Concurrent Timestamping Made Simple

Rainer Gawlick[†] Nancy Lynch[†] Nir Shavit[‡]

March 3, 1995

Abstract

Concurrent Time-stamp Systems (CTSS) allow processes to temporally order concurrent events in an asynchronous shared memory system, a powerful tool for concurrency control, serving as the basis for solutions to coordination problems such as mutual exclusion, ℓ-exclusion, randomized consensus, and multi-writer multi-reader atomic registers. Solutions to these problems all use an "unbounded number" based concurrent time-stamp system (UCTSS), a construction which is as simple to use as it is to understand. A *bounded* "black-box" replacement of UCTSS would imply equally simple bounded solutions to most of these extensively researched problems. Unfortunately, while all know applications use UCTSS, all existing solution algorithms are only proven to implement the Dolev-Shavit CTSS axioms, which have been widely criticized as "hard-to-use." While it is easy to show that a UCTSS implements the CTSS axioms, there is no proof that a system meeting the CTSS axioms implements UCTSS. Thus, the problem of constructing a *bounded* black-box replacement for *uctss* remains open.

This paper presents the first such bounded black-box replacement of UCTSS. The key to the solution is a simplified variant of the Dolev-Shavit CTSS algorithm based on the atomic snapshot object proposed by Afek et. al. and Anderson, in a way that limits the number of interleavings that can occur, and whose behaviours can be readily mapped to those of UCTSS. Using the forward simulation techniques of the I/O Automata model, we are then able show that our bounded algorithm behaves like UCTSS. The forward simulation allows us to present, what would otherwise be a complicated proof, as an extensive, yet at each step simple case analysis. In fact, we believe that large parts of the forward simulation proof can be checked using an automatic proof checker such as Larch.

For read/write memory, our easy to use bounded UCTSS is only a logaritmic factor from the most efficient known bounded CTSS constructions. Moreover, unlike these efficient algorithms, our modular use of an atomic snapshot object implies that our constructions are not limited to read/write memory, and can be applied in any computation model whose basic operations suffice to provide a wait-free snapshot implementation. The complexity of our bounded UCTSS will be the same as the complexity of the underlaying snapshot implementation used.

---

[*] A preliminary version of this work appeared in the *Proceedings of the Annual Israel Symposium on Theory and Practice of Computing*, Haifa, May 1992, and as Technical Report MIT/LCS/TR-556.

[†] Laboratory for Computer Science, MIT. 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.

[‡] Contact Author: MIT and Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel. E-mail: shanir@theory.lcs.mit.edu

1 Introduction

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, need only scan through all the numbers and observe the 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. A CTSS is required to be *waitfree*, which means that a process is guaranteed to finish any of the two above mentioned label-taking or scanning 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 (Herlihy, see [13]).

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 algorithms include Lamport's *first come first served* mutual exclusion [17], Vitanyi and Awerbuch's construction of a multi-reader multi-writer atomic register [25], Abrahamson's randomized consensus [1], and Afek, Dolev, Gafni, Merritt, and Shavit's *first come first enabled* ℓ-exclusion [3]. Solutions to these problems all use an "unbounded number" based concurrent timestamp system (UCTSS), a construction which is as simple to use as it is to understand. A bounded "black-box" replacement of UCTSS would imply equally simple bounded solutions to these extensively researched problems.

1.1 Related Research

Israeli and Li, in [15], 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 [21], Cori and Sopena [8] and Saks and Zaharoglou [26]. Dolev and Shavit [9] were the first to provide an axiomatic definition and a construction of a bounded *concurrent* timestamp system using read/write registers. 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 more efficient and simple to understand bounded CTSS algorithms. Israeli and Pinchasov [16] have simplified the [9] algorithm by modifying the labeling scheme of [9], introducing a new label scanning method, and replacing the ordering-of-events based formal proof [18] of the CTSS axioms. Concurrent with our work, Dwork and Waarts [10] presented the most efficient read/write register based CTSS construction to date, taking only $O(n)$ time for either a scan or update. They model their bounded construction after a new type of unbounded CTSS construction, where processes choose from "local pools" of label values instead of the simple "global pool" based UCTSS [1, 3, 17, 25]. In order to bound the number of possible label values in the local pool of the bounded implementation, they introduce a form of amortized garbage collection. They then prove that the linear time bounded implementation meets the CTSS axioms of [9]. In [11], Dwork, Herlihy, Plotkin, and Waarts introduce an alternative linear complexity bounded CTSS construction that combines a *time-lapse snapshot* with the bounded CTSS algorithm of [9]. The proof of their algorithm leverages the axiomatic proof in [9] by arguing that the executions of

their algorithm are a subset of the executions of the algorithm in [11].

We observe that all known applications use UCTSS, but all existing bounded solutions are only proven to implement the CTSS axioms, which have been widely criticized as "hard-to-use." It is easy to show that a UCTSS implements the CTSS axioms, yet there is no proof that a system meeting the CTSS axioms implements UCTSS. Thus, the problem of constructing a *bounded* black-box replacement for *uctss* remains open.

1.2 Our Results

This paper presents the first *bounded* concurrent time-stamp system algorithm (BCTSS) that provably implements UCTSS. The key to the solution is an algorithm whose behaviours can be readily mapped to those of UCTSS, and the use of a forward simulation proof technique. Our solution consists of the following steps.

