-formed and response-live behavior, where $\beta_c \in behs$(CTSS). Consequently, Definition 2.8 gives a partial order $\longrightarrow'$ on all SCAN and LABEL operations of β. Note that $W_i^{[a]} \longrightarrow R_j^{[b]}$ implies that $L_i^{[a]} \longrightarrow' S_j^{[b]}$. However $L_i^{[a]} \longrightarrow' S_j^{[b]}$ does not imply $W_i^{[a]} \longrightarrow R_j^{[b]}$.

An *atomic* multireader multiwriter register is characterized by the following serial specifi-

cation S [23],[28]:

Definition 8.1 (serial specification S) Let s be a sequence of READ and WRITE operations. Then $s \in S$, if every READ operation returns the value written by the WRITE operation that immediately precedes the READ operation in s. If no such WRITE operation exists, the READ operation returns the initial value v_o. ■

In order to prove that the MRMW is an atomic multireader multiwriter register, we must show that MRMW is well-formed-preserving and response-live. Furthermore, we must show that for every well-formed and response-live behavior β, where $\beta \in$ *fairbehs*(MRMW), there exists an order $\Longrightarrow$ such that (see Definition 2.10):

1. $\Longrightarrow$ is a total order on all READ and WRITE operations that is consistent with $\longrightarrow$.
2. If s is the sequence of READ and WRITE operations ordered by $\Longrightarrow$, then $s \in S$ of Definition 8.1.

Consider any schedule β where $beh(\beta) \in$ *fairbehs*(MRMW) and $beh(\beta)$ is well-formed and response-live. Define order $\Longrightarrow'$ and choice function c for β_c as characterized by **P0–P4**. We construct $\Longrightarrow$ in several steps.

Notice that each WRITE operation includes a LABEL operation from the underlying CTSS. By **P1** the LABEL operations are totally ordered by $\Longrightarrow'$ in a manner that is consistent with the partial order $\longrightarrow'$. Now define $\Longrightarrow$ as follows:

$$W_i^{[a]} \Longrightarrow W_j^{[b]} \text{ iff } L_i^{[a]} \Longrightarrow' L_j^{[b]}.$$

Note that $\Longrightarrow$ so far is only defined on the WRITE operations. Now extend $\Longrightarrow$ to include the READ operations.

> Insert $R_i^{[a]}$ in $\Longrightarrow$ such that $R_i^{[a]}$ is between $W_{x(i,a)}^{[r(i,a)]}$ and the WRITE operation that immediately succeeds $W_{x(i,a)}^{[r(i,a)]}$ in $\Longrightarrow$. If $r(i,a) = 0$ then let $R_i^{[a]}$ precedes the first WRITE operation.

Now $\Longrightarrow$ orders each READ operation with respect to every WRITE operation. However, $\Longrightarrow$ is not yet a total order. READ operations are ordered amongst themselves only if they are transitively ordered by a WRITE operation. Let R be any set of READ operations that are

ordered between two WRITE operations that are consecutive in the $\Longrightarrow$ order. Now extend $\Longrightarrow$ such that the elements of R are totally ordered in a manner that is consistent with $\longrightarrow$. Repeat this procedure for each set of READ operations that are ordered between two WRITE operations that are consecutive in the $\Longrightarrow$ order. Finally, extend $\Longrightarrow$ for the READ operations that are ordered before any WRITE operations in a manner that is consistent with $\longrightarrow$. Now $\Longrightarrow$ is a total order. Specifically, $\Longrightarrow$ is irreflexive, antisymmetric, transitive and total. We now show that $\Longrightarrow$ is consistent with $\longrightarrow$.

Lemma 8.1 *For any $i, j \in \{1 \ldots n\}$, if $W_i^{[a]} \Longrightarrow W_j^{[b]}$ then $W_j^{[b]} \not\longrightarrow W_i^{[a]}$.*

Proof: Since $W_i^{[a]} \Longrightarrow W_j^{[b]}$, the construction of $\Longrightarrow$ shows that $L_i^{[a]} \Longrightarrow' L_j^{[b]}$. Now **P1a** implies that $L_j^{[b]} \not\longrightarrow' L_i^{[a]}$. Consequently $W_j^{[b]} \not\longrightarrow W_i^{[a]}$. ∎

Lemma 8.2 *For any $i, j \in \{1 \ldots n\}$, if $R_i^{[a]} \Longrightarrow W_j^{[b]}$ then $W_j^{[b]} \not\longrightarrow R_i^{[a]}$.*

Proof: We consider the cases $b = c(i,a,j)$, $b < c(i,a,j)$, and $b > c(i,a,j)$ separately.

$b = c(i,a,j)$: There are two cases to consider: $j = x(i,a)$ and $j \neq x(i,a)$. When $j = x(i,a)$, then by construction of the $\Longrightarrow'$ order, $c(i,a,j) = r(i,a)$ and $W_j^{[b]} \Longrightarrow R_i^{[a]}$. This contradicts the assumption that $R_i^{[a]} \Longrightarrow W_j^{[b]}$, so this case cannot arise. Now consider the case $j \neq x(i,a)$. Assume that $r(i,a) > 0$. Since $R_i^{[a]} \Longrightarrow W_j^{[b]}$, the construction of the $\Longrightarrow$ order implies that $W_{x(i,a)}^{[r(i,a)]} \Longrightarrow R_i^{[a]} \Longrightarrow W_j^{[b]}$. Consequently, $L_{x(i,a)}^{[r(i,a)]} \Longrightarrow' L_j^{[b]}$. Now, **P1b** implies that $S_i^{[a]}$ finds $x(i,a) < j$ in $\bar{o}_i$. However, by definition of $x(i,a)$ no such j exists. Therefore, this case cannot arise. Now consider the case $j \neq x(i,a)$ when $r(i,a) = 0$. Since $b \neq 0$, **P1b** implies that $S_i^{[a]}$ finds $x(i,a) < j$ in $\bar{o}_i$. However, by definition of $x(i,a)$ no such j exists. Therefore, this case cannot arise.

$b < c(i,a,j)$: In the previous case we proved that $W_j^{[c(i,a,j)]} \Longrightarrow R_i^{[a]}$. Since $b < c(i,a,j)$, it must be the case that $L_j^{[b]} \longrightarrow' L_j^{[c(i,a,j)]}$. Now, **P1a** shows that $L_j^{[b]} \Longrightarrow' L_j^{[c(i,a,j)]}$. Consequently, the construction of the $\Longrightarrow$ order implies $W_j^{[b]} \Longrightarrow W_j^{[c(i,a,j)]} \Longrightarrow R_i^{[a]}$, which contradicts the assumption that $R_i^{[a]} \Longrightarrow W_j^{[b]}$. Therefore, this case cannot arise.

$b > c(i,a,j)$: We proceed by showing that $L_j^{[b]} \not\longrightarrow' S_i^{[a]}$. In order to reach a contradiction, assume that $L_j^{[b]} \longrightarrow' S_i^{[a]}$. Assume also that $c(i,a,j) > 0$. Since $b > c(i,a,j)$, it follows

that $L_j^{[c(i,a,j)]} \longrightarrow' L_j^{[b]}$. Thus $L_j^{[c(i,a,j)]} \longrightarrow' L_j^{[b]} \longrightarrow' S_i^{[a]}$, which is impossible by **P2**. Therefore $L_j^{[b]} \not\rightarrow' S_i^{[a]}$. Furthermore, if $c(i,a,j) = 0$, **P2** directly show that $L_j^{[b]} \not\rightarrow' S_i^{[a]}$. Since $L_j^{[b]} \not\rightarrow' S_i^{[a]}$, we conclude that $W_j^{[b]} \not\rightarrow R_i^{[a]}$.

■

Lemma 8.3 *For any $i, j \in \{1 \ldots n\}$, if $W_j^{[b]} \Longrightarrow R_i^{[a]}$ then $R_i^{[a]} \not\rightarrow W_j^{[b]}$.*

Proof: We consider the cases $b = c(i,a,j)$, $b < c(i,a,j)$, and $b > c(i,a,j)$ separately.

$b = c(i,a,j)$: **P2** implies that $S_i^{[a]} \not\rightarrow' L_j^{[b]}$. This shows directly that $R_i^{[a]} \not\rightarrow' W_j^{[b]}$.

$b < c(i,a,j)$: **P2** implies that $S_i^{[a]} \not\rightarrow' L_j^{[c(i,a,j)]}$. Since $b < c(i,a,j)$, $L_j^{[b]} \longrightarrow' L_j^{[c(i,a,j)]}$. Consequently, $S_i^{[a]} \not\rightarrow' L_j^{[b]}$. This shows directly that $R_i^{[a]} \not\rightarrow' W_j^{[b]}$.

$b > c(i,a,j)$: We proceed by showing that $S_i^{[a]} \not\rightarrow' L_j^{[b]}$. In order to reach a contradiction, assume that $S_i^{[a]} \longrightarrow' L_j^{[b]}$. Assume that $r(i,a) > 0$. Now **P4** implies that $L_{x(i,a)}^{[r(i,a)]} \Longrightarrow' L_j^{[b]}$. By construction of the $\Longrightarrow$ order, this implies that $R_i^{[a]} \Longrightarrow W_j^{[b]}$. If $r(i,a) = 0$, the construction of the $\Longrightarrow$ order shows that $R_i^{[a]} \Longrightarrow W_j^{[b]}$. However, the fact that $R_i^{[a]} \Longrightarrow W_j^{[b]}$ contradicts that assumption that $W_j^{[b]} \Longrightarrow R_i^{[a]}$. Consequently, $S_i^{[a]} \not\rightarrow' L_j^{[b]}$. This shows immediately that $R_i^{[a]} \not\rightarrow W_j^{[b]}$.

■

Lemma 8.4 *For any $i, j \in \{1 \ldots n\}$, If $R_i^{[a]} \Longrightarrow R_j^{[b]}$ then $R_j^{[b]} \not\rightarrow R_i^{[a]}$.*

Proof: We consider two cases. First consider the case where there does not exist $W_k^{[d]}$ such that $R_i^{[a]} \Longrightarrow W_k^{[d]} \Longrightarrow R_j^{[b]}$. In this case the construction of the $\Longrightarrow$ order immediately shows that $R_j^{[b]} \not\rightarrow R_i^{[a]}$ when $R_i^{[a]} \Longrightarrow R_j^{[b]}$. For the second case assume that there exists $W_k^{[d]}$ such that $R_i^{[a]} \Longrightarrow W_k^{[d]} \Longrightarrow R_j^{[b]}$. The right-most $W_k^{[d]}$ is given by $k = x(j,b)$ and $d = r(j,b)$. Now define $k' = x(i,a)$ and $d' = r(i,a)$ assuming that $r(i,a) > 0$. Consequently,

$$W_{k'}^{[d']} \Longrightarrow R_i^{[a]} \Longrightarrow W_k^{[d]} \Longrightarrow R_j^{[b]}. \tag{12}$$

In order to reach a contradiction, we assume that $R_j^{[b]} \longrightarrow R_i^{[a]}$. Consider Equation 12. By definition of x and r, $S_j^{[b]}$ sees $v_k^{[d]}$, and $S_i^{[a]}$ sees $v_k^{[c(i,a,k)]}$. We now wish to show that $c(i,a,k) \neq$

d. To reach a contradiction assume that $c(i,a,k) = d$. Since $S_i^{[a]}$ sees $v_k^{[d]}$ and $v_{k'}^{[d']}$, and $k' = x(i,a)$, $S_i^{[a]}$ finds $k < k'$ in $\bar{o}_i$. Now **P1b** shows that $L_k^{[d]} \Longrightarrow' L_{k'}^{[d']}$. By definition of $\Longrightarrow$ this implies that $W_k^{[d]} \Longrightarrow W_{k'}^{[d']}$, which contradicts Equation 12. Thus $c(i,a,k) \neq d$.

By assumption $R_j^{[b]} \longrightarrow R_i^{[a]}$, thus $S_j^{[b]} \longrightarrow' S_i^{[a]}$. Since $S_j^{[b]}$ sees $v_k^{[d]}$ $S_i^{[a]}$ sees $v_k^{[c(i,a,k)]}$, and $c(i,a,k) \neq d$, **P3** now shows that $d < c(i,a,k)$. This implies that $L_k^{[d]} \Longrightarrow' L_k^{[c(i,a,k)]}$. Thus, by definition of $\Longrightarrow$ it follows that:

$$W_k^{[d]} \Longrightarrow W_k^{[c(i,a,k)]}. \tag{13}$$

Next we show that $W_k^{[c(i,a,k)]} \Longrightarrow R_i^{[a]}$. If not, the construction of the $\Longrightarrow$ order and the facts that $k' = x(i,a)$ and $d' = r(i,a)$ imply that $W_{k'}^{[d']} \Longrightarrow R_i^{[a]} \Longrightarrow W_k^{[c(i,a,k)]}$. Consequently, $L_{k'}^{[d']} \Longrightarrow L_k^{[c(i,a,k)]}$. Then, **P1b** implies that $S_i^{[a]}$ finds $k' = x(i,a) < k$ in $\bar{o}_i$. However, by definition of $x(i,a)$, no such k exists. Therefore $W_k^{[c(i,a,k)]} \Longrightarrow R_i^{[a]}$. This fact along with Equation 13 and the fact that $\Longrightarrow$ is transitive implies that $W_k^{[d]} \Longrightarrow R_i^{[a]}$. Thus we have a contradiction to Equation 12.

Finally, consider the case where $r(i,a) = 0$. As in the previous case, $R_i^{[a]} \Longrightarrow W_k^{[d]} \Longrightarrow R_j^{[b]}$, where $W_k^{[d]}$ is given by $k = x(j,b)$ and $d = r(j,b)$. Since $r(i,a) = 0$, the definition of $r(i,a)$ and **P1b** imply that $c(i,a,z) = 0$ for all $z \in \{1 \dots n\}$. In order to reach a contradiction assume that $R_j^{[b]} \longrightarrow R_i^{[a]}$. This implies that $S_j^{[b]} \longrightarrow' S_i^{[a]}$. Furthermore since $c(i,a,k) = 0$ and $d = r(j,b) > 0$, $c(i,a,k) \neq r(j,b)$. Now **P3** shows that $r(j,b) < c(i,a,k)$, which contradicts the fact that $c(i,a,k) = 0$ and $d = r(j,b) > 0$. ∎

We now show that the READ and WRITE operations ordered by the $\Longrightarrow$ order form a sequence permitted by the serial specification S of Definition 8.1.

Lemma 8.5 *Let s be the sequence of* READ *and* WRITE *operations of β ordered by the $\Longrightarrow$ order. Then $s \in S$.*

Proof: There are two cases: $r(i,a) > 0$ and $r(i,a) = 0$. When $r(i,a) > 0$ the definition of $\Longrightarrow$ implies that $R_i^{[a]}$ is immediately preceded by $W_{x(i,a)}^{[r(i,a)]}$, where $r(i,a) = c(i,a,x(i,a))$. Now, **P0** shows that $v_{i_{x(i,a)}} = val_{x(i,a)}^{[r(i,a)]}$. When $r(i,a) = 0$, the definition of $\Longrightarrow$ implies that $R_i^{[a]}$ precedes all WRITE operations. Also, **P0** shows that $v_{i_{x(i,a)}} = val_{x(i,a)}^{[0]} = v_o$. Noting that $R_i^{[a]}$ returns $v_{i_{x(i,a)}}$ completes the proof. ∎

Finally, we prove that MRMW is well-formed-preserving and response-live.