- Our algorithm is a variation on the Dolev-Shavit algorithm [9] based on the use of the atomic snapshot primitive introduced by Afek et. al [2] and Anderson [6]. A snapshot primitive allows a process P_i to UPDATE the ith memory location, or SNAP the memory, that is, collect an "instantaneous" view of all n shared memory locations. By using a snapshot primitive, we limit the number of interleavings that can occur, and are able to introduce a much simplified version of the labeling algorithm of [9] that is tailored so as to allow a forward simulation proof [22]. Moreover, as we elaborate later, our algorithm is no longer limited to read/write memory, and holds in any computation model whose basic operations suffice to provide a wait-free snapshot implementation.

- Our proof that the bounded algorithm satisfies the UCTSS specification requires a different technique than any of the operational proofs of the CTSS axioms as found in the literature. We begin by introducing a UCTSS specification that uses, instead of integers, label values taken from the unbounded positive reals. This system is broader than (i.e. strictly includes) the integer based UCTSS used in actual applications, yet surprisingly allows for a simpler proof. We then use the forward simulation techniques of the I/O Automata model of Lynch and Tuttle [22], to show that our bounded algorithm implements the real-number based UCTSS specification. (See [23] for references and a discussion of forward simulation techniques.) The forward simulation techniques allow us to present, what would otherwise be a complicated proof, as an extensive, yet at each step simple case analysis. In fact, we believe that large parts of the forward simulation proof can be checked using an automatic proof checker such as Larch [27, 12].

As mentioned, our algorithm provides a wait-free solution in whatever computation model the atomic snapshot object [2] is implemented, be it single-writer multi-reader registers [4], multi-reader multi-writer registers [14], consensus objects [7], or memory with hardware supported *compare-and-swap* and *fetch-and-add* primitives.

The time complexity of our UCTSS construction is simply the complexity of the underlying atomic snapshot implementation. For single-writer multi-reader memory, this is the intricate Attiya and Rachman algorithm [4], which takes $O(n \log n)$ operations for either a scan or an update of memory.

Hence the complexity of our algorithm is $O(n \log n)$ for each operation, a logarithmic factor away from the best know constructions [10, 11, 16].

One can also use snapshot implementations in other computation models: the snapshot algorithm of Inoue, Chen, Masuzawa and Tokura [14] to get an $O(n)$ per operation bounded UCTSS using multi-reader multi-writer registers, the Chandra and Dwork [7] algorithm to get $O(1)$ label and $O(n)$ scan operations using consensus or randomized consensus objects, or the Riany, Shavit and Touitou [24] algorithm to get $O(1)$ label and $O(n)$ scan operations using memory with hardware supported *compare-and-swap* and *fetch-and-add* primitives. Note that the most efficient known shared-memory CTSS algorithms [10, 11, 16] do not readily imply efficient algorithms in other computation models. Moreover, stronger computational models do not seem to immediately imply significant simplifications of bounded CTSS algorithms.

The paper is organized as follows. Section 2 presents the I/O Automaton model. Our unbounded CTSS is introduced in Section 3, and the bounded CTSS is introduced in Section 4. Section 5 introduces several key invariants that are needed for the simulation proof of Section 6. Some of the invariant proofs are postponed until Section 7.

2 The I/O Automata Model

We present our algorithm in the context of the I/O Automata model. This model, introduced by Lynch and Tuttle [22], 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).

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 [22].

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 $(a, b) \in$* R.
2. *Suppose a is a reachable state of A and b is a reachable state of B such that $(a, b) \in$* 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 $(a', b') \in$* R.
3. *If action π is enabled in state b of B and $(a, b) \in$* R *then action π is enabled in state a of A.*

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

In the I/O Automaton model, actions provide the basic communications mechanism. CTSS algorithms, as described in the introduction communicate using shared registers. We can encode register based communication in the I/O Automatom model, with allowable operations ranging from single-writer multi-reader to powerful *read-modify-write*, by encoding the register values in the state of the Automaton. Actions are then used to modify the state.

3 An Unbounded Concurrent Timestamp System

This section introduces our unbounded implementation of a concurrent timestamp system, UCTSS. It is a generalization of the traditional unbounded number UCTSS. In particular, it uses timestamps from $\Re^+$ instead of the natural numbers. The code for the operations of UCTSS is presented in two forms. Figure 1 uses psuedocode. Figure 2 presents the code in the precondition-effect notation commonly used to describe I/O Automata[1]. We use the precondition-effect notation as the basis for our definitions and correctness proof and include the compact and intuitive pseudocode 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. This function actually picks the new label. In particular, it non-deterministically picks any real