Lemma 8.6 MRMW *is well-formed-preserving and response-live.*

Proof: Notice, by inspecting the *precondition* clauses in the code of Figure 7, that for equivalence class C_i of *part*(MRMW), there is always at most one action enabled. Furthermore each action remains enabled until it is executed. Consequently, the actions must be executed in the sequence in which they are enabled. Furthermore, in a fair execution each enabled action will eventually be executed.

Now consider any fair execution whose behavior has a well-formed-input. Since CTSS is well-formed-preserving and response-live, inspection of the precondition-effects code in Figure 7 shows that the following sequence of actions are executed in response to a BEGINREAD$_i$ input action: BEGINSCAN$_i$, ENDSCAN$_i(\bar{o}_i, \bar{v}_i)$, and ENDREAD$_i(v_{i_{max}})$. In response to a BEGINWRITE$_i(val_i)$ input action, the following sequence of actions is executed: BEGINLABEL$_i(val_i)$, ENDLABEL$_i$, and ENDWRITE$_i$. Finally, no actions of C_i are enabled between the execution of a ENDREAD$_i(v_{i_{max}})$ or ENDWRITE$_i$ action and the next execution of a BEGINREAD$_i$ or a BEGINWRITE$_i(val_i)$ action. Inspection of these action sequences and the definitions of well-formed-preserving and response-live, immediately show that MRMW is well-formed-preserving and response-live. ■

We can now conclude that MRMW, if it uses our bounded CTSS construction, is a bounded atomic multireader multiwriter register.

Lemma 8.7 MRMW *is an atomic register satisfying serialization specification S.*

Proof: By Lemma 8.6, MRMW is well-formed-preserving and response-live. Now consider any behavior $\beta \in$ *fairbehs*(MRMW) that has a well-formed-input. Since MRMW is well-formed-preserving and response-live, β is well-formed and response-live. Consider the order $\Longrightarrow$ on the operations in β defined in the preceding discussion. Lemma 8.5 shows that the order satisfies the serial specification S. Lemma 8.1, Lemma 8.2, Lemma 8.3, and Lemma 8.4 show that $\Longrightarrow$ is consistent with partial order $\longrightarrow$. ■

8.2 Mutual Exclusion

The mutual exclusion problem, originally due to Dijkstra [10], is stated informally as follows (a more formal treatment that also introduces fault tolerance issues, can be found in [22])[10]. A system of n asynchronous processes communicate via shared memory consisting of single writer multireader atomic registers. The program of every process consists of two distinguished sections: a *remainder section* and a *critical section.* Processes alternate between executing the remainder and the critical section. The fundamental goal of the mutual exclusion algorithm is to limit the number of processes concurrently executing the critical section to 1. To solve the mutual exclusion problem, one is required to design *trying* and *exit* program sections to be performed before and after executing the critical section respectively. The trying section coordinates the entry into the critical section. In our algorithm the trying section has a subsection called the *doorway* section. This section is the first part of the trying section and is waitfree. The behavior of a mutual exclusion algorithm is characterized as follows (in order to simplify the discussion, this section uses a slightly less formal approach than the previous sections):

Mutual Exclusion: In any reachable state, no two process are executing the critical section.

Deadlock Freedom: In any reachable state, if there exists some process that is in the trying section, then there exist a process that is in the critical section or a process that will eventually enter the critical section.

Lockout Freedom:

1. In any execution, if there is no process that is forever executing the critical section, any process executing the trying section will eventually execute the critical section.
2. In any reachable state, if there is some process in the exit section, then some process will eventually enter the remainder section.

The fairness property of lockout freedom is strengthened in the following way.

First Come First Serve: If process p_i finishes executing the doorway section before process p_j begins executing the doorway section, then p_i executes the critical section before p_j does.

[10] Many solutions to the problem have been proposed over the years. (See [31].)

The psuedocode version of our mutual exclusion algorithm is presented in Figure 8. The algorithm is a simplified version of Lamport's Bakery Algorithm [19]. Our notation uses $\text{BEGINLABEL}_i()$ and ENDLABEL_i instead of just $\text{LABEL}_i()$ in order to clearly indicate what the atomic actions are. The reason for using BEGINSCAN_i and ENDSCAN_i instead of SCAN_i is the same. Lines $1-8$ represent the trying section and line 10 the exit section. The doorway section consists of lines $1-4$. In addition to the shared variables associated with the CTSS, each processes, p_i, has a shared variable called x_i which is implemented as a single writer multireader atomic register. Process p_i writes x_i and all other processes read x_i. The variable $\bar{o}_i$ is a local variable that contains the result of the SCAN_i operation of lines 6 and 7. Lines $1,2,3,4,6,7,9,10$, and 11 each represent atomic actions. Since lines 5 and 8 read the shared atomic variables x_j for $j \in \{1 \ldots n\}$, lines 5 and 8 consist of one atomic action for each time a particular x_j is read. For every execution of lines 5 and 8 each x_j, for $j \in \{1 \ldots n\}$, is read once. The states of the Lamport-Bakery mutual exclusion algorithm are defined by the values of the variables associated with the CTSS, the shared variables x_i for all i, as well as all local variables and the program counter, pc, of each process.

Our correctness proof essentially follows the arguments given in [28] and [22]. The contribution of our proof is that it is based on the CTSS specification. We now introduce some notation that will be used in the correctness proof. Consider the state s in any execution. If process p_i is not executing the LABEL_i operation in state s, in other words $pc_i \neq 2$ and $pc_i \neq 3$, we define the function $l(i,s)$ which is a function from the set of process indexes and the set of states to the set of execution numbers of the LABEL operations of the execution. $s(i,s)$ is defined in a similar manner for the SCAN_i operations.

Definition 8.2 (function l) Consider an execution α. Let s be a state in α where $pc_i \neq 2$ and $pc_i \neq 3$. Then, define $l(i,s)$ to be the execution number of the LABEL_i operation whose ENDLABEL_i action was the last ENDLABEL_i action executed in α before state s. ■

Definition 8.3 (function s) Consider an execution α. Let s be a state in α where $pc_i \neq 6$ and $pc_i \neq 7$. Then, define $s(i,s)$ to be the execution number of the SCAN_i operation whose ENDSCAN_i action was the last ENDSCAN_i action executed in α before state s. ■

Intuitively, for a state s, $L_i^{[l(i,s)]}$ is the most recently executed LABEL$_i$ operation and $S_i^{[s(i,s)]}$ is the most recently executed SCAN$_i$ operation. In order to simplify the presentation, we do not provide the argument for why p_i has $pc_i \neq 2$ and $pc_i \neq 3$ or $pc_i \neq 6$ and $pc_i \neq 7$ when discussing $L_i^{[l(i,s)]}$ or $S_i^{[s(i,s)]}$ in cases where it is obvious. The order $\longrightarrow$ is used to order states of an execution as well as the CTSS operation instances in the execution.

Definition 8.4 ($\longrightarrow$ order) Let A_1 and A_2 be CTSS operation instances and s_1 and s_2 be occurrences of states in an execution α of the Lamport-Bakery mutual exclusion algorithm. Then:

1. $A_1 \longrightarrow A_2$ iff the response action associated with A_1 occurs before the request action associated with A_2.
2. $A_1 \longrightarrow s_1$ iff the response action associated with A_1 occurs before s_1.
3. $s_1 \longrightarrow A_1$ iff the request action associated with A_1 occurs after s_1.
4. $s_1 \longrightarrow s_2$ iff s_1 occurs before s_2.

■

Note that $\longrightarrow$ provides a total order for the states and a partial order for the CTSS operation instances. Now consider any execution α of the Lamport-Bakery mutual exclusion algorithm. We wish to show that the execution satisfies the four properties for mutual exclusion given above. Notice that the projection of the execution onto the external actions of CTSS, gives a behavior of CTSS that has a well-formed-input. Consequently, the projection of the execution onto the external actions of CTSS must satisfy axioms **P0**, **P1**, and **P2**[11] of Section 3. Let $\Longrightarrow$ and c be an order and a choice function that satisfy **P0**, **P1**, and **P2** for the projection of α onto the external actions of CTSS. Now consider the following lemma which will be used to prove the mutual exclusion property.

Lemma 8.8 *In any state s of the execution α, if p_i is in the critical section and $x_j = T$ then* $L_i^{[l(i,s)]} \Longrightarrow L_j^{[l(j,s)]}$.

[11] Axioms **P3** and **P4** are not needed for the Lamport-Bakery mutual exclusion algorithm.

```
            repeat forever
1                 x_i ← L
2                 BEGINLABEL_i()
3                 ENDLABEL_i
4                 x_i ← T
5           L1:   If ∃j such that x_j = L then goto L1
6           L2:   BEGINSCAN_i
7                 ō_i ← ENDSCAN_i
8                 If ∃j such that j < i in ō_i and x_j = T then goto L2
9                 critical section
10                x_i ← NIL
11                remainder section
            end repeat
```

Figure 8: Psuedocode for Lamport-Bakery mutual exclusion algorithm

Proof: Consider the first state in the execution α after the action in which p_i reads $x_j \neq L$ in line 5 for the last time before state s. Call this state s_1. Since $x_j \neq L$, $pc_j \neq 2$ and $pc_i \neq 3$ in state s_1. Hence we can now consider two cases: $L_j^{[l(j,s)]} \longrightarrow s_1$ and $s_1 \longrightarrow L_j^{[l(j,s)]}$.

$\underline{L_j^{[l(j,s)]} \longrightarrow s_1}$: Consider the last state in α before the action in which p_i considers p_j for the last time in line 8 before state s. Call this state s_2. Since p_i enters the critical section, there are three cases to consider: $i < j$ in $\bar{o}_i$ and $x_j = T$, $i < j$ in $\bar{o}_i$ and $x_j \neq T$, and $j < i$ in $\bar{o}_i$ and $x_j \neq T$. We consider the last two cases together by showing that the case $x_j \neq T$ cannot arise.

$\underline{i < j \text{ in } \bar{o}_i \text{ and } x_j = T}$: In this case $L_j^{[l(j,s)]} \longrightarrow s_1 \longrightarrow S_i^{[s(i,s_2)]}$, therefore $L_j^{[l(j,s)]} \longrightarrow S_i^{[s(i,s_2)]}$. Furthermore, by definition of $l(j,s)$ there exists no $L_j^{[b]}$, where $b \neq l(j,s)$, such that $L_j^{[l(j,s)]} \longrightarrow L_j^{[b]} \longrightarrow S_i^{[s(i,s_2)]}$. Consequently, **P2** shows that $c(i, s(i,s_2), j) = l(j,s)$. The same argument shows that $c(i, s(i,s_2), i) = l(i,s)$. Since p_i found $i < j$ in $\bar{o}_i$ of $S_i^{[s(i,s_2)]}$, **P1b** shows that $L_i^{[l(i,s)]} \Longrightarrow L_j^{[l(j,s)]}$.

$\underline{x_j \neq T}$: In this case $x_j = \text{NIL}$ or $x_j = L$. If $x_j = \text{NIL}$, then since $x_j = T$ in state s and $s_2 \longrightarrow s$, p_j must execute the LABEL$_j$ operation of lines 2 and 3 between s_2 and s. Consequently $s_1 \longrightarrow L_j^{[l(j,s)]}$, which contradicts the assumption that $L_j^{[l(j,s)]} \longrightarrow s_1$. So, it must be that $x_j = L$ in state s_2. Recall that $x_j \neq L$ in s_1 and $x_j = T$ in s. Since $s_1 \longrightarrow s_2 \longrightarrow s$, inspection of the code shows that p_j must execute the LABEL$_j$ operation of lines 2 and 3 between s_1 and s. This implies that $s_1 \longrightarrow L_j^{[l(j,s)]}$, which contradicts the

assumption that $L_j^{[l(j,s)]} \longrightarrow s_1$. Therefore this case cannot arise.

$s_1 \longrightarrow L_j^{[l(j,s)]}$: Since $L_i^{[l(i,s)]} \longrightarrow s_1$, we can conclude that $L_i^{[l(i,s)]} \longrightarrow L_j^{[l(j,s)]}$. Now **P1a** implies that $L_i^{[l(i,s)]} \Longrightarrow L_j^{[l(j,s)]}$.

■

With this lemma, it is easy to show mutual exclusion.

Lemma 8.9 *In any state s of the execution α, if p_i is in the critical section, then there exists no $j \neq i$ such that p_j is in the critical section.*

Proof: We proceed by contradiction. Assume that there exists a state s such that p_i and p_j are in the critical section where $i \neq j$. Since p_i and p_j are in the critical section, $x_i = T$ and $x_j = T$. Now Lemma 8.8 implies that $L_i^{[l(i,s)]} \Longrightarrow L_j^{[l(j,s)]}$ and $L_j^{[l(j,s)]} \Longrightarrow L_i^{[l(i,s)]}$. By **P1**, $\Longrightarrow$ is a total order, so we have a contradiction. ■

The following Lemma shows that Lamport-Bakery mutual exclusion algorithm satisfies the *fcfs* property.

Lemma 8.10 *Consider the execution α. Let s_i be any state after p_i executes the action on line 3 but before p_i is in the critical section for the first time after the execution of the action. Let s_j be any state before p_j executes the action on line 2 such that p_j must execute line 2 before it enters the critical section for the first time after s_j. Assume that $s_i \longrightarrow s_j$. Let s_{c_i} be the first state in which p_i is in the critical section after s_i. Let s_{c_j} be the first state in which p_j is in the critical section after s_j. Then $s_{c_i} \longrightarrow s_{c_j}$.*

Proof: For a contradiction assume that $s_{c_j} \longrightarrow s_{c_i}$. In s_{c_j}, p_j is in the critical section and $x_i = T$. Hence Lemma 8.8 implies that $L_j^{[l(j,s_{c_j})]} \Longrightarrow L_i^{[l(i,s_{c_j})]}$. However, since $s_i \longrightarrow s_j$, we know that $L_i^{[l(i,s_{c_j})]} \longrightarrow L_j^{[l(j,s_{c_j})]}$, which by **P1a** is a contradiction. ■

Next we consider the deadlock freedom property of the Lamport-Bakery mutual exclusion algorithm. We consider the second part of the property first.

Lemma 8.11 *Suppose that process p_i is in the exit section in state s of execution α. Then p_i will eventually enter the remainder section.*

Proof: The lemma follows immediately from the fact that the exit section, line 10, consists of a single waitfree action. ■

Lemma 8.12 *If p_i is in the trying section in state s_i of execution α, then there exist some process that is in the critical section, or there exists some process that eventually enters the critical section.*

Proof: Let p_i be in the trying section in s_i. If there exist some process in the critical section in s_i, then we are done. Therefore, assume that no such process exists. Let s_c, where $s_i \longrightarrow s_c$, be the state in which the first processes is in the critical region after s_i. Since the code of Figure 8 is waitfree, except for lines 5 and 8, p_i will eventually reach line 5. Label this state in the execution as s_1. Now let S be the set of processes that are in the trying section in state s_1. If there are any processes in the exit section in state s_1, Lemma 8.11 implies that there exists a state s_2, where $s_1 \longrightarrow s_2$, such that there are no processes in the exit section in state s_2.