[1] BCTSS is the name for our bounded 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 4.

```
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 1: Psuedocode for UCTSS and BCTSS

number bigger than the largest current label. (Note that the traditional UCTSS implementations pick $X = 1$ or X as a non-zero natural number.) To determine the largest current label, we use the SNAP_i operation. The SNAP_i operation, defined by Afek et al. [2] and Anderson [6], atomically reads an array of single writer multireader locations. Here we use it to atomically read the current labels for the NEWLABEL_i function. To write the new label determined by the NEWLABEL_i function we use the UPDATE_i operation. A UPDATE_i operation, also defined by [2], writes a value to a single location in the array of single writer multireader location read by SNAP_i. SNAP_i and UPDATE_i are waitfree, therefore their use does not compromise the waitfree properties of our timestamp algorithm. A rigorous theoretical foundation for this claim can be found in [**?**].

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 2. 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 has input actions $\text{BEGINLABEL}_i(val_i)$ and BEGINSCAN_i for $i \in \{1 \ldots n\}$. The output actions are ENDLABEL_i and $\text{ENDSCAN}_i(\bar{o}_i, \bar{v}_i)$. 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. We note here that we are not concerned with the implementation details of the atomic SNAP_i and UPDATE_i actions. We use them has a black box.

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

LABEL_i:

BEGINLABEL_i	*Eff:*	$op_i \leftarrow \text{LABEL}_i$
		$pc_i \leftarrow \text{SNAP}_i(\bar{t}_i, \bar{v}_i)$
$\text{UPDATE}_i((t_i, v_i), (nt_i, val_i))$	*Pre:*	$pc_i = \text{UPDATE}_i((t_i, v_i), (nt_i, val_i))$
	Eff:	$pc_i \leftarrow \text{ENDLABEL}_i$
ENDLABEL_i	*Pre:*	$pc_i = \text{ENDLABEL}_i$
	Eff:	$pc_i \leftarrow \text{NIL}$

Figure 2: Precondition-Effect code for UCTSS and BCTSS

```
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

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

4 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 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 h^{th} 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 4.1 ($\prec$ 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 4.1 $4 \ldots 4.5.2 \prec 4 \ldots 4.3.1$

The intuition behind the label set $\mathcal{A}^{n-1}$ and the order $\prec$ can be best understood by examining the inadequacies of using a bounded set of natural numbers as the label set. Consider the natural numbers from 0 to n with the usual $<$ ordering. As with UCTSS, ties between processes are broken based on process index. The obvious problem with this label set is deciding what happens when

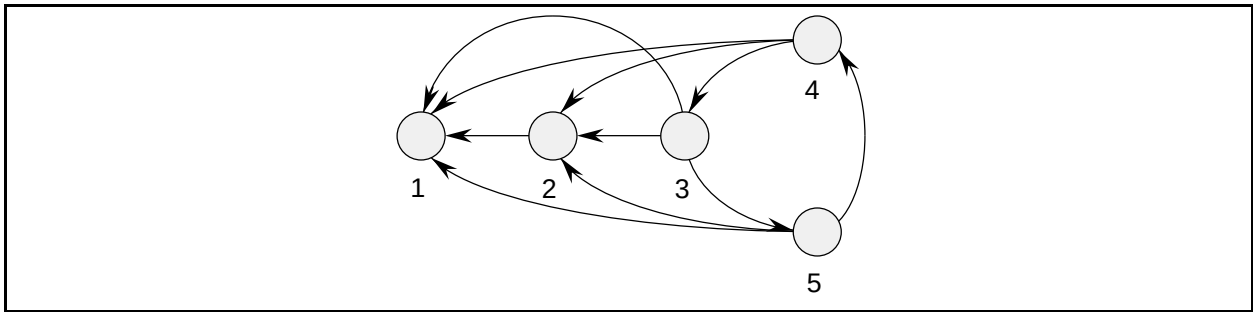

Figure 4: A graphical illustration of the $\prec_{\mathcal{A}}$ order between the elements of $\mathcal{A} = \{1 \ldots 5\}$

some processes has the label n and another process needs a new, bigger label. The obvious solution is to wrap around and use the number 0. Then the ordering among labels would be the usual $<$ ordering with the additional feature that $n < 0$.

Using a wrap around strategy provides a good solution for two processes. In particular consider the processes p_1 and p_2 with the label set $\{0, 1, 2\}$. It is easy to see that this works since there will always be an extra number between the labels of p_1 and p_2 to make sure that they are totally ordered. However, the wrap around strategy does not work for three processes. Consider the following situation for three processes. Let each process p_i have label i. Also assume that process p_2 with label 2 wants a new label that is bigger than the label 3 of p_3. Using our wrap around strategy, that label would be 0. However, now process p_2's label is ordered below that of p_1 which has a label 1, which violates the ordering properties of a timestamp system since p_1's current label was acquired before p_2 acquired the label 0. One solution might be to extend the set of numbers from which the labels are chose so that the wrap around happens later. However, it is easy to see that this will not help. In particular, processes p_2 and p_3 can ask for new labels alternately until one of them reaches the highest label. The first process to wrap around will encounter the same problem we just identified for $n = 3$. What is needed is the ability to create a cycle of numbers for processes p_2 and p_3 such that all numbers in that cycle are ordered above zero.