Let $p_k \in \{1 \dots n\} - S$. If $x_k = T$ in any state between s_2 and s_c, it must be that p_k executes the LABEL_k operation of lines 2 and 3 after the state s_1. Furthermore p_i last executes the LABEL_i operation before state s_1 Hence **P1a** shows that for any state s between s_2 and s_c where $x_k = T$:

$$L_i^{[l(i,s)]} \implies L_k^{[l(k,s)]}. \tag{14}$$

Consider any $p_j \in S$. If $x_j = T$, then $L_j^{[l(j,s_2)]}$ is defined. If $x_j = L$, then $L_j^{[l(j,s_2)]}$ may not be defined. Since lines $1-4$ are waitfree, it will eventually be the case that $x_j = T$ for all $p_j \in S$. Call this state s_3. Now $L_j^{[l(j,s_3)]}$ is defined for all $p_j \in S$. Consider $p_j \in S$ such that $L_j^{[l(j,s_3)]} \implies L_k^{[l(k,s_3)]}$ for all $p_k \in S$ and $k \neq j$. By **P1**, $\implies$ is a total order, hence p_j exists. Since none of the processes in S pick a new label between s_3 and s_c, $L_j^{[l(j,s)]} \implies L_k^{[l(k,s)]}$ for all $k \neq j$, $k \in S$, and s between s_3 and s_c. Furthermore, for all $p_k \in \{1 \dots n\} - S$ where $x_k = T$ and s between s_3 and s_c, $L_j^{[l(j,s)]} \implies L_k^{[l(k,s)]}$. This is a consequence of Equation 14 which shows that $L_i^{[l(i,s)]} \implies L_k^{[l(k,s)]}$ and the definition of p_j which shows that $L_j^{[l(j,s)]} \implies L_i^{[l(i,s)]}$.

The process p_j will progress past line 5 unless there exists some process p_k such that $x_k = L$. Eventually, it must be that $x_k \neq L$. Furthermore, $x_k \neq L$ at least until s_c. Thus each process that is preventing p_j's process at line 5 will eventually have $x_k \neq L$. At this point p_j will advance to line 8. Process p_j will advance to the critical section unless there exists

some processes p_k such that $x_k = T$ and p_j orders $k < j$ in the $\bar{o_j}$ returned by the SCAN$_j$ operation executed in lines 6 and 7 just prior to p_j finding $k < j$ in line 8. Since the SCAN$_j$ operation of lines 6 and 7 continues to be executed while there exists some processes p_k such that $x_k = T$ and $k < j$ in $\bar{o_j}$, there must eventually be a state s between states s_3 and s_c such that $L_k^{[l(k,s)]} \longrightarrow S_j^{[s(j,s)]}$. By definition of $l(k,s)$ there exists no $L_k^{[b]}$, where $b \neq l(k,s)$, such that $L_k^{[l(k,s)]} \longrightarrow L_k^{[b]} \longrightarrow S_j^{[s(j,s)]}$. Consequently **P2** shows that $c(j, s(j,s), k) = l(k,s)$. The same argument shows that $c(j, s(j,s), j) = l(j,s)$. Since p_j orders $k < j$ in $\bar{o_j}$, **P1b** shows that $L_k^{[l(k,s)]} \Longrightarrow L_j^{[l(j,s)]}$. However, such a k cannot exist in state s since s is between the states s_3 and s_c and, for all states s' between s_3 and s_c, $L_j^{[l(j,s')]} \Longrightarrow L_k^{[l(k,s')]}$ for all $k \in S$ and all $k \in \{1 \ldots n\} - S$ where $x_k = T$. Therefore, p_j will eventually enter the critical section. ■

Finally, we consider the no lockout property.

Lemma 8.13 *Suppose in the state s_i in execution α, p_i is in the trying section. If there is no p_j, such that p_j is in the critical section for all states after some state s_j, then p_i will eventually enter the critical section.*

Proof: The first 4 lines of the trying section are waitfree. Therefore, p_i will eventually complete these lines. Call the first state after line 4 completes s_1. Let S be the set of processes p_j for which it is possible that p_j is in the critical section in some state which succeeds s_1, but proceeds the state in which p_i enters the critical section. Clearly $S \subseteq \{1, \ldots, n\} - \{i\}$. Since p_i is in the trying section Lemma 8.12 says that p_i or some $p_j \in S$ will eventually enter the critical section. The proof is complete if p_i enters the critical section, so assume that p_j enters the critical section. After p_j exits the critical section, p_j must start executing line 2 after some state s_j, where $s_i \longrightarrow s_j$, before p_j enters the critical section a subsequent time. Now Lemma 8.10, shows that $S = S - \{j\}$ after after p_j exits the critical section. We repeat this argument until $S = \emptyset$. Then Lemma 8.12 says that p_i eventually enters the critical section. ■

9 Formal Justification for Use of Snapshot

The purpose of this section is to formally justify the manner in which the snapshot operations SNAP and UPDATE of [1] are used in BCTSS and UCTSS. Specifically, we must justify the fact that we do not use separate actions for the invocation and response of each snapshot operation.

9.1 Theory

In order to provide a strong theoretical foundation for the discussion, we extend some of the concepts introduced in Section 2. Most of the ideas in following discussion are taken from Goldman, Lynch and Yelick [15]. We present a simplified and less general version of their results.

Goldman et al. introduce the concept of an *environment*, a *process* and an *object*. Intuitively, an environment refers to the user of a particular I/O Automaton. The I/O Automata model generally does not model the users of I/O Automata except to describe the situations in which a user is expected to issue input actions. A process is an I/O Automaton that performs an operation on behalf of the environment. Typically the interface between the environment and a process is described by a set of input actions that are used by the environment to request an operation and output actions that are used by the process to respond to an operation request. Finally, objects are I/O Automata that model shared data types that provide a means for a set of processes to communicate. The following discussion formalizes these concepts. Note that we largely retain the notational conventions used in Section 2.

Definition 9.1 (object I/O Automata) An *object* I/O Automaton, o, which can be used by n process I/O Automata (see Definition 9.2) is an I/O Automaton with an operational interface which is characterized as follows. For each $i \in \{1 \ldots n\}$, there exists a disjoint set of operation types $ops_i(o) \subseteq ops(exsig(o))$. For each operation type $a_i \in ops_i(o)$, we denote the input actions by $\text{INVOKE}_{o,p_i}(a_i, v)$ and the output actions by $\text{RESPONSE}_{o,p_i}(a_i, r)$. ■

As a shorthand for an object I/O Automaton we use the term object. The subscript o, p_i indicate that a process I/O Automaton denoted by p_i will use this action to communicate with the object o when o and p_i are composed. We now present a formal definition for a process I/O Automaton.

Definition 9.2 (process I/O Automata) A *process* I/O Automaton, p_i, is an I/O Automaton with an operational interface which is comprised of two disjoint sets of operation types:

- There are a set of operation types which describe the interface between the process and the environment. For any such operation type called a_i we denote the input actions by $\text{INVOKE}_{p_i}(a_i, v)$ and the output actions by $\text{RESPONSE}_{p_i}(a_i, r)$.

- There are a set of operation types which describe the interface between the process and an object[12] denoted by o. For any such operation type called a_i we denote the input actions[13] by $\text{RESPONSE}_{o,p_i}(a_i, r)$ and the output actions by $\text{INVOKE}_{o,p_i}(a_i, v)$.

■

For the discussion that follows, let A be any I/O Automaton that is a composition of n processes $\{p_1 \ldots p_n\}$ and one object o where the external actions of o are hidden. We now define various characteristics of schedules of A. These characteristics will be used in the definition of an I/O Automaton called an IR system. Let β be a schedule of A. Then $\beta|p_i$ is the projection of β onto all $\text{INVOKE}_{p_i}(a_i, v)$ and $\text{RESPONSE}_{p_i}(a_i, r)$ actions that constitute p_i's interface with the environment. Similarly, $\beta|o, p_i$ is the projection of β onto all $\text{INVOKE}_{o,p_i}(a_i, v)$ and $\text{RESPONSE}_{o,p_i}(a_i, r)$ actions that constitute p_i's interface with the object o. In order to insure that a process only issues requests to an object when that process is servicing a request from the environment, we introduce the concept of a process p_i being *active* after a prefix of a particular schedule. Specifically, a process p_i is active after a prefix β' of the schedule β of A if the last action in $\beta'|p_i$ is an $\text{INVOKE}_{p_i}(a_i, v)$ action.

Definition 9.3 (IR-well-formed) Let β be a schedule of A. We say β is *IR-well-formed* if

1. $beh(\beta)$ is well-formed.
2. Every $\text{INVOKE}_{o,p_i}(a_i, v)$ action in $\beta|o, p_i$ occurs from a prefix of β after which p_i is active.
3. $\beta|o, p_i$ consists of an alternating sequence of input and output actions of o, starting with an input action, such that each $\text{RESPONSE}_{o,p_i}(a_i, r)$ action is immediately preceded by an $\text{INVOKE}_{o,p_i}(a_i, v)$ action.
4. In β no actions of p_i occur between any pair of corresponding $\text{INVOKE}_{o,p_i}(a_i, v)$ and $\text{RESPONSE}_{o,p_i}(a_i, r)$ actions.

■

[12] [15] allows processes to have an interface to an arbitrary number of objects. For the sake of simplicity, we restrict attention to processes which have an interface to only one object.

[13] Notice that we have changed the notational convention for the process' interface with the object. This arises from the fact that the input actions of the object must have the same name as the output actions of the process. In this way, the process can initiate operation instances on the object (see discussion of composition in Section 2).

Definition 9.4 (IR-well-formed-preserving) Let β be a schedule of A. β is *IR-well-formed-preserving* if, for all prefixes β' of β, where $beh(\beta')$ has a well-formed-input, β' is IR-well-formed. ■

We say that A is IR-well-formed-preserving if every schedule of A is IR-well-formed-preserving.

Definition 9.5 (IR system) Let A be an I/O Automaton that is a composition of n processes, $\{p_1 \ldots p_n\}$, and one object, o, where the external actions of o are hidden. A is an IR system iff:

1. The object o of A is an atomic I/O Automaton that satisfies some specification S.
2. A is IR-well-formed-preserving.
3. A is response-live.

■

We now define an IRA system which is the same as an IR system except that it combines the $\text{INVOKE}_{o,p_i}(a_i, v)$ and $\text{RESPONSE}_{o,p_i}(a_i, r)$ actions into a single action called $\text{ATOMIC}_{o,p_i}(a_i, v, r)$.

Definition 9.6 (IRA system) Let $I = \{1 \ldots n\}$. Let A be an IR system composed of n processes, $\{p_1 \ldots p_n\}$, and an atomic object, o, satisfying specification S. Then the IRA system A' that *corresponds* to A is defined as follows:

- $states(A') = states(A)$
- $start(A') = start(A)$
- $sig(A') = (in(A), out(A), (int(A) - \bigcup_{i \in I}\{\text{INVOKE}_{o,p_i}(a_i, v), \text{RESPONSE}_{o,p_i}(a_i, r)\}) \cup \bigcup_{i \in I}\{\text{ATOMIC}_{o,p_i}(a_i, v, r)\}.$
- $steps(A')$ = the set of all steps (a, π, a'') such that either:
  - $\pi \notin \bigcup_{i \in I}\{\text{INVOKE}_{o,p_i}(a_i, v), \text{RESPONSE}_{o,p_i}(a_i, r)\}$ and $(a, \pi, a'') \in steps(A)$.

- $\pi \in \bigcup_{i \in I}\{\text{ATOMIC}_{o,p_i}(a_i, v, r)\}$ and there exists state a' of A such that: $(a, \text{INVOKE}_{o,p_i}(a_i, v), a') \in steps(A)$ and $(a', \text{RESPONSE}_{o,p_i}(a_i, r), a'') \in steps(A)$, and, for any schedule β of A', the projection of β onto the set of all $\text{ATOMIC}_{o,p_i}(a_i, v, r)$ actions must be an element of the sequential specification of the atomic object o.

- $part(A') = part(A)$ except that the set of $\text{ATOMIC}_{o,p_i}(a_i, v, r)$ actions, for all v and r, replace the set of $\text{INVOKE}_{o,p_i}(a_i, v)$ actions for all v.

■

In the action signature we are replacing pair of actions $\text{INVOKE}_{o,p_i}(a_i, v)$, $\text{RESPONSE}_{o,p_i}(a_i, r)$ by a single action $\text{ATOMIC}_{o,p_i}(a_i, v, r)$ such that $\text{ATOMIC}_{o,p_i}(a_i, v, r)$ can be executed in A' for situations where the pair of actions $\text{INVOKE}_{o,p_i}(a_i, v)$, $\text{RESPONSE}_{o,p_i}(a_i, r)$ can be executed in A. The following significant theorem due to Goldman et al. [15] can be used to show that A' implements A.

Theorem 9.1 *Let A be an* IR *system and A' be the* IRA *system corresponding to A. If α is a fair execution of A, then there exists a fair execution α' of A' such that $beh(\alpha') = beh(\alpha)$.*

Corollary 9.2 *Let A be an* IR *system. Then A implements the* IRA *system corresponding it.*

Proof: This follows immediately from Theorem 9.1. ■

9.2 Proof

Figure 9 shows the code for UCTSS and BCTSS[14] that uses the invocation and response actions for SNAP_i and UPDATE_i. We call these new I/O Automata UCTSS′ and BCTSS′. Since the interface provided by [1] uses request and response actions, we can technically only use the SNAP_i and UPDATE_i primitives as is done in UCTSS′ and BCTSS′. In order to show that UCTSS′ and BCTSS′ solve CTSS will will show that UCTSS′ implements UCTSS and BCTSS′ implements BCTSS.

We proceed as follows. We show that UCTSS′ and BCTSS′ are IR systems, and then note that the IRA systems corresponding to UCTSS′ and BCTSS′ are UCTSS and BCTSS respectively.

[14] UCTSS and BCTSS share the code that is relevant to this discussion.

SCAN$_i$:	
BEGINSCAN$_i$	*Eff:* $op_i \leftarrow$ SCAN$_i$ $pc_i \leftarrow$ BEGINSNAP$_i$
BEGINSNAP$_i$	*Pre:* $pc_i =$ BEGINSNAP$_i$ *Eff:* $pc_i \leftarrow$ NIL
ENDSNAP$_i(\bar{t}_i, \bar{v}_i)$	*Eff:* **If** $op_i =$ SCAN$_i$ **then** $\bar{o}_i \leftarrow$ the sequence of indexes where j appears before k in o_i iff $(t_j, j) \ll (t_k, k)$ $pc_i \leftarrow$ ENDSCAN$_i(\bar{o}_i, \bar{v}_i)$ **If** $op_i =$ LABEL$_i$ **then** $nt_i \leftarrow$ NEWLABEL$_i(\bar{t}_i)$ $pc_i \leftarrow$ BEGINUPDATE$_i((t_i, v_i), (nt_i, val_i))$
ENDSCAN$_i(\bar{o}_i, \bar{v}_i)$	*Pre:* $pc_i =$ ENDSCAN$_i(\bar{o}_i, \bar{v}_i)$ *Eff:* $pc_i \leftarrow$ NIL
LABEL$_i$:	
BEGINLABEL$_i(val_i)$	*Eff:* $op_i \leftarrow$ LABEL$_i$ $pc_i \leftarrow$ BEGINSNAP$_i$
BEGINUPDATE$_i((t_i, v_i), (nt_i, val_i))$	*Pre:* $pc_i =$ BEGINUPDATE$_i((t_i, v_i), (nt_i, val_i))$ *Eff:* $pc_i \leftarrow$ NIL$_i$
ENDUPDATE$_i$	*Eff:* $pc_i \leftarrow$ ENDLABEL$_i$
ENDLABEL$_i$	*Pre:* $pc_i =$ ENDLABEL$_i$ *Eff:* $pc_i \leftarrow$ NIL

Figure 9: Precondition-Effect code for UCTSS′ and BCTSS′

This will allows us to use Corollary 9.2 to conclude that UCTSS′ implements UCTSS and BCTSS′ implements BCTSS.