Now consider how $\mathcal{A}^2$ would create such a cycle. First note that the label set A contains a size three cycle, (using the numbers 3, 4, 5). We saw above from the two processor example that a size three cycles can accommodate two processes. The labels in $\mathcal{A}^2$ that share the same first digit each form a size three cycle, that can accommodate two processes. Now consider the situation where p_1 starts out with label 3.3, p_2 starts out with label 3.4 and p_3 starts out with 4.3. Using $\prec$ these labels are ordered in order of the process indexes. Now let p_2 and p_3 alternate picking new labels. They can use the size three cycles defined by the labels in $\mathcal{A}^2$ that start with the prefix 4. All labels in that cycles are ordered above p_1's label of 3.3. If p_0 now becomes active, it can jump to 5.3 which will be higher that the two labels used by p_2 and p_3.

The label set $\mathcal{A}^{n-1}$ with the order $\prec$ is the generalization of these ideas to n processes. In particular, any set of label from $\mathcal{A}^{n-1}$ which agree on the first h digits form a size 3^{n-1-h} cycle, which consists of three ordered size $3^{n-1-h-1}$ cycles, each of which in turn consist of three ordered $3^{n-1-h-2}$ cycles, etc. Such a set of labels can accommodate $n - h$ processes in any sequential execution. The cycle system construction is due to Israeli and Li [15].

This discussion has not identified a need for the numbers 1 and 2 in the set $\mathcal{A}$. These numbers are the key to making $\mathcal{A}^{n-1}$ work in concurrent executions. They are needed to deal with the fact that we use atomic snapshot objects as the underlying communications model. More powerful snapshot primitives that can atomically (i.e. in a single atomic step) read all of the labels *and* write one label will eliminate the need for these numbers (note that constructing an efficient implementation of such an object, even if one is given a *compare-and-swap* or other powerful operation, is not an obvious task). We will make our intuitive justification of the $\mathcal{A}^{n-1}$ label set with the order $\prec$ more concete with some specific examples once we have introduced some additional notation.

The following lemma shows that any two labels in $\mathcal{A}$ are always totally ordered by the $\prec$ relation.

Lemma 4.1 *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 $\prec_{\mathcal{A}}$. ■

Next we define some notation and functions for BCTSS labels. Before giving the formal definitions we give intuitive definitions. Two labels are h-equivalent, $\stackrel{h}{=}$, if their first h digits are the same. The function NEXTLABEL(ℓ, h) picks the label that is the same as ℓ for the first $h-1$ digits, changes the h^{th} digit based on the NEXT function, and sets the remaining digits to 1. The NEXTLABEL function is used to pick new labels during the LABEL operation. Finally, the set CYCLE(ℓ, h) consists of the labels that are $h-1$-equivalent to ℓ and have the h^{th} digit equal to 3,4,5.

Definition 4.2 ($\stackrel{h}{=}$ equivalence relation) For any $h \in \{0 \ldots n-1\}$, $\ell_1 \stackrel{h}{=} \ell_2$ 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 4.3 (NEXTLABEL) For any $h \in \{1 \ldots n-1\}$, $\ell' =$ NEXTLABEL(ℓ, h) iff $\ell' \stackrel{h-1}{=} \ell$, $\ell'[h] =$ NEXT$(\ell[h])$ and $\ell'[h'] = 1$ for all $h' \in \{h+1 \ldots n-1\}$. ■

Definition 4.4 (CYCLE) For any $h \in \{1 \ldots n-1\}$, $\ell' \in$ CYCLE(ℓ, h) iff $\ell' \stackrel{h-1}{=} \ell$ and $\ell'[h] \in \{3, 4, 5\}$. ■

The following Lemma gives a neccessary and sufficient condition for three label in $\mathcal{A}^{n-1}$ to form a cycle under the $\prec$ ordering.

Lemma 4.2 *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 $h \in \{1 \ldots n-1\}$ 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 4.1. 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 4.1 it cannot be that $\ell_1 = \ell_3$, therefore $\ell_3 \prec \ell_1$. Since $\prec$ is a lexicographical order, there must exist $h \in \{1 \ldots n-1\}$ 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. The set AGREE$(b.\ell, h)$ is the set of process indexes j such that the t_j label in state b that is h-equivalent to the label $b.\ell$. NUM$(b.\ell, h)$ is the cardinality of AGREE$(b.\ell, h)$. Finally, NUM$_i(b.\ell, h)$ is the cardinality of AGREE$(b.\ell, h)$ once process i is removed from AGREE$(b.\ell, h)$. We remind the reader that the following definitions are based on those in Figure 2.

Definition 4.5 (AGREE) For any $h \in \{0 \ldots n-1\}$, AGREE$(b.\ell, h) = \{j |\ b.t_j \stackrel{h}{=} b.\ell\}$. ■

Definition 4.6 (NUM) For any $h \in \{0 \ldots n-1\}$, NUM$(b.\ell, h) = |$AGREE$(b.\ell, h)|$. ■

Definition 4.7 (NUM$_i$) For any $h \in \{0 \ldots n-1\}$, NUM$_i(b.\ell, h) = |$AGREE$(b.\ell, h) - \{i\}|$. ■

Definition 4.8 (choice vector) A *choice vector* for state b is any vector $(b.\ell_1 \ldots b.\ell_n)$ such that $b.\ell_i \in \{b.t_i, b.nt_i\}$ for each i. ■

Definition 4.9 (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 4.10 ($\ll$ 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 5 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 h^{th} digit. Then the new label is the same as t_{max} for the first $h-1$ digits, it differs from t_{max} at the h^{th} 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 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 NUM$_i(t_{max}, n-1) \geq 1$ when $i \neq i_{max}$, hence FULL$_i(n-1) =$ *true*.