Formally, UCTSS′ and BCTSS′ are a composition of n process I/O Automata $\{p_1 \dots p_n\}$ and one object I/O Automaton o where p_i and o are defined as follows: Each process I/O Automaton has two operation types that constitute its interface with the environment, LABEL_i and SCAN_i. The object interface of p_i consists of the SNAP_i and the UPDATE_i operation types. These operation types consist of the following external actions: LABEL_i consists of the input action $\text{BEGINLABEL}_i(val_i)$ and the output action ENDLABEL_i. SCAN_i consists of the input action BEGINSCAN_i and the output action $\text{ENDSCAN}_i(\bar{o}_i, \bar{v}_i)$. SNAP_i consists of the output action BEGINSNAP_i and the input action $\text{ENDSNAP}_i(\bar{t}_i, \bar{v}_i)$. UPDATE_i consists of the output action $\text{BEGINUPDATE}_i((t_i, v_i), (nt_i, val_i))$ and the input action ENDUPDATE_i. There are no internal actions. The partition is the same as it was for the UCTSS and BCTSS version of p_i (see Section 4) except that BEGINSNAP_i replaces $\text{SNAP}_i(\bar{t}_i, \bar{v}_i)$ and $\text{BEGINUPDATE}_i((t_i, v_i), (nt_i, val_i))$ replaces $\text{UPDATE}_i((t_i, v_i), (nt_i, val_i))$. The steps of p_i are determined by the pc_i variable, and the states and start states are defined as they were for the UCTSS and BCTSS version of p_i. The object I/O Automaton o is the implementation of the snapshot object given in [1]. We do not provide the code for o, but present some of its characteristics relevant to our discussion. The interface with the processes consists of $2n$ operations types SNAP_i and UPDATE_i for $i \in \{1 \dots n\}$. Each of theses operation types consists of the following external actions: SNAP_i consists of the input action BEGINSNAP_i and the output action $\text{ENDSNAP}_i(\bar{t}_i, \bar{v}_i)$, and UPDATE_i consists of the input action $\text{BEGINUPDATE}_i((t_i, v_i), (nt_i, val_i))$ and the output action ENDUPDATE_i. Furthermore, o is an atomic I/O Automaton satisfying the SNAPSHOT serial specification.

Definition 9.7 (SNAPSHOT serial specification) A sequence of operations instances α is in SNAPSHOT if and only if the following conditions hold. For any i, if a SNAP_i operation instance returns the set of values, $\bar{v}$, and labels, $\bar{t}$, v_k and t_k are the value and label written by the UPDATE_k operation instance that immediately proceeds SNAP_i in α. If a SNAP_i operation instance is not proceeded by a UPDATE_k operation instance, then v_k and t_k are equal to their initial values. ■

Lemma 9.3 *UCTSS′ and BCTSS′ are IR systems.*

Proof: From [1] we know that the object I/O Automaton of UCTSS′ and BCTSS′ is an atomic object I/O Automaton that satisfies the SNAPSHOT serial specification given in Definition 9.7. So we must show that UCTSS′ and BCTSS′ are IR-well-formed-preserving and response-live.

Notice by inspecting the *precondition* clauses in the code of Figure 9 that for any equivalence class C_i of *part*(UCTSS′) and *part*(BCTSS′), there is always at most one action enabled. Furthermore each action remains enabled until it is executed. Consequently, the actions must be executed in the sequence in which they are enabled. Furthermore, in a fair execution each enabled action will eventually be executed.

Now consider any fair execution whose behavior has a well-formed-input. Since the object o is well-formed-preserving and response-live, inspection of the precondition-effects code in Figure 9 shows that the following sequence of actions is executed in response to a BEGINSCAN$_i$ input action: BEGINSNAP$_i$, ENDSNAP$_i(\bar{t_i}, \bar{v_i})$, and ENDSCAN$_i(\bar{o_i}, \bar{v_i})$. Following a BEGINLABEL$_i(val_i)$ input action, the following sequence of actions is executed: BEGINSNAP$_i$, ENDSNAP$_i(\bar{t_i}, \bar{v_i})$, BEGINUPDATE$_i((t_i, v_i), (nt_i, val_i))$, ENDUPDATE$_i$, and ENDLABEL$_i$. Finally, no actions of C_i are enabled between the execution of a ENDSCAN$_i(\bar{o_i}, \bar{v_i})$ or ENDLABEL$_i$ action and the next execution of a BEGINSCAN$_i$ or BEGINLABEL$_i(val_i)$ action. Inspection of these action sequences and the definitions of IR-well-formed-preserving and response-live, immediately show that UCTSS′ and BCTSS′ are IR-well-formed-preserving and response-live. ■

Now that we have shown that UCTSS′ and BCTSS′ are IR systems, note that the IRA systems corresponding to UCTSS′ and BCTSS′ are UCTSS and BCTSS respectively. Specifically, in UCTSS and BCTSS the BEGINSNAP$_i$ and ENDSNAP$_i(\bar{t_i}, \bar{v_i})$ actions of UCTSS′ and BCTSS′ are replaced by the SNAP$_i(\bar{t_i}, \bar{v_i})$ action. Similarly, the BEGINUPDATE$_i((t_i, v_i), (nt_i, val_i))$ and ENDUPDATE$_i$ actions are replaced by the UPDATE$_i((t_i, v_i), (nt_i, val_i))$ action (see Definition 9.6).

Theorem 9.4 BCTSS′ *and* UCTSS′ *solve* CTSS.

Proof: Using Corollary 9.2 we conclude that BCTSS′ implements BCTSS and UCTSS′ implements UCTSS. From Theorem 7.3 we know that BCTSS solves CTSS, hence BCTSS′ solves CTSS. Similarly, Lemma 4.10 shows that UCTSS solves CTSS, therefore UCTSS′ solves CTSS. ■

10 Discussion and Future Work

Critical to constructing and proving the correctness of our simple bounded timestamping system are the design technique of composition and the analysis techniques provided by the I/O Automata Model.

The composition of the label structure of [11] with the atomic snapshot primitive of [1] greatly reduces the complexity of our algorithm relative to [11]. Many possible executions are eliminated by the fact that the snapshot primitive returns an *instantaneous* (in the sense of [20]) view of the current labels. Even though the construction of the snapshot primitive is complex, its complexity is hidden from the timestamping system. Our simple constructions for the multireader multiwriter atomic register and first come first serve mutual exclusion further demonstrate the power of using composition to simplify the design and analysis of algorithms.

Due to the fact that our algorithm uses the snapshot primitive, the complexity of our timestamping system is worse by $O(\sqrt{n})$ than the most efficient known bounded timestamping system [12]. The complexity of our bounded timestamping system is the same as the complexity of the underlying snapshot primitive. The complexity of the original construction in [1] was $O(n^2)$. The best construction currently known has complexity $O(n\sqrt{n})$ [3]. In addition to our bounded timestamping system, there are several other areas in which the snapshot primitive is useful (see [1]). Consequently, improving the complexity of the snapshot primitives would provide a significant contribution. Since the SNAP operation must read n registers, $\Omega(n)$ is a lower bound for the SNAP operation. We see no reason why $O(n)$ algorithms for both the SNAP and UPDATE operations should not be possible.

An important feature of the I/O Automata Model is the concept of stepwise refinements [29], [21]. Specifically, the I/O Automata Model defines the concept of one I/O Automaton *implementing* another I/O Automaton. Therefore the correctness of complex algorithms can be proved by designing a series of algorithms of increasing complexity. The simulation proof techniques are used to show that the complex algorithms implement the simpler ones. In this way, the complexities of an algorithm are introduced in a stepwise manner. Our use of the simple unbounded real number based timestamping specification demonstrates these techniques (see [29] for thorough discussion of these issues).