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

We now return to our intuitive justification of the $\mathcal{A}^{n-1}$ label set with the order $\prec$. Specifically we will strengthen our intuition using some examples. The examples will show how the following "invariants" are maintained:

```
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

(1) For any reachable state b and any $h \in \{1 \ldots n-1\}$ consider any set of labels that agree on the first $h-1$ digits. Then it will not be the case that h^{th} digits of that set of labels includes the numbers 3, 4 and 5. More formally, for any $i \in \{1 \ldots n\}$: $\{b.t_j[h] | b.t_j \in \text{CYCLE}(b.t_i, h)\} \neq \{3,4,5\}$.

(2) For any reachable state b and any $h \in \{1 \ldots n-1\}$ consider any set of labels that agree on the first $h-1$ digits and have the h^{th} digit in the set $\{3,4,5\}$. The cardinality of that set of labels is at most $n-h+1$. More formally, for any $i \in \{1 \ldots n\}$: $|\text{CYCLE}(b.t_i, h)| \leq n-h+1$.

Maintaining the second invariant is the key to maintaining the first, and the first implies that TOT$(b) = true$ when the choice vector is restricted to the t–labels. While the invariants provide good intuiation about the correctness of the algorithm, it turns out that they are too general to be used the induction proof, and Theorem 5.1 must use a set of more refined inductive statments to capture the details of the possible concurrent behaviors.

The manner in which the invariants (1) and (2) are preserved, is explained via several examples. The first example considers a serial setting, while the second example considers a concurrent setting. The concurrent setting will illustrate the need for the numbers 1 and 2 in the $\mathcal{A}$ set.

For simplicity, the examples consider the case where $n = 3$. Thus the labels will be taken from A^2. For both examples, processes p_1, p_2, p_3 start out with labels $t_1 = 3.4$, $t_2 = 3.5$, and $t_3 = 4.1$. It is easy to see that the labels are totally ordered by the $\prec$ ordering.

Example 4.2 Assume that the following sequence of labeling operation are executed sequentially. Process p_1 performs a label operations that reads t_1, t_2 and t_3, and picks the new label $t_1' = 4.2$ based on NEWLABEL$(\overline{t_1})$. Process p_3 performs a label operations that reads the new label t_1'. It thus picks a label t_3' with first digit 4, following the rule that the node chosen should be the "lowest node dominating all other nodes with labels." This is actually the most basic rule implied by the definition of NEWLABEL.

Processes p_1 and p_2 can continue forever to choose $t_1'' = 4.4$, $t_3'' = 4.5$, $t_1''' = 4.3$... (that is, pick labels first digit 4 and second digit taken from $\{3,4,5\}$), maintaining the above invariants, because the set of labels with first digit 4 and second digit taken from $\{3,4,5\}$ represent a size three cycle that can accomadate two processes.

If at some point p_2 picks a new label, L'_2, it will read the labels of both p_1 and p_3 has having the first digit 4. Any set of labels in A^2 that have the first digit in common can accomdate at most two processes. In particular, p_2's label operation will find that $\text{NUM}_2(t_{max}) = 2$. Since $\text{FULL}_2(2) = \textit{true}$ p_2 will pick $t'_2 = 5.1$, and so on... ■

From Example 4.2 the reader can see that given that the second "invariant" is maintained, only two processes have labels from the cycle $\{4.3, 4.4, 4.5\}$, the first "invariant" is readily maintained. The basis for guaranteeing the second "invariant," is that the algorithm is structured so that node 1 on an arbitrary level h has at least $n - h + 1$ processes agree on it before any process can choose node 2. This implies that processes move to cycle nodes only from nodes 1 and 2. If there were by way of contradiction an $(n - h + 2)$-nd process p_i moving to a cycle node, say to a node 3, it would have had to see a maximal label on node 2, which in turn means node 1 would have had at least $n - h + 1$ labels when p_i performed its SNAP, so p_i would have detected $n - h + 2$ labels that agree with it on level $h - 1$ and would change its $h - 1$ level label and not choose a cycle label on level h. The following is an example of the role of nodes 1 and 2.

Example 4.3 Let processes p_2 and p_1 begin performing labeling operations concurrently, reading t_2, t_1 and t_3 and computing NEWLABEL, such that $nt_2 = nt_1 = 4.2$. If they then complete their operations by writing their labels (i.e., do the UPDATE operation), they will choose the same label. Their labels can be ordered using their processor *ids*. If either process performed a subsequent label operation it would choose the label 5.1 since $\text{NUM}_2(t_{max}, 1) = \text{NUM}_1(t_{max}, 1) = 2$. Thus, neither process would choose a label in the cycle defined by the labels $4.3, 4.4, 4.5$.

Now change the above scenario so that both processes do lot complete their label operations. In particular, suppose that p_2 is stalled just before writing $nt_2 = 4.2$ to the shared variable t_2 (using the UPDATE operation), while p_1 writes $t'_1 = nt_1 = 4.2$. Now let process p_3 perform a label operation that reads the new label $t'_1 = 4.2$ and the old label $t_2 = 3.5$, thus picking $t'_3 = 4.3$. If processes p_1 and p_2 continue to pick new labels, they will pick them from the cycle defined by the labels $4.3, 4.4, 4.5$, since they continue to read p_2's old label. At some point let p_2 complete its label operation writing $t'_2 = nt_2 = 4.2$. Now there are three labels $\stackrel{1}{=}$ to 4 (two of them in $\text{CYCLE}(t'_2, 2)$. However, if p_2 now performs a subsequent labeling operation, it will read the labels of both p_1 and p_3 as being $\stackrel{1}{=}$ to 4. Since $\text{NUM}_2(t_{max}, 1) = 2$, $\text{FULL}_2(1) = \textit{true}$, so p_2 will pick the new label $t''_2 = 5.1$, not picking a label represented by a node in the cycle defined by the labels $4.3, 4.4, 4.5$.

If the labels were not structured to include the values 1 and 2, which are not part of the cycle generated by the values 3, 4, and 5, then a process would always have a label that is represented by a node in a cycle. The reader can verify that the sequence of operations in this example, would cause the labels of p_1,p_2 and p_3 to end up each on a different nodes of the cycle defined by the labels $4.3, 4.4, 4.5$, contradicting invariant (1). ■

5 Invariants

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

Theorem 5.1 *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* $\text{NUM}(b.t_{max}, h-1) \geq n-h+1$.

■

The following is an intuitive explanation of the inductive claims of Theorem 5.1 that hold for any system state.

I: The set of labels that for each process includes either its current label or its newly chosen label (yet unwritten) is totally ordered. This implies the first "invariant," but is slightly stronger in the sense that it tracks the behavior of yet unwritten labels.

II: The process with the maximal label does not choose a new one.

III: The NEXTLABEL function defines a label's successor for a given digit. This invariant states that a process' new (not yet updated) label must be a successor of the maximum at some digit level h, even though the state (including the maximum) may have changed since the new label was picked.

IV: Consider a process p_i that has chosen a new label which is no longer the maximum (due to subsequent new labels chosen by others) but has not yet updated the shared state. Then, for all processes whose new labels were chosen based on p_i's current label, their choice would have been the same had they used p_i's new label. This is due to the fact that updating the shared state does not affect the NUM count.

V: If a process chooses a new label in the cycle at digit h, (i.e., $\in \{3,4,5\}$), then it must be that its new label is the same as its current label for the first $h-1$ digits. This implies that a process must choose either 1 or 2 before it can choose digits $\{3,4,5\}$.

VI: For any new label,

a: if it is a successor of the max at digit h, then the number of labels that agree with max on the first h digits is the maximum that can be accomodated by that h-digit prefix. In terms of the algorithm this means that a process uses a new prefix only when necessary.

b: Once some process has chosen an h-th level digit that is greater than 1 (i.e., $\{2,3,4,5\}$), then the number of labels that agree with it on the $h-1$ prefix must be at least the maximum that can be accomodated by that prefix.

Invariants 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 $nt_i = \text{NEXTLABEL}(t_{max}, h)$. 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 $\text{NUM}(t_{max}, h-1) \geq n-h+1$. The code for NEWLABEL_i insures that $\text{NUM}_i(t_{max}, h-1) < n-h+1$. 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 5.1 uses induction. It depends on a series of claims, one for the initial state and one for each action in the inductive step. Most of these claims appear in the final section since they use a fairly straightforward, if tedious, case analysis. This section presents the key claims associated with invariant I. In the following claims assume that state b transitions to b' using the $\text{SNAP}_k(\bar{t_k}, \bar{v_k})$ action.

Claim 5.1.1 *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 5.1.2 *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 5.1.3 *If $k \neq b.i_{max}$ then I is true in b'.*

Proof: For a contradiction assume that $\text{TOT}(b') = \textit{false}$. Then there must exist a choice vector C whose values are not totally ordered. By Lemma 4.2, 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 4.2 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 5.1.2 $\text{NUM}_k(b.t_{max}, h'-1) = \text{NUM}_k(b.t_{max}, h')$. 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 5.1.2, $\text{NUM}_k(b.t_{max}, h'-1) = \text{NUM}_k(b.t_{max}, h')$. 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, $\text{TOT}(b') = \mathit{true}$. ■

Proof: (For Theorem 5.1) We proceed by induction on the length of the execution ending in the reachable state b. The base case is established by Lemma 7.1. 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 7.2. If $\pi = \text{UPDATE}_k((t_k, v_k), (nt_k, val_k))$, the induction step follows from Lemma 7.3. If $\pi = \text{SNAP}_k(\bar{t_k}, \bar{v_k})$, the induction step follows from Lemma 7.4. ■

6 The Simulation Proof

In this section we use Theorem 2.1 to show that *fairbehs*(BCTSS) $\subseteq$ *fairbehs*(UCTSS). This implies that BCTSS implements UCTSS. 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 6.1 (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 6.1 *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 $op_i = pc_i = \text{NIL}$ 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 6.2 *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 $(b, u) \in$ 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 2 shows that $b'.op_k = u'.op_k$ and $b'.pc_k = u'.pc_k$. This suffices to shows that $(b', u') \in$ 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 $(b', u') \in$ R.

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

Since $(b, u) \in$ 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 $(b, u) \in$ 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 2 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 $(b', u') \in$ R.

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

Since $(b, u) \in$ 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 $(b, u) \in$ 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 2 shows $b'.pc_k = u'.pc_k = \text{UPDATE}_k((t_k, v_k), (nt_k, val_k))$. This suffices to show that $(b', u') \in$ R.

So assume that $k \neq b.i_{max}$ for the remainder of the proof of this case. Since $(b, u) \in$ 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 $(b', u') \in \text{R}$. Only nt_k and pc_k change as a result of the action. Figure 2 shows $b'.pc_k = u'.pc_k = \text{UPDATE}_k((t_k, v_k), (nt_k, val_k))$. 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 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 $S = \{j \mid j \neq k \text{ and } b.t_{max} \prec b.nt_j\}$. Define $T = \{j \mid j \neq k \text{ and } b.t_{max} \succeq b.nt_j\}$. Similarly, define $S_u = \{j \mid j \neq k \text{ and } u.t_{max} < u.nt_j\}$. Define $T_u = \{j \mid j \neq k \text{ and } u.t_{max} \geq u.nt_j\}$. 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 $(b, u) \in$ 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 $(b, u) \in$ R, part 4 of R shows that $b.val_k = u.val_k$. Thus, $b'.v_k = u'.v_k$. Figure 2 shows $b'.pc_k = u'.pc_k = \text{ENDLABEL}_k$. Consequently, $(b', u') \in$ R if we can show that part 2 of R holds for states b' and u'. For part 2 of 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 $(b, u) \in$ 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 UCTSS.

Theorem 6.3 BCTSS *implements* UCTSS.

Proof: By definition of BCTSS and UCTSS, $sig(\text{BCTSS}) = sig(\text{UCTSS})$ and $part(\text{BCTSS}) = part(\text{UCTSS})$. Lemma 6.1, and Lemma 6.2 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 $fairbehs(\text{BCTSS}) \subseteq fairbehs(\text{UCTSS})$. Thus BCTSS implements UCTSS. ■

7 Claims for Proof of Theorem 5.1

This section contains the bulk of the claims needed for Theorem 5.1.

Lemma 7.1 *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 7.2 *Let b be a state of* BCTSS *that satisfies I - VI. If (b, π, b') is a step of* BCTSS *where $\pi \in \{\text{BEGINSCAN}_k, \text{ENDSCAN}_k(\bar{o_k}, \bar{v_k}), \text{BEGINLABEL}_k(val_k), \text{ENDLABEL}_k\}$ 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 7.3 *Let b be a state of* BCTSS *satisfying I - VI. If $(b, \text{UPDATE}_k((t_k, v_k), (nt_k, val_k)), b')$ 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 7.3.4 *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 7.3.5 *If $k \neq b.i_{max}$ then I is true in b'.*

Proof: Assume for a contradiction that $\text{TOT}(b') = false$. Since $\text{TOT}(b) = true$ 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 $\text{TOT}(b) = true$. ■

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 7.3.6 *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 7.3.7 *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 7.3.6, 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 7.3.8 *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 7.3.6, $i = k$ 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 7.3.7, $\text{NUM}(b'.t_{max}, h) \geq \text{NUM}(b.t_{max}, h)$ 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 7.3.8 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 7.3.9 *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 7.3.10 *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 7.3.9 $b'.t_{max} = b'.t_k$. 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 7.3.10. Parts 1-5 of Claim 7.3.10 will be used throughout the remaining claims.

Claim 7.3.11 *If $k \neq b.i_{max}$ and $b.t_{max} \prec b.nt_k$ then II is true in b'.*

Proof: By Claim 7.3.9 $b'.i_{max} = k$. Part 1 of Claim 7.3.10 shows that $b'.t_k = b'.nt_k$. ■

Claim 7.3.12 *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 7.3.10, $b'.t_{max} = b'.nt_k$ 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 7.3.10 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 4.1. 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 $\text{TOT}(b) = \textit{true}$, 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 7.3.13 *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 7.3.10, $b'.nt_k = b'.t_{max}$, which directly implies IV. Now consider $i \neq k$ and any h:

$h < h'$: Part 1 of Claim 7.3.10 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 4.1, $b.t_{max} \prec b.nt_i$. 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 7.3.10 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 7.3.10 shows that $b'.t_i \stackrel{h}{\neq} b'.t_{max}$. Hence, IV is vacuously true in b'.

■

Claim 7.3.14 *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 $b'.nt_i \stackrel{h-1}{=} b'.t_{max}$. We consider two cases:

$h \leq h'$: First consider $i \neq k$. Part 1 of Claim 7.3.10 and the definition of NEXTLABEL show that $b'.t_{max} \stackrel{h-1}{=} b.t_{max}$. 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 7.3.10 shows that $b'.t_i = b'.t_{max}$. This suffices to show V.

$h > h'$: Since $b'.nt_i \stackrel{h-1}{=} b'.t_{max}$ and $h > h'$, it follows that $b'.nt_i \stackrel{h'}{=} b'.t_{max}$. Thus part 3 of Claim 7.3.10 implies that $b'.nt_i = b'.t_{max}$. By part 2 of Claim 7.3.10, $b'.t_{max}[h] = 1$. 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 7.3.15 *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 7.3.10 and the definition of NEXTLABEL show that $b'.t_{max} \stackrel{h}{=} b.t_{max}$. 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 7.3.10 $\text{NUM}(b'.t_{max}, h-1) \geq \text{NUM}(b.t_{max}, h-1)$. 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 7.3.10 $\text{NUM}(b'.t_{max}, h-1) \geq \text{NUM}(b.t_{max}, h-1)$. 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 7.3.10 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 7.3.10 $\text{NUM}_k(b'.t_{max}, h-1) \geq \text{NUM}_k(b.t_{max}, h-1)$. 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 7.3.10 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 7.3.16 *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 7.3.10 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 7.3.10 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'.

$h = h'$[2]: Using part 1 of Claim 7.3.10 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 7.3.16 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 7.3.10 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 7.3.4 shows that b' satisfies I - VI when $k = b.i_{max}$. When $k \neq b.i_{max}$ Claim 7.3.5 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 7.3.8 shows that II - VI hold when $b.nt_k \preceq b.t_{max}$. Claim 7.3.11, Claim 7.3.12, Claim 7.3.13, Claim 7.3.14, Claim 7.3.15 and Claim 7.3.16 each consider one of the invariants to show that II - VI hold when $b.t_{max} \prec b.nt_k$. ■

[2] 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.

Lemma 7.4 *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 7.4.17 *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 5.1.2 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 5.1.2 implies that $\text{NUM}_i(b'.t_{max}, h'-1) = n - h'$.

■

We can now complete the proof of the lemma. Claim 5.1.1 shows that I - VI hold for b' when $k = b.i_{max}$. When $k \neq b.i_{max}$, Claim 5.1.3 shows that I holds in b' and Claim 7.4.17 shows that II - VI hold for b'. ■

8 Discussion and Future Work

All know applications of timestamp systems use UCTSS. This paper provides a bounded implementation of UCTSS, so the correctness proofs of the applications using timestamp systems can assume that they are using a UCTSS even though the actual implementation would make use of our bounded BCTSS. The time complexity of our BCTSS construction is simply the complexity of the underlying atomic snapshot implementation.

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 5 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 [27]. In testing a theorem prover on our algorithm we hope to determine whether or not I/O Automata proofs might in the future utilize theorem provers on a regular basis.

References

[1] K. Abrahamson. On achieving consensus using a shared memory. In *Proceedings of 7th ACM Symposium on the Principles of Distributed Computing*, Toronto, Ontario, Canada, August 1988.

[2] 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.

[3] Y. Afek, D. Dolev, E. Gafni, M. Merritt and N. Shavit. A Bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM TOPLAS*, (16)3, pages 939–953, May 1994.

[4] Attiya, H., and Rachman, O. Atomic snapshots in O(nlogn) operations. *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing*, (Aug. 1993) 29–40.

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

[6] J. H. Anderson. Multiple-writer composite registers. Distributed Computing, Vol. 7, No. 4, pages 175–, 1994.

[7] Chandra T. D. and Dwork, C. Using Consensus to solve Atomic Snapshots. Manuscript, 1993.

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

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

[10] C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible!, *ACM Symposium on Theory of Computing*, 1992.

[11] C. Dwork, M. Herlihy, S. Plotkin, and O. Waarts. Time lapse snapshots. *Proceedings of the Israel Symposium on the Theory of Computing and Systems.* Haifa, Israel, May 1992(Dolev D., Galil Z., and Rodeh M. eds.) 154–170.

[12] J. Guttag and J. Horning. Larch: Languages and tools for formal specification. Springer Verlag, 1993.

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

[14] M. Inoue, W. Chen, T. Masuzawa and N. Tokura. Linear-time Snapshot using Multi-writer Multi-reader registers. *Workshop on Distributed Algorithms*, pages 130-140, Spinger Verlag, 1994.

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

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

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

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

[19] M. Li and P. Vitanyi. A very simple construction for atomic multiwriter registers. Report, Aiken Computation Laboratory, Harvard University, 1987.

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

[21] M. Li and P. Vitanyi. How to share concurrent asynchronous wait-free variables. In *Proceedings of the* 16^{th} *International Colloquium on Automata, Languages and Programming*, pages 488–505, 1989. Unpublished manuscript.

[22] N. Lynch and M. Tuttle. Hierarchical correcntess proofs for distributed algorithms. Tecnical Report MIT/LCS/TR-387, Laboratory for Computer Science, MIT, 1987.

[23] 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.

[24] Y. Riany, N. Shavit, and D. Touito. Towards a practical snapshot algorithm. *Proceedings of the Third Israel Symposium on Theory and Computing Systems (ISTCS)*, Tel-Aviv, January 1995.

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

[26] M. Saks and F. Zaharoglou. Optimal space distributed move-to-front lists. In *Proceedings of the* 10^{th} *Symposium on the Principals of Distributed Computing*, pages 65-73, Montreal, 1991.

[27] J. Søogard-Andersen, J. Guttag, J. Garland, A. Pogosyants Encoding automata and simulation proofs in LP. Unpublished manuscript, MIT, 1992.