The use of the I/O Automata Model in our paper suggests several avenues of research for

I/O Automata theory. The reader will notice that the I/O Automata section is fairly long since it develops several concepts. The need to develop these concepts is due to the fact that the I/O Automata Model is much more general than the shared memory system model that is needed in this paper. Hence much of the structure of the shared memory model must be developed for the I/O Automata Model. A research effort that develops structure for specific system models such as the shared memory model and the network model would be an invaluable contribution. [15] is a good step in this direction for the shared memory model.

In recent years, much progress has been made in the area of automatic theorem provers. Large parts of our correctness proof, especially the proof for the invariants in Section 6 use an extensive, well structured case analysis. Each case is proved by a simple but tedious argument. Consequently, we view the correctness proof of our bounded timestamp algorithms as an ideal candidate with which to test the effectiveness of automatic theorem provers [6]. In testing a theorem prover on our algorithm we hope to determine wether or not I/O Automata proofs might in the future utilize theorem provers on a regular basis.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic Snapshots of Shared Memory. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1990, pp. 1–14.

[2] H. Attiya, D. Dolev, and N. Shavit. Bounded polynomial randomized consensus. In *Proceedings of the* 8th *Annual ACM Symposium on Principles of Distributed Computing*, pages, ACM SIGACT and SIGOPS, ACM, 1989.

[3] H. Attiya and M. P. Herlihy. Private Communication, 1991.

[4] J. Anderson and M. Gouda. The virtue of patience: concurrent programming with and without waiting. 1988.

[5] J. H. Anderson. *Multiple-Writer Composite Registers.* Technical Report, The University of Texas at Austin, September 1989.

[6] W. Bevier and J. Søgaard-Andersen. Mechanically Checked Proof of Kernel Specification. To appear in *Proceedings of the 3rd Workshop on Computer Aided Verification*, 1991.

[7] J. Burns and G. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages, ACM, August 1987.

[8] B. Chor, A. Israeli, and M. Li. On process coordination using asynchronous hardware. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, August 1987.

[9] R. Cori and E. Sopena. Some Combinatorial aspects of timestamp systems. Unpublished Manuscript, 1991.

[10] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications Of The ACM*, 8:165, 1965.

[11] D. Dolev and N. Shavit. Bounded concurrent time-stamps are constructible. *SIAM Journal on Computing*, to appear. Also in *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, Seattle, Washington*, pages 454–465, 1989.

[12] C. Dwork and O. Waarts. Simple and Efficient Bounded Concurrent timestamping – or – Bounded concurrent time-stamps are comprehensible. Unpublished manuscript, Stanford University, Stanford,1991.

[13] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *Proceedings of 20th FOCS*, pages, October 1979.

[14] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Distributed fifo allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114, January 1989.

[15] K. Goldman, N. Lynch, and K. Yelick Modelling Shared State in a Shared Action Model. Unpublished manuscript, Washington University, St Louis, 1992.

[16] M. P. Herlihy. Wait-free synchronization. In *ACM TOPLAS*, 13(1), pages 124–149, January 1991.

[17] A. Israeli and M. Li. Bounded time stamps. In 28th *Annual Symposium on Foundations of Computer Science, White Plains, New York*, pages 371–382, 1987.

[18] A. Israeli and M. Pinchasov. A linear time bounded concurrent timestamp scheme. Technical Report, Technion, Haifa, Israel, March 1991.

[19] L. Lamport A new solution of *Dijkstra's* concurrent programming problem. *Communications of the ACM*, 78(8):453–455, 1974.

[20] L. Lamport Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 27(7):558–565, 1978.

[21] L. Lamport Specifying concurrent modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[22] L. Lamport The mutual exclusion problem. parts I and II. *J. ACM*, 33(2):313-348, 1986.

[23] L. Lamport On interprocess communication. parts I and II. *Distributed Computing, 1*, 1 (1986) 77–101.

[24] M. Li and P. Vitanyi. A Very Simple Construction for Atomic Multiwriter Registers. Report, Aiken Computation Laboratory, Harvard University, 1987.

[25] M. Li and P. Vitanyi. Uniform construction for wait-free variables. 1988. Unpublished manuscript.

[26] M. Li and P. Vitanyi. How to Share Concurrent Asynchronous Wait-free Variables. In *Proceedings of the* 16th *International Colloquium on Automata, Languages and Programming*, pages 488–505, 1989. Unpublished manuscript.

[27] M. Lui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[28] N. Lynch and K. Goldman Distributed Alogorithms. Lecture Notes for 6.852, Fall 1988. MIT/LCS/RSS-5, Laboratory for Computer Science, MIT, 1989.

[29] N. Lynch and M. Tuttle. Hierarchical Correcntess Proofs for Distributed Algorithms. Tecnical Report MIT/LCS/TR-387, Laboratory for Computer Science, MIT, 1987.

[30] N. Lynch and F. Vaandrager. Forward and backward simulations for timing based systems. To appear in *Proceedings of REX Workshop on Real-time: theory in practice*, Mook, 1991.

[31] M. Raynal. *Algorithms for Mutual Exclusion.* North Oxford Academic, England, 1986. Originally published in French in 1984. Translated by D. Beeson.

[32] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica,* 6(1):319–340,1976.

[33] G.L. Peterson and J. Burns. Concurrent reading while writing ii: the multi-writer case. In 28th *Annual Symposium on Foundations of Computer Science, White Plains, New York*, pages, May 1987.

[34] P. Vitanyi and B. Awerbuch. Shared register access by asynchronous hardware. In 27th *Symposium on the Foundations of Computer Science*, 1986.

[35] M. Saks and F. Zaharoglou. Optimal Space Distributed Move-to-front Lists. In *Proceedings of the* 10th *Symposium on the Principals of Distributed Computing*, pages 65-73, Montreal, 1991.

[36] R. Schaffer. On the correctness of atomic multi-writer registers. 1988. Bachelor's Thesis, June 1988, Massachusetts Institute Technology. Also, Technical Memo MIT/LCS/TM-364.

[37] N. Shavit. Concurrent time stamping. Ph.D. Thesis, Hebrew University, 1989.

[38] W. Wiehl. Specification and Implementation of Atomic Data Types. Ph.D. Thesis, Technical Report MIT/LCS/TR-314, MIT Laboratory for Computer Science, MIT, 1984.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1992	

4. TITLE AND SUBTITLE

Consurrent Timestamping Made Simple

5. FUNDING NUMBERS

6. AUTHOR(S)

Gawlick, R.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

MIT, Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

MIT/LCS/TR-556

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

N00014-89-J-1988

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Concurrent Timestamp Systems (CTSS) allow processes to temporally order concurrent events in an asynchronous shared memory system. Bounded memory constructions of a CTSS are extremely powerful tools for concurrency control, and are the basis for solutions to many coordination problems including mutual exclusion, randomized consensus, and multiwriter multireader atomic registers. Unfortunately, known bounded CTSS constructions seem to be complex from the algorithmic point of view.
Because of the importance of bounded , CTSS the rather involved original construction by Dolev and Shavit was followed by a series of papers that tried to provide more easily verifiable CTSS constructions.

In this paper, we present what we believe is the simplest, most modular, and most easily proven bounded CTSS algorithm known to date. The algorithm is constructed and its correctness proven by carefully reasoned use of several tools. Our algorithm combines the labeling method of the Dolev-Shavit CTSS with the atomic snapshot algorithm proposed in Afek et. al, in a way that limits the number of interleavings that can occur. To facilitate our correctness proof, we introduce a specially tailored intermediate CTSS specification using unbounded label values taken from the positive reals. Our correctness proof first shows that the real-number based specification meets the CTSS axioms. Using the forward simulation techniques of the I/O Automata model, we then show that our bounded algorithm implements the real-number based specification. Finally, we prove that any CTSS that meets the CTSS axioms can be used to implement multireader multiwriter atomic registers and first-some-first-serve (*fcfs*) mutual exclusion.

14. SUBJECT TERMS

timestamps, concurrent algorithms, distributed systems, I/O automata

15. NUMBER OF PAGES

73

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT