Finite-state concurrent programs can be expressed pairwise

Paul C. Attie

Department of Computer Science
American University of Beirut

and

Center for Advanced Mathematical Sciences
American University of Beirut

paul.attie@aub.edu.lb

February 2, 2008

Abstract

We present a pairwise normal form for finite-state shared memory concurrent programs: all variables are shared between exactly two processes, and the guards on transitions are conjunctions of conditions over this pairwise shared state. This representation has been used to efficiently (in polynomial time) synthesize and model-check correctness properties of concurrent programs. Our main result is that any finite state concurrent program can be transformed into pairwise normal form. Specifically, if $Q$ is an arbitrary finite-state shared memory concurrent program, then there exists a finite-state shared memory concurrent program $P$ expressed in pairwise normal form such that $P$ is strongly bisimilar to $Q$. Our result is constructive: we give an algorithm for producing $P$, given $Q$.

1 Introduction

The state explosion problem is recognized as a fundamental impediment to the widespread application of mechanical finite-state verification and synthesis methods, in particular, model-checking. The problem is particularly severe when considering finite-state concurrent programs, as the individual processes making up such programs may be quite different (no similarity) and may be only loosely coupled (leading to a large number of global states).

In previous work [1, 2, 5], we have suggested a method of avoiding state-explosion by expressing the synchronization and communication code for each pair of interacting processes separately from that for other (even intersecting) pairs. In particular, all shared variables are shared by exactly one pair of processes. This “pairwise normal form” enables us, for any arbitrarily large concurrent program, to model-check correctness properties for the concurrent compositions of small numbers of processes (so far 2 or 3) and then conclude that these properties also hold in the large program. If $P$ is a concurrent program consisting of $K$ processes each having $O(N)$ local states, then we can verify the deadlock freedom of $P$ in $O(K^3N^3b)$ time\(^1\) or $O(K^4N^4)$ time, using either of two conservative tests [5], and we can verify safety and liveness properties of $P$ in $O(K^2N^2)$ time [1, 2].

A key question regarding the pairwise approach is: does it give up expressive power? That is, in requiring synchronization and communication code to be expressed pairwise, do we constrain the set of concurrent programs that can be represented? In this paper, we answer this question in the negative: we show that for any concurrent program $Q$, we can (constructively) produce a concurrent program $P$ that is in pairwise normal form, and that is strongly bisimilar to $Q$.

The rest of the paper is as follows. Section 2 presents our model of concurrent computation and defines the global state transition diagram of a concurrent program. Section 3 defines pairwise normal form. Section 4 presents our main result: any finite-state concurrent program can be expressed in pairwise normal form. Section 5 discusses related work, and Section 6 concludes.

\(^1\) $b$ is the maximum branching in the local state transition relation of a single process
2 Technical Preliminaries

2.1 Model of concurrent computation

We consider finite-state shared memory concurrent programs of the form \( P = P_1 \parallel \cdots \parallel P_K \) that consist of a finite number \( n \) of fixed sequential processes \( P_1, \ldots, P_K \) running in parallel. Each \( P_i \) is a synchronization skeleton [11], that is, a directed multigraph where each node is a (local) state of \( P_i \) (also called an \( i \)-state and is labeled by a unique name \( (s_i) \), and where each arc is labeled with a guarded command [9] \( B_i \rightarrow A_i \) consisting of a guard \( B_i \) and corresponding action \( A_i \). Each node must have at least one outgoing arc, i.e., a skeleton contains no “dead ends.” With each \( P_i \), we associate a set \( AP_i \) of atomic propositions, and a mapping \( V_i \) from local states of \( P_i \) to subsets of \( AP_i \); \( V_i(s_i) \) is the set of atomic propositions that are true in \( s_i \). As \( P_i \) executes transitions and changes its local state, the atomic propositions in \( AP_i \) are updated.

Different local states of \( P_i \) have different truth assignments: \( V_i(s_i) \neq V_i(t_i) \) for \( s_i \neq t_i \). Atomic propositions are not shared: \( AP_i \cap AP_j = \emptyset \) when \( i \neq j \). Other processes can read (via guards) but not update the atomic propositions in \( AP_i \). We define the set of all atomic propositions \( AP = AP_1 \cup \cdots \cup AP_K \). There is also a set \( SH = \{x_1, \ldots, x_m\} \) of shared variables, which can be read and written by every process. These are updated by the action \( A_i \). A global state is a tuple of the form \((s_1, \ldots, s_K, v_1, \ldots, v_m)\) where \( s_i \) is the current local state of \( P_i \) and \( v_1, \ldots, v_m \) is a list giving the current values of \( x_1, \ldots, x_m \), respectively. A guard \( B_i \) is a predicate on global states, and so can reference any atomic proposition and any shared variable. An action \( A_i \) is any piece of terminating pseudocode that updates the shared variables.\(^2\) We write just \( A_i \) for \( true \rightarrow A_i \) and just \( B_i \) for \( B_i \rightarrow \text{skip} \), where skip is the empty assignment.

We model parallelism as usual by the non-deterministic interleaving of the “atomic” transitions of the individual processes \( P_i \). Let \( s = (s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m) \) be the current global state, and let \( P_i \) contain an arc from node \( s_i \) to \( s_i' \) labeled with \( B_i \rightarrow A_i \). We write such an arc as the tuple \((s_i, B_i \rightarrow A_i, s_i')\), and call it a \( P_i \)-arc from \( s_i \) to \( s_i' \). We use just arc when \( P_i \) is specified by the context. If \( B_i \) holds in \( s \), then a permissible next state is \( s' = (s_1, \ldots, s_i', \ldots, s_K, v_1', \ldots, v_m') \) where \( v_1', \ldots, v_m' \) are the new values for the shared variables resulting from action \( A_i \). Thus, at each step of the computation, a process with an enabled arc is non-deterministically selected to be executed next. The transition relation \( R \) is the set of all such \((s, i, s')\). The arc from node \( s_i \) to \( s_i' \) is enabled in state \( s \). An arc that is not enabled is blocked. Our model of computation is a high-atomicity model, since a process \( P_i \) can evaluate the guard \( B_i \), execute the action \( A_i \), and change its local state, all in one action.

Recall that we define a global state to be a tuple of local states and shared variable values, rather than a “name” together with a labeling function \( L \) that gives the associated valuation. A consequence of this definition is that two different global states must differ in either some local state or some shared variable value. Since we require different local states to differ in at least one atomic proposition value, we conclude that two different global states differ in at least one atomic proposition value or one shared variable value.

We define the valuation corresponding to a global state \( s = (s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m) \) as follows. For an atomic proposition \( p_i \in AP_i \): \( s(p_i) = true \) if \( p_i \in V_i(s_i) \), and \( s(p_i) = false \) if \( p_i \notin V_i(s_i) \). For a shared variable \( x_i \), \( 1 \leq i \leq m \): \( s(x_i) = v_i \). We define \( s|AP \) to be the set \( \{p \in AP \mid s(p) = true\} \) i.e., the set of propositions that are true in state \( s \). \( s|AP \) is essentially the projection of \( s \) onto the atomic propositions. Also, \( s|i \) is defined to be \( s_i \), i.e., the local state of \( P_i \) in \( s \). We also define \( s|SH \) to be the set \( \{p, s(x)\} \mid x \in SH\} \), i.e., the set of all pairs consisting of a shared variable \( x \) in \( SH \) together with the value that \( s \) assigns to \( x \).

Let \( St \) be a given set of initial states in which computations of \( P \) can start. A computation path is a sequence of states whose first state is in \( St \) and where each successive pair of states is related by \( R \). A state is reachable if it lies on some computation path. Since we must specify the start states \( St \) in order for the computation paths to be well-defined, we re-define our notion of a program to be \( P = (St, P_1 \parallel \cdots \parallel P_K) \), i.e., a program consists of the parallel composition of \( K \) processes, together with a set \( St \) of initial states.

For technical convenience, and without loss of generality, we assume that no synchronization skeleton contains a node with a self-loop. The functionality of a self-loop (e.g., a busy wait) can always be achieved.

\(^2\) We will only use straight-line code in this paper, so termination is always guaranteed.
by using a loop containing two local states. Thus, a transition by \( P_i \) changes the local state of \( P_i \), and therefore the value of at least one atomic proposition in \( \mathcal{AP}_i \). Hence, no global state \( s \) has a self loop, i.e., a transition by some \( P_i \) both starting and finishing in \( s \).

For a local state \( s_i \), define \( \{s_i\} \) as follows:

**Definition 1 (State-to-Formula Translation)**

\[ \{s_i\} = \left( \bigwedge_{p \in V_i(s_i)} p \right) \land \left( \bigwedge_{p \notin V_i(s_i)} \neg p \right) \]

where \( p \) ranges over \( \mathcal{AP}_i \).

\( \{s_i\} \) converts a local state \( s_i \) into a propositional formula over \( \mathcal{AP}_i \).

If \( s \) is a global state and \( B \) is a guard, we define \( s(B) \) by the usual inductive scheme: \( s("x = c") = true \) iff \( s(x) = c \), \( s(B_1 \land B_2) = true \) iff \( s(B_1) = true \) and \( s(B_2) = true \), \( s(\neg B_1) = true \) iff \( s(B_1) = false \). If \( s(B) = true \), we also write \( s \models B \).

### 2.2 The Global State Transition Diagram of a Concurrent Program

**Definition 2 (Global state transition diagram)** Given a concurrent program \( P = P_1 \parallel \cdots \parallel P_k \) and a set \( St \) of initial global states for \( P \), the global state transition diagram generated by \( P \) is a Kripke structure \( M = (St, S, R) \) given as follows: (1) \( S \) is the smallest set of global states satisfying (1.1) \( St \subseteq S \) and (1.2) if there exist \( s \in S, i \in [K]^3 \), and \( u \) such that \( (s, i, u) \) is in the next-state relation defined above in Section 2.1, then \( u \in S \), and (2) \( R \) is the next-state relation restricted to \( S \).

We define strong bisimulation in the standard way.

**Definition 3 (Strong Bisimulation)** Let \( M = (St, S, R) \) and \( M' = (St', S', R') \) be two Kripke structures with the same underlying set \( \mathcal{AP} \) of atomic propositions. A relation \( B \subseteq S \times S' \) is a strong bisimulation iff:

1. if \( B(s, s') \) then \( s \models \mathcal{AP} = s' \models \mathcal{AP} \)
2. if \( B(s, s') \) and \( (s, i, u) \in R \) then \( \exists u' : (s', i, u') \in R' \land B(u, u') \)
3. if \( B(s, s') \) and \( (s', i, u') \in R \) then \( \exists u : (s, i, u) \in R \land B(u, u') \)

We also define \( \sim \) to be the union of all strong bisimulation relations:

\[ \sim = \bigcup \{B : B \text{ is a strong bisimulation}\} \]

We say that \( M \) and \( M' \) are strongly bisimilar, and write \( M \sim M' \), if and only if there exists a strong bisimulation \( B \) such that \( \forall s \in St, \exists s' \in St' : B(s, s') \) and \( \forall s' \in St' \exists s \in St : B(s, s') \).

### 3 Pairwise normal form

Let \( \oplus, \otimes \) be binary infix operators. A *general guarded command* [2] is either a guarded command as given in Section 2.1 above, or has the form \( G_1 \oplus G_2 \) or \( G_1 \otimes G_2 \), where \( G_1, G_2 \) are general guarded commands. Roughly, the operational semantics of \( G_1 \oplus G_2 \) is that either \( G_1 \) or \( G_2 \), but not both, can be executed, and the operational semantics of \( G_1 \otimes G_2 \) is that both \( G_1 \) or \( G_2 \) must be executed, that is, the guards of both \( G_1 \) and \( G_2 \) must hold at the same time, and the bodies of \( G_1 \) and \( G_2 \) must be executed simultaneously, as a single parallel assignment statement. For the semantics of \( G_1 \otimes G_2 \) to be well-defined, there must be no conflicting assignments to shared variables in \( G_1 \) and \( G_2 \). This will always be the case for the programs we consider. We refer the reader to [2] for a comprehensive presentation of general guarded commands.

---

3We use \([K]\) for the set consisting of the natural numbers \( 1, \ldots, K \).
Definition 4 (Pairwise Normal Form) A concurrent program \( P = P_1 \| \cdots \| P_K \) is in pairwise normal form iff the following four conditions all hold:

1. every arc \( a_i \) of every process \( P_i \) has the form
   \[ a_i = (s_i, \bigoplus_{j \in \{1, \ldots, n\}} B_{i,\ell}^j \rightarrow A_{i,\ell}^j, t_i) \]
   where \( B_{i,\ell}^j \rightarrow A_{i,\ell}^j \) is a guarded command, \( I \) is an irreflexive symmetric relation over \([K]\) that defines a “interconnection” (or “neighbors”) relation amongst processes, and \( I(i) = \{ j \mid (i, j) \in I \} \).

2. variables are shared in a pairwise manner, i.e., for each \((i, j) \in I \), there is some set \( SH_{ij} \) of shared variables that are the only variables that can be read and written by both \( P_i \) and \( P_j \).

3. \( B_{i,\ell}^j \rightarrow A_{i,\ell}^j \) can reference only variables in \( SH_{ij} \) and atomic propositions in \( AP_j \), and

4. \( A_{i,\ell}^j \) can update only variables in \( SH_{ij} \).

For each neighbor \( P_j \) of \( P_i \), \( \bigoplus_{\ell \in \{1, \ldots, n\}} B_{i,\ell}^j \rightarrow A_{i,\ell}^j \) specifies \( n \) alternatives \( B_{i,\ell}^j \rightarrow A_{i,\ell}^j, 1 \leq \ell \leq n \) for the interaction between \( P_i \) and \( P_j \) as \( P_i \) transitions from \( s_i \) to \( t_i \). \( P_i \) must execute such an interaction with each of its neighbors in order to transition from \( s_i \) to \( t_i \). We emphasize that \( I \) is not necessarily the set of all pairs, i.e., there can be processes that do not directly interact by reading each others atomic propositions or reading/writing pairwise shared variables. We do not assume, unless otherwise stated, that processes are isomorphic, or “similar.”

We use a superscript \( I \) to indicate the relation \( I \), e.g., process \( P_i^I \) and \( P_j^I \)-arc \( a_i^I \). We define \( a_i^I.begin = s_i, a_i^I.guard = \bigvee_{\ell \in \{1, \ldots, n\}} B_{i,\ell}^j \rightarrow A_{i,\ell}^j \), and \( a_i^I.guard = \bigwedge_{\ell \in \{1, \ldots, n\}} a_i^I.guard_j \). If \( P_i^I = P_i^I \| \cdots \| P_k^I \) is a concurrent program with interconnection relation \( I \), then we call \( P_i^I \) an \( I \)-system. For the special case when \( I = \{(i, j) \mid i, j \in [K], i \neq j\} \), i.e., \( I \) is the complete interconnection relation, we omit the superscript \( I \).

In pairwise normal form, the synchronization code for \( P_i^I \) with one of its neighbors \( P_j^I \) (i.e., \( \bigoplus_{\ell \in \{1, \ldots, n\}} B_{i,\ell}^j \rightarrow A_{i,\ell}^j \)) is expressed separately from the synchronization code for \( P_j^I \) with another neighbor \( P_k^I \) (i.e., \( \bigoplus_{\ell \in \{1, \ldots, n\}} B_{i,\ell}^j \rightarrow A_{i,\ell}^j \)). We can exploit this property to define “subsystems” of an \( I \)-system \( P \) as follows. Let \( J \subseteq I \) and \( range(J) = \{ i \mid \exists j : (i, j) \in J \} \). If \( a_i^I \) is a arc of \( P_i^I \) then define \( a_i^J \) = \( (s_i, \bigoplus_{j \in \{i, \ldots, n\}} B_{i,j}^j \rightarrow A_{i,j}^j, t_i) \). Then the \( J \)-system \( P^J \) is \( P_i^J \| \cdots \| P_k^J \) where \( \{ j_1, \ldots, j_n \} = range(J) \) and \( P_j^J \) consists of the arcs \( a_i^J \mid a_i^I \) is a arc of \( P_i^I \). Intuitively, a \( J \)-system consists of the processes in \( range(J) \), where each process contains only the synchronization code needed for its \( J \)-neighbors, rather than its \( I \)-neighbors. If \( J = \{i, j\} \) for some \( i, j \) then \( P^J \) is a pair-system, and if \( J = \{i, j, k\} \) for some \( i, j, k \) then \( P^J \) is a triple-system. For \( J \subseteq I \), \( M_J = (S_{i,j}, S_j, R_j) \) is the GSTD of \( P^J \) as defined in Section 2.1, and a global state of \( P^J \) is a \( J \)-state. If \( J = \{i, j\} \), then we write \( M_{ij} = (S_{i,j}, S_j, R_j) \) instead of \( M_J = (S_{i,j}, S_j, R_j) \).

In [1, 2, 4] we give, in pairwise normal form, solutions to many well-known problems, such as dining philosophers, drinking philosophers, mutual exclusion, k-out-of-n mutual exclusion, two-phase commit, and replicated data servers. We conjecture that any finite-state concurrent program can be rewritten (up to strong bisimulation) in pairwise normal form. The restriction to pairwise normal form enables us to mechanically verify certain correctness properties very efficiently. Recall that \( K \) is the number of processes, \( b \) is the maximum branching in the local state transition relation of a single process, and \( N \) is the size of the largest process. Then, safety and liveness properties that can be expressed over pairs of processes can be verified in time \( O(K^2N^2) \) by model-checking pair-systems, [1, 2], and deadlock-freedom can be verified in time in \( O(K^3N^3) \) or \( O(K^4N^4) \) using either of two conservative tests [5], which in turn operate by model checking triple-systems. Exhaustive state-space enumeration would of course require \( O(N^K) \) time.

4 The Pairwise Expressiveness Result

Let \( Q = (S_{Q,1}, Q_1 \| \cdots \| Q_K) \) be an arbitrary finite-state shared memory concurrent program as defined in Section 2.1 above, with each process \( Q_i \) having an associated set \( AP_i \) of atomic propositions and with shared variables \( x_1, \ldots, x_m \). The transformation of \( Q \) to pairwise normal form proceeds in three phases, as given in the sequel.
TRANSFORM($M_Q, M'_Q$)

$S'_Q := S_Q; S'_Q := S_Q; R'_Q := R_Q$;

repeat until there is no change in $M'_Q$

let $s$ be a state in $M'_Q$ such that $|in\_procs(s)| > 1$;

forall $i \in in\_procs(s)$ do

create a new marked state $s^i$ such that $s^i |\mathcal{AP} = s |\mathcal{AP}$, $s^i |\mathcal{SH} = s |\mathcal{SH}$

if $s \in S_Q$ then $S'_Q := S'_Q \cup \{s^i\}$ endif;

forall $j, u : (s, j, u) \in R_Q$ do $R'_Q := R'_Q \cup \{(s^i, j, u)\}$ endfor;

forall $u : (u, i, s) \in R_Q$ do $R'_Q := R'_Q \cup \{(u, i, s^i)\}$ endfor;

$S'_Q := S'_Q - \{s^i\};$

$S'_Q := S'_Q - \{s\};$

remove all transitions incident on $s$ from $R'_Q$
endfor
endrepeat

Figure 1: Transformation of $M_Q$ so that all incoming transitions are labeled with the same process index.

4.1 Phase One

First, we generate $M_Q$, the GSTD of $Q$, as given by Definition 2. By construction of Definition 2, all states in $M_Q$ are reachable. We then execute the algorithm given in Figure 1 on $M_Q$ which transforms $M_Q$ into a Kripke structure $M'_Q = (S'_Q, S'_Q, R'_Q)$ which is bisimilar to $M_Q$ and which has the property that all incoming transitions into a state are labeled with the same process index. This is not strictly necessary, but significantly simplifies the transformation to pairwise normal form.

Define $in\_procs(s) = \{i \in [K] | \exists s' : (s', i, s) \in R_Q\}$. We also introduce a new shared variable $in$ whose value in a state $s$ will be the process index that labels the transitions incoming into $s$.

**Proposition 1** Procedure TRANSFORM terminates.

**Proof.** Each iteration of the repeat loop (line 2) reduces the number of states $s$ such that $|in\_procs(s)| > 1$ by one. Since $M'_Q$ is initially set to $M_Q$, which is finite, this cannot go on forever. $\square$

**Proposition 2** $M'_Q \sim M_Q$ is a loop invariant of the repeat loop (line 2) of TRANSFORM.

**Proof of Proposition 2.** Proof. Let $n_0$ be the number of iterations that the repeat loop executes. Let $M^n = (S^n, S^n, R^n)$ be the value of $M'_Q$ at the end of the $n$'th iteration, (for all $n \leq n_0$) with $M^0$ being the initial value $M_Q$. We will also use the superscript $n$ for states in $M^n$, when needed. We show that $\forall n : 0 < n \leq n_0 : M^{n-1} \sim M^n$.

Consider the $n$'th iteration of the repeat loop. In this iteration, $M^n$ results from $M^{n-1}$ by deleting some state $s$ and adding some states $s^{i_1} \ldots s^{i_\ell}$, where $\{i_1, \ldots, i_\ell\} = in\_procs(s)$. Since each of $s^{i_1} \ldots s^{i_\ell}$ have the same successor states as $s$, and agree with $s$ on the values of all atomic propositions, we have $s \sim s^{i_1} \ldots s^{i_\ell}$. Let $u$ be an arbitrary predecessor of $s$ in $M^{n-1}$, i.e., $(u^{n-1}, j, s) \in R^{n-1}$, where $u^{n-1}$ indicates the occurrence of $u$ in $M^{n-1}$. At the end of the iteration, we have $(u^n, j, s^{i}) \in R^n$. Since $s \sim s^{i}$, we have $u^{n-1} \sim u^n$, i.e., the occurrence of $u$ in $M^{n-1}$ is bisimilar to the occurrence of $u$ in $M^n$. Since all other states in $M^{n-1}$ and $M^n$ have an unchanged set of successors, we conclude that $M^{n-1} \sim M^n$.

By a straightforward induction on $n$, and using the transitivity of $\sim$, we can show that $\forall n : 0 < n \leq n_0 : M^0 \sim M^n$. Thus $M^0 = M^{n_0}$. Now $M_Q = M^0$ and $M'_Q = M^{n_0}$, and the proposition is established. $\square$
Proposition 3 Upon termination of procedure TRANSFORM, 
(1) $M'_Q \sim M_Q$, and 
(2) every state $s$ in $M'_Q$ satisfies $|\text{in_procs}(s)| \leq 1$.

Proof. (1) follows from Proposition 2. (2) follows immediately from inspecting line 2 of procedure TRANSFORM. □

For all $s \in S'_Q$ such that $|\text{in_procs}(s)| = 1$, define $\bar{m}(s)$ to be the unique $i$ such that $\exists s' : (s', i, s) \in R'_Q$.

Proposition 4 Upon termination of procedure TRANSFORM, for any two states $s, u$ in $M'_Q$, $s|\mathcal{AP} \neq u|\mathcal{AP}$ or $s|\mathcal{SH} \neq u|\mathcal{SH}$ or $\text{in}(s) \neq \text{in}(u)$.

Proof. Immediate by construction of procedure TRANSFORM. □

4.2 Phase Two

We exploit the unique incoming process index property of $M'_Q$ to extract a program $P = (St_P, P_1, \cdots, P_K)$ from $M'_Q$ such that $P$ is bisimilar to $Q = (St_Q, Q_1, \cdots, Q_K)$ and $P$ is in pairwise normal form. The interconnection relation $I$ for $P$ is the complete relation, and so we omit the superscripts $I$ on $P$ and $P_i$. $P$ operates by emulating the execution of $Q$. In the sequel, let $i, j, k$ implicitly range over $[K]$, with possible further restriction, e.g., $i \neq j$. With each process $P_i$ we associate the following state variables, with the indicated access permissions and purpose

- The atomic propositions in $\mathcal{AP}_i$. These are written by $P_i$ and read by all processes. For each process $P_i$, these enable $P_i$ to emulate the local state of $Q_i$, which is defined by the same set $\mathcal{AP}_i$ of atomic propositions.

- A shared variable $x_{ij}$ for every $x \in \mathcal{SH}$ and $j \in [K]$. These are written by $P_i$ and read by $P_j$. These enable $P_i$ to emulate the updates that $Q_i$ makes to $x$. When $P_i$ is the last process to have executed, any other process $P_j$ will read $x_{ij}$ to find the correct emulated value of $x$, since this value will have been computed by $P_i$ and stored in $x_{ij}$ for all $j \in [K]$. For technical convenience, we admit $x_{ij}$. We select some $\ell \in [K] - \{i\}$ arbitrarily and define $x_{ij}$ to be shared pairwise between $P_i$ and $P_{\ell}$. This is needed to conform technically to Definition 4. $P_{\ell}$ will not actually reference $x_{ij}$.

- A timestamp $t_{ij}$ for every $j \in [K]$. These are written and read by $P_i$ only. Timestamps have values in $\{0, 1, 2\}$. We define orderings $<_\alpha, >_\alpha$ on timestamps as follows [8]: $0 <_\alpha 1$, $1 <_\alpha 2$, and $2 <_\alpha 0$, and $t \succ_\alpha t'$ iff $t' <_\alpha t$. Note that $<_\alpha$ is not transitive. The purpose of $t_{ij}$ is to enable the pair of processes $P_i$ and $P_j$ to establish an ordering between themselves by computing $t_{ij} <_\alpha t_{ij}$. If $t_{ij} >_\alpha t_{ij}$, then $P_i$ executed a transition more recently than $P_j$, and vice-versa. The timestamp $t_{ij}$ is unused, so we do not worry about initializing it, or what is value is in general.

- A timestamp vector $tv_{ij}$ for every $j \in [K]$. A $K$-tuple whose value is maintained equal to $(t_{i1}, \ldots, t_{K})$. It is written by $P_i$ and read by $P_i$ and $P_j$. Its purpose is to allow $P_i$ to communicate to $P_j$ the values of $P_i$’s timestamps w.r.t. all other processes. By reading all $tv_{ij}$, $i \in [K] - \{j\}$, process $P_j$ can correctly infer the index of the last process to execute. This allows $P_j$ to read the correct emulated values of all shared variables. We use $tv_{ij}_k$ to denote the $k$th element of $tv_{ij}$, which is the value of $t_{ij}^k$.

For technical convenience, we admit $tv_{ij}$. We select some $\ell \in [K] - \{i\}$ arbitrarily and define $tv_{ij}$ to be shared pairwise between $P_i$ and $P_{\ell}$. This is needed to conform technically to Definition 4. $P_{\ell}$ will not actually reference $tv_{ij}$.

For all the above, the order of subscripts does not matter, e.g., $tv_{ij}$ and $tv_{ji}$ are the same variable, etc.

The essence of the emulation is to deal correctly with the shared variables. This depends upon every process being able to compute the index of the last process to execute, as described above. Define the
auxiliary ("ghost") variable last to be the index of the last process to make a transition. As described above, every process \( P_i \) can compute the value of last (last is not explicitly implemented, since doing so would violate pairwise normal form). Then, \( P_j \) reads the variable \( x_{last,j}^i \) that it shares with \( P_{last} \) to find an up to date value for the variable \( x \) in \( Q \). Together with the unique incoming process index property of \( M'_Q \), this allows \( P_j \) to accurately determine the currently simulated global state of \( M'_Q \). \( P_j \) can then update its associated shared variables and atomic propositions to accurately emulate a transition in \( M'_Q \).

Let \( M_P \) be the GSTD of \( P \), as given by Definition 2. We will define \( P = (St_P, P_1 || \cdots || P_K) \) so that \( M'_Q \) and \( M_P \) are bisimilar.

We start with \( St_P \). For each initial state \( u_0 \) of \( M'_Q \), we create a corresponding initial state \( r_0 \in St_P \) so that:

\[
\begin{align*}
&\quad r_0 \upharpoonright AP = u_0 \upharpoonright AP \\
&\quad \forall x \in SH, \forall i, j, r_0(x_{ij}^0) = u_0(x)
\end{align*}
\]

Now for the bisimulation between \( M'_Q \) and \( M_P \) to work properly, we will require that \( in(u) = s(last) \), where \( u, s \) are bisimilar states of \( M'_Q \) and \( M_P \), respectively. It is possible, however, that some initial state \( u_0 \) of \( M'_Q \) does not have an incoming transition, and so \( in(u_0) \) is undefined. We deal with this as follows.

Call an initial state (of either \( M'_Q \) or \( M_P \)) that does not have an incoming transition a source state. Since we defined the corresponding \( r_0 \) above so that \( x_{ij}^0 \) has the correct value (namely \( u_0(x) \)) for all \( i, j \), we can let any process be the “last”, as determined by the timestamps. Thus, for a source state \( u_0 \) in \( M'_Q \) and its corresponding source state \( r_0 \) in \( M_P \), we set:

\[
r_0(t_i^j) = \begin{cases} 1 & \text{if } i = 1 \land j \neq 1 \\ 0 & \text{if } i \neq 1 \land j = 1 \\ X & \text{if } i \neq 1 \land j \neq 1 \end{cases}
\]

where \( X \) denotes a “don’t care,” i.e., any value in \( \{0, 1, 2\} \) can be used. This has the effect of making \( P_1 \) the “last” process to have executed in a source state, i.e., setting \( r_0(last) = 1 \). We now extend the definition of \( in \) to source states by defining \( in(u_0) = 1 \) for every source state \( u_0 \in St'_Q \). Together with the fact that states in \( M'_Q \) are uniquely determined by the atomic proposition and shared variable values, this automatically takes care of the bisimulation matching between source states in \( M'_Q \) and source states in \( M_P \), without the need for an extra case analysis. Note also that \( in(u) \) is now defined for all states \( u \) in \( M'_Q \).

For an initial state \( u_0 \) of \( M'_Q \) that is not a source state, and its corresponding initial state \( r_0 \) in \( M_P \), we set:

\[
r_0(t_i^j) = \begin{cases} 1 & \text{if } i = in(u_0) \land j \neq in(u_0) \\ 0 & \text{if } i \neq in(u_0) \land j = in(u_0) \\ X & \text{if } i \neq in(u_0) \land j \neq in(u_0) \end{cases}
\]

where again \( X \) means “don’t care.” This has the effect of setting \( r_0(last) = in(u_0) \), as required.

For all initial states \( r_0 \in St_P \), whether they are source states or not, we set the timestamp vector values so that:

\[
\forall i, j, k, r_0(t_v^i,j,k) = r_0(t_k^j)
\]

For each transition \((u, i, v) \in M'_Q\), we generate a single arc \( ARC_i^{u,v} \) in \( P_i \) as follows. \( ARC_i^{u,v} \) starts in local state \( u|i \) of \( P_i \) and ends in local state \( v|i \) of \( P_i \). Let \( in(u) = c \). Then the guard \( B_i^{v,u} \) of \( ARC_i^{u,v} \) is defined as follows:

\[
B_i^{v,u} \overset{\text{df}}{=} (last = c) \land \bigwedge_{j \neq i} \{u|j\} \land (\bigwedge_{x \in SH} x_{ci}^e = u(x))
\]

The first conjunct checks that the last process that executed is the process with index \( in(u) \). The second
\[ \text{step}(t, t') \]
Precondition: \( 0 \leq t, t' \leq 2 \), that is, \( t, t' \) are timestamp values
if \( t >_o t' \) then return \((t)\)
else
  if \( t = 0 \land t' = 1 \) then return \((2)\) endif;
  if \( t = 1 \land t' = 2 \) then return \((0)\) endif;
  if \( t = 2 \land t' = 0 \) then return \((1)\) endif;
endif

Figure 2: The \textit{step} procedure.

conjunct checks that all atomic propositions have the values assigned to them by global state \( u \). The third conjunct checks that all shared variables have the values assigned to them by global state \( u \).

The action \( A_{u,v}^i \) of \( \text{ARC}_{u,v}^i \) is defined to be
\[
\| j \neq i \ t^j_i := \text{step}(t^j_i, tv^j_{ji}); \\
\| j \ tv^j_{ij} := (t^1_i, \ldots, t^K_i); \\
\| j, x \in SH \ x^j_{ij} := v(x)
\]
where \( \text{step}(t, t') \) is given in Figure 2. This cannot be factored into pairwise actions \( A^i_{j,m} \) because all the \( t^j_i \) are used to update all the \( tv^j_{ij} \). The solution is to make the \( t^j_i \) part of the local state of \( P_i \). We do this in phase 3 below. For now, we show that program \( P \) with the arcs given by \( \text{ARC}_{u,v}^i = (u|i, B_{u,v}^i \rightarrow A_{u,v}^i, v|i) \) is bisimilar to program \( Q \).

\textbf{Proposition 5} The following are invariants of \( P \):

1. \( \land_{i,j,k \neq i} tv^j_{ij}.k = t^k_i \)
2. \( \land_i((\text{last} = i) \equiv \land_{j \neq i} t^j_i >_o t^j_i) \)
3. \( \land_{i,j,k} x^j_{ij} = x^j_{ik} \)

\textbf{Proof}: By construction of \( P \): \( St_P \) is defined so that the initial states all satisfy the above, and the actions \( A_{u,v}^i \) of every process \( P_i \) of \( P \) are defined so that their execution preserves the above. \( \square \)

\textbf{Definition 5} Define \( \bowtie \subseteq S'_Q \times S_P \) as follows. For \( u \in S'_Q \), \( r \in S_P \), \( u \bowtie r \) iff:

1. \( u \upharpoonright \mathcal{AP} = r \upharpoonright \mathcal{AP} \)
2. \( \in(u) = r(last) \)
3. \( \land_{x \in S_H, k} r(last) = k \Rightarrow (\land_i u(x) = r(x^k_i)) \)

\textbf{Theorem 6} \( \bowtie \) is a strong bisimulation

\textbf{Proof of Theorem 6}. \textit{Proof}. Let \( u \in S'_Q \), \( r \in S_P \), and \( u \bowtie r \). We must show that all three clauses of Definition 3 hold, that is:
1. if \( u \mathbin{ar} r \) then \( u \mathbin{AP} = r \mathbin{|AP} \)

2. if \( u \mathbin{ar} r \) and \((u, i, v) \in R_Q \) then \( \exists s : (r, i, s) \in R_P \land v \mathbin{ar} s \)

3. if \( u \mathbin{ar} r \) and \((r, i, s) \in R_P \) then \( \forall v : (u, i, v) \in R_Q \land v \mathbin{ar} s \)

Clause 1 holds by virtue of clause 1 of Definition 5.

Proof of clause 2. Assume \((u, i, v) \in R_Q \) and let \( in(u) = c \). We show that there exists \( s \) such that \((r, i, s) \in R_P \) and \( v \mathbin{ar} s \). By our construction of \( P \) above, the transition \((u, i, v) \) generates the arc \( A_{c}^{u,v} \) in \( P \). By definition, the guard \( B_{c}^{u,v} \) of \( A_{c}^{u,v} \) is

\[
(last = c \land \bigwedge_{j \neq i} \{u[j]\} \land (\bigwedge_{x \in SH} x_{ci}^c = u(x))).
\]

(a)

Now by Definition 5 and \( u \mathbin{ar} r \), we have \( in(u) = r(last) \). Hence \( r \models last = c \). Also by Definition 5 and \( u \mathbin{ar} r \), we have \( u \mathbin{AP} = r \mathbin{|AP} \). Again by Definition 5 and \( u \mathbin{ar} r \), we have \( \bigwedge_{x \in SH} r(last) = c \Rightarrow u(x) = r(x_{ci}^c) \). Hence \( \bigwedge_{x \in SH} u(x) = r(x_{ci}^c) \). And so \( r \models (\bigwedge_{x \in SH} x_{cj}^c = u(x)) \).

Since \( r \) satisfies all three conjuncts of (a), it follows that the guard of \( A_{c}^{u,v} \) is true in state \( r \), and therefore \( A_{c}^{u,v} \) is enabled in \( r \). By Proposition 5 and inspection of the action \( A_{c}^{u,v} \) of \( A_{c}^{u,v} \), executing of \( A_{c}^{u,v} \) leads to a state \( s \) such that

\[
s(last) = i \text{ and } s \mathbin{AP} = v \mathbin{AP} \text{ and } (\bigwedge_{j} x_{ij}^i = v(x)).
\]

By Definition 5, we have \( v \mathbin{ar} s \), as required.

Proof of clause 3. Assume \((r, i, s) \in R_P \) . We show that there exists \( v \) such that \((u, i, v) \in R_Q \) and \( v \mathbin{ar} s \).

By our construction of \( P \) above, the transition \((r, i, s) \) results from executing an arc \( A_{c}^{u,v} \) in \( P \), for some \( w, v \). Let \( in(w) = c \). By definition of \( A_{c}^{u,v} \), we have \( r \models (\bigwedge_{x \in SH} r(x_{ci}^c) = w(x)) \). Hence, by the definition of \((w) \) (Definition 1), \( r \mathbin{|AP} = w \mathbin{|AP} \). Also by definition of \( A_{c}^{u,v} \), we have \( r(last) = in(w) = c \land (\bigwedge_{x \in SH} r(x_{ci}^c) = w(x)) \). Hence:

\[
r(last) = in(w) = c \text{ and } u \mathbin{AP} = w \mathbin{|AP} \text{ and } (\bigwedge_{x \in SH} r(x_{ci}^c) = w(x)).
\]

(b)

Since \( u \mathbin{ar} r \), we have

\[
r(last) = in(u) \text{ and } u \mathbin{|AP} = r \mathbin{|AP} \text{ and } (\bigwedge_{x \in SH} r(x_{ci}^c) = u(x)).
\]

From (b), \( r(last) = c \). Hence

\[
r(last) = in(u) \text{ and } u \mathbin{AP} = r \mathbin{|AP} \text{ and } (\bigwedge_{x \in SH} r(x_{ci}^c) = u(x)).
\]

(c)

From (b,c) we have

\[
in(w) = in(u) \text{ and } w \mathbin{AP} = u \mathbin{|AP} \text{ and } (\bigwedge_{x \in SH} w(x) = u(x)).
\]

(d)

Since all global states differ in either some atomic proposition or some shared variable, or some incoming transition, by Proposition 4, we conclude from (d) that \( w = u \).

By Proposition 5 and inspection of the action \( A_{c}^{u,v} \) of \( A_{c}^{u,v} \), executing \( A_{c}^{u,v} \) can only lead to a state \( s \) such that

\[
s(last) = i \text{ and } s \mathbin{AP} = v \mathbin{|AP} \text{ and } (\bigwedge_{j} x_{ij}^i = v(x)).
\]

By Definition 5, we have \( v \mathbin{ar} s \), as required.

\(\square\)
Corollary 7 \( M'_Q \sim M_P \).

Proof. From Definition 5 and our definition of the initial states of \( P \), we see that for every initial state \( u_0 \) of \( M'_Q \), there exists an initial state \( r_0 \) of \( M_P \) such that \( u_0 \sim r_0 \), and vice-versa. The result then follows from Theorem 6 and Definition 3. \( \Box \)

4.3 Phase Three

We now express \( ARCl_i^{u,v} \) in a form that complies with Definition 4, that is, as \( \otimes_{j \in t(i)} \otimes_{\ell \in \{1, \ldots, n\}} B_{i,\ell}^{j} \rightarrow A_{i,\ell}^{j} \), where \( B_{i,\ell}^{j} \) can reference only variables in \( SH_{ij} \) and atomic propositions in \( AP_{ij} \), and \( A_{i,\ell}^{j} \) can update only variables in \( SH_{ij} \). Recall that \( ARCl_i^{u,v} = (u[i], B_{i}^{u,v} \rightarrow A_{i}^{u,v}). \) For the rest of this section, let \( in(u) = c. \)

First consider \( B_i^{u,v} \). By definition \( B_i^{u,v} = (last = c) \land \bigwedge_{j \neq i} \{u[j]\} \land (\bigwedge_{x \in SH} x^c_i = u(x)) \). Now \( \{u[j]\} \) is a propositional formula over \( AP_{ij} \), and so \( \bigwedge_{j \neq i} \{u[j]\} \) is a conjunction of propositional formulae over \( AP_{ij} \), and so it poses no problem. Likewise, since \( (\bigwedge_{x \in SH} x^c_i = u(x)) \) is a conjunction over pairwise shared variables, it also is unproblematic. \( last = c \) is not in the pairwise form given above since it refers to the ghost variable \( last \).

Now \( last = c \) checks that the last process to execute is \( P_c \). In terms of timestamps, it is equivalent to \( \bigwedge_{j \neq c} tv_{ci,j} \supset tv_{ji,c} \), i.e., \( P_c \) has executed more recently than all other processes. However, the timestamps \( t_j^c \) are inaccessible to \( P_c \), and the \( t_j^c \) are accessible to \( P_i \) only in the special case that \( c = i \), which does not hold generally. The purpose of the timestamp vectors is precisely to deal with this problem. Recall that \( tv_{ci,j} \) is maintained equal to \( t_j^c \), and \( tv_{ji,c} \) is maintained equal to \( t_j^c \). Hence, we replace \( last = c \) by the equivalent

\[
\bigwedge_{j \neq c} tv_{ci,j} \supset tv_{ji,c}. (*)
\]

which moreover can be evaluated by \( P_i \), since it refers only to timestamp vectors that are accessible to \( P_i \).

Now the expression \( tv_{ci,j} \supset tv_{ji,c} \) refers to \( tv_{ci} \), which is shared by \( P_c \) and \( P_i \), and \( tv_{ji} \), which is shared by \( P_j \) and \( P_i \). Thus it is not in pairwise form. We fix this as follows. \( tv_{ci,j} \supset tv_{ji,c} \) is equivalent to \( (tv_{ci,j} = 0 \land tv_{ji,c} = 1) \lor (tv_{ci,j} = 1 \land tv_{ji,c} = 2) \lor (tv_{ci,j} = 2 \land tv_{ji,c} = 0) \), by definition of \( \supset \). Hence, (*) is equivalent to

\[
\bigwedge_{j \neq c} (tv_{ci,j} = 0 \land tv_{ji,c} = 1) \lor (tv_{ci,j} = 1 \land tv_{ji,c} = 2) \lor (tv_{ci,j} = 2 \land tv_{ji,c} = 0).
\]

This formula has length in \( O(K) \). We convert this to disjunctive normal form, resulting in a formula of length in \( O(exp(K)) \). Let the result be \( D_1 \lor \ldots \lor D_n \) for some \( n \). Each \( D_m \), \( 1 \leq m \leq n \) is a conjunction of literals, where each literal has one of the forms \( (tv_{ci,j} = op ts) \), \( (tv_{ji,c} = op ts) \), where \( op \in \{=, \neq \} \), and \( ts \in \{0, 1, 2\} \). Specifically,

\[
D_m = LIT_m^c(tv_{ci,j}) \land \bigwedge_{j \neq \{c, i\}} LIT_m^j(tv_{ji,c}),
\]

where \( LIT_m^c(tv_{ci,j}) \) is a conjunction of literals of the form \( tv_{ci,j} = op ts \), and \( LIT_m^c(tv_{ji,c}) \) is a conjunction of literals of the form \( tv_{ji,c} = op ts \). Moreover, since logical equivalence to (*) has been maintained, we have

\[
(D_1 \lor \ldots \lor D_n) \equiv (last = c).
\]

For \( m \in \{1, \ldots, n\} \), define:

\[
B_i^{u,v}(m) \overset{df}{=} D_m \land \bigwedge_{j \neq i} \{u[j]\} \land (\bigwedge_{x \in SH} x^c_i = u(x))
\]

where we abuse notation by using \( B_i^{u,v} \) as the name for the “array” of guards \( B_i^{u,v}(m) \), and also as the name for the guard of \( ARCl_i^{u,v} \), as defined above. The use of the index \( (m) \) will always disambiguate these two uses.

10
We now define the set of arcs $\text{ARCS}^u,v_i$ to contain $n$ arcs, $a(1), \ldots, a(n)$, where

$$a(m) \overset{\text{df}}{=} (u|i, B_i^u,v(m) \rightarrow A_{i}^u,v, v|i)$$

for all $m \in 1, \ldots, n$. In particular, all these arcs start in local state $u|i$ of $P_i$ and end in local state $v|i$ of $P_i$.

**Proposition 8** \(\bigvee_{1 \leq m \leq n} B_i^u,v(m) \equiv B_i^u,v\)

**Proof.** Immediate from the definitions and distribution of $\land$ through $\lor$. \(\square\)

It remains to show how each $a(m)$ can be rewritten into pairwise normal form. For all $j \notin \{i, c\}$, define

$$B_i^u,v(m, j) \overset{\text{df}}{=} \text{LIT}^c_i((tv_j^i) \land \{u|j\})$$

For $j = c$.

$$B_i^u,v(m, c) \overset{\text{df}}{=} \text{LIT}^c_i((tv_{ci}^i) \land \{u|c\}) \land (\bigwedge_{x \in \text{SH}} x_{ci}^c = u(x))$$

Note that this works for both $c \neq i$ and $c = i$. The case $c = i$ is why we needed to allow $x_i^i$ and $tv_i^i$. Otherwise we would need a special case to deal with $c = i$. In effect, when $c = i$ we include $B_i^u,v(m, c)$ as a conjunct of $B_i^u,v(m, i)$, where $P_i$ is the process arbitrarily chosen to “share” $x_i^i$ and $tv_i^i$ with $P_i$. This allows us to conform to pairwise normal form, and use \((\bigwedge_{j \neq i} B_i^u,v(m, j))\) as the guard of the arc:

**Proposition 9** \(\bigwedge_{j \neq i} B_i^u,v(m, j) \equiv B_i^u,v(m)\)

**Proof of Proposition 9.** Proof. by definition, $B_i^u,v(m) = D_m \land \bigwedge_{j \neq i} \{u|j\} \land (\bigwedge_{x \in \text{SH}} x_{ci}^c = u(x))$. We also have, by construction, $D_m = \text{LIT}^c_i((tv_{ci}^i) \land \bigwedge_{j \neq \{c,i\}} \text{LIT}^c_i((tv_j^i) \land \{u|j\}))$. Hence $B_i^u,v(m) \equiv \text{LIT}^c_i((tv_{ci}^i) \land (\bigwedge_{j \neq \{c,i\}} \{u|j\} \land (\bigwedge_{x \in \text{SH}} x_{ci}^c = u(x))$.

Splitting up conjunctions and rearranging gives us:

$$B_i^u,v(m) = (\bigwedge_{j \notin \{c,i\}} \text{LIT}^c_i((tv_j^i) \land \{u|j\})) \land (\bigwedge_{j \notin \{c,i\}} \{u|j\}) \land (\bigwedge_{x \in \text{SH}} x_{ci}^c = u(x))$$

Grouping together the first two conjunctions, and the last three:

$$B_i^u,v(m) = (\bigwedge_{j \notin \{c,i\}} \text{LIT}^c_i((tv_j^i) \land \{u|j\})) \land \{u|c\} \land (\bigwedge_{x \in \text{SH}} x_{ci}^c = u(x))$$

Now $\text{LIT}^c_i((tv_j^i) \land \{u|j\}$ is just $B_i^u,v(m, j)$, and $\{u|c\} \land (\bigwedge_{x \in \text{SH}} x_{ci}^c = u(x))$ is just $B_i^u,v(m, c)$. Hence

$$B_i^u,v(m) = \bigwedge_{j \neq \{c,i\} B_i^u,v(m, j) \land B_i^u,v(m, c)$

The timestamps $t_i^j$ are written and read by $P_i$ and no other process. To achieve pairwise normal form, we now make the $t_i^j$ part of the local state of $P_i$. Thus, we replace each local state $r_i$ of $P_i$ by $3^K$ local states, each of which agrees with $r_i$ on the atomic propositions in $\text{AP}_i$. There is one such state for every different assignment of timestamp values to $t_1^i, \ldots, t_k^i$. Call the new process that results $P\bar{P}_i$ and let $PP = (St, PP_1 \| \cdots \| PP_K)$. Note that $PP$ has the same initial states as $P$. Let $r_i^j$ be a local state of $PP_i$, and let $t_1^j, \ldots, t_k^j$ have some values $d_1, \ldots, d_K$ in $r_i^j$. Likewise let $s_i^j$ agree with $s_i$ on the atomic propositions in $\text{AP}_i$, and let $t_1^j, \ldots, t_k^j$ have some values $d_1^j, \ldots, d_K^j$ in $s_i^j$. Then, the set of arcs $\text{ARCS}^u,v_i(r_i^j, s_i^j)$ is defined as follows.

$$\text{ARCS}^u,v_i(r_i^j, s_i^j) \text{ contain } n \text{ arcs, } a'(1), \ldots, a'(n), \text{ where } a'(m) \overset{\text{df}}{=} (r_i^j, \otimes_{j \neq i} \text{BB}^u,v_i(m, j) \rightarrow AA^u,v_i(m, j), s_i^j) \text{ for all } m \in 1, \ldots, n. \text{ In particular, all these arcs start in } r_i^j \text{ and end in } s_i^j. \text{ Also:}$

For all $j \neq i$,

$$\text{BB}^u,v_i(m, j) \overset{\text{df}}{=} B_i^u,v(m, j) \land \text{step}(d_j, tv_j^i, i) = d_j^\prime$$
For all \( j \neq i \),

\[
AA_i^{u,v}(m,j) \overset{df}{=} (tv_j^i := (\ldots, \text{step}(d_j, tv_j^i,i), \ldots); \parallel x \in S_H x_j^i := v(x))
\]

The new conjunct \( \text{step}(d_j, tv_j^i,i) = d_j' \) in effect checks that the values of the timestamps \( t_j^i \) for all \( j \) in the new local states are exactly those that the operation \( \text{step}(t_j^i, t_j^i) \) would return, i.e., those values that would indicate that \( P_i \) has executed later than \( P_j \). The timestamp vector \( tv_j^i \) can now be updated correctly without violating pairwise normal form, since the update can be performed using the \( d_j \) values, which are constants, and the \( tv_j^i,i \), which are shared pairwise between \( P_i \) and \( P_j \), and are therefore permitted by pairwise normal form.

Let \( M_{PP} = (St_P, S_P, R_{PP}) \) be the state-transition diagram of \( PP \). Note that \( PP \) and \( P \) have the same initial states, and the same global states, by definition.

**Theorem 10** \( M_P \sim M_{PP} \)

**Proof of Theorem 10** Proof. Let \( (r,i,s) \in R_P \). \( (r,i,s) \) results from executing an arc \( ARC_i^{u,v} \). Hence \( B_i^{u,v} \) is true in state \( r \). By Proposition 8, some \( B_i^{u,v}(m) \) is true in state \( r \). Hence \( \bigwedge_{j \neq i} B_i^{u,v}(m,j) \) is true in state \( r \), by Proposition 9.

Now let \( r', s' \) be the states in \( M_{PP} \) that correspond to states \( r,s \) in \( M_P \), that is \( r' \) and \( r \) agree on all atomic propositions and shared variables (including timestamps) and likewise \( s \) and \( s' \).

Let \( r_i' = r_i'[i], s_i' = s_i'[i] \). Let \( t_i, \ldots, t_K \) have values \( d_1, \ldots, d_K \) in \( r_i' \) (and hence also in \( r' \)), and values \( d_1', \ldots, d_K' \) in \( s_i' \) (and hence also in \( s' \)). \( (r,r') \) are essentially different ways of referring to the same state, to indicate whether the containing structure is \( M_P \) or \( M_{PP} \), and likewise \( s,s' \).

Since \( (r,i,s) \) results from executing \( ARC_i^{u,v} \), \( \text{step}(d_j, tv_j^i,i) = d_j' \) must hold, since the action \( A_i^{u,v} \) of \( ARC_i^{u,v} \) contains the assignment \( t_j^i := \text{step}(t_j^i, tv_j^i,j) \). Hence \( \bigwedge_{j \neq i} B_i^{u,v}(m,j) \) is true in state \( r' \). Thus, arc \( a'(m) \) of the set \( ARCS_i^{u,v}(r_i', s_i') \) is enabled in state \( r' \). Execution of \( a'(m) \) in state \( r' \) leads to state \( s' \), by definition of \( AA_i^{u,v}(m,j) \). Hence \( (r',i,s') \in R_{PP} \).

Now let \( (r',i,s') \in R_{PP} \). \( (r',i,s') \) results from executing an arc \( a'(m) \) of some set \( ARCS_i^{u,v}(r_i', s_i') \), where \( r_i' = r_i'[i], s_i' = s_i'[i] \). We can run the previous argument “backwards” to show that \( ARC_i^{u,v} \) is enabled in state \( r \) of \( M_P \), and its execution results in state \( s \) of \( M_P \). Hence \( (r,i,s) \in R_P \).

We have in fact showed that \( R_P = R_{PP} \), i.e., that the structures \( M_P \) and \( M_{PP} \) are identical. Hence they are certainly bisimilar.

**Corollary 11** \( M_Q \sim M_{PP} \)

**Proof.** Immediate from Proposition 3, Corollary 7 and Theorem 10, along with the transitivity of bisimulation.

Since \( PP \) is in pairwise normal form by construction, our main result follows immediately:

**Theorem 12** Let \( Q \) be any finite-state concurrent program. Then there exists a concurrent program \( PP \) such that (1) the global state transition diagrams of \( Q \) and \( PP \) are bisimilar, and (2) \( PP \) is in pairwise normal form.

Our result shows that \( PP \) and \( Q \) have essentially the same behavior, since strong bisimulation is the strongest notion of equivalence between concurrent programs. A consequence of our result is that \( PP \) and \( Q \) satisfy the same specifications, for many logics of programs. Recall that \( M_{PP} \) and \( M_Q \) are the global state transition diagrams of \( P \) and \( Q \), respectively. Let \( f \) be a formula of the temporal logic CTL* [10], and define \( M_Q, u \models f \) to mean \( \forall u \in St_Q : M_Q, u \models f \), and \( M_{PP}, s \models f \) to mean \( \forall s \in St_P : M_P, s \models f \), where \( M_Q, u \models f \) and \( M_{PP}, s \models f \) refer to the usual satisfaction relation of CTL* [10]. Then we have:

**Corollary 13** Let \( f \) be a formula of CTL*. Then \( M_Q \models f \) iff \( M_{PP} \models f \).
Proof. Immediate from Corollary 11 and Theorem 14 in [7, chapter 11].

We could easily establish similar results for other logics, such as the mu-calculus.

4.4 Complexity Results

For a single process $Q_i$, define $|Q_i|$, the size of $Q_i$, to be the size of the representation of $Q_i$ using a standard complexity-theoretic encoding, i.e., enumeration for sets, character strings for guards and actions etc. Likewise define $|PP|$. Define $|Q|$, the size of $Q$, to be $|St_Q| + |Q_1| + \cdots + |Q_K|$, and $|PP|$, the size of $PP$, to be $|St_P| + |PP_1| + \cdots + |PP_K|$.

Define the size of a Kripke structure to be the number of states plus the number of transitions.

Theorem 14 $|PP|$ is in $O(K \exp(|Q| + K))$.

Proof. $|M_Q|$ is in $O(\exp(|Q|))$ by Definition 2. $|M'_Q|$ is in $O(K \cdot |M_Q|)$, since each state and transition in $M_Q$ is “replicated” at most $K$ times. So $|M'_Q|$ is in $O(K \exp(|Q|))$.

For each transition in $M'_Q$, $PP$ contains a number of arcs that is in $O(\exp(K))$. Hence $|PP|$ is in $O(|M'_Q| \cdot \exp(K))$, and so $|PP|$ is in $O(K \cdot \exp(|Q|) \cdot \exp(K))$. Thus $|PP|$ is in $O(K \exp(|Q| + K))$. □

5 Related Work

It has been long known that a multiple-reader multiple writer atomic register can be implemented using a set of single-reader single-writer registers, and three are many such atomic register constructions in the literature [6, chapter 10]. Since, by definition, a single-reader single-writer register is shared by two processes, these constructions may seem to subsume our result. However, the atomic register constructions do not respect pairwise normal form. For example, they may involve the operation of taking the maximum over a set of single-reader single-writer registers that involve many different pairs of processes. This direct use of register values corresponding to many different pairs, in computing a single expression value, is a direct violation of pairwise normal form.

6 Conclusions and Future Work

We showed that any finite-state shared memory concurrent program can be rewritten in pairwise normal form, up to strong bisimulation, for a high-atomicity model of concurrent computation. A topic of future work is to establish a similar result in a low-atomicity model, for example that presented in [3]. Our results have significant implications for the efficient synthesis and model-checking of finite-state shared memory concurrent programs. In particular, they show that the approaches of [1, 2, 5] do not sacrifice any expressive power by restricting attention to pairwise normal form.

References

[1] P. C. Attie. Synthesis of large concurrent programs via pairwise composition. In CONCUR '99: 10th International Conference on Concurrency Theory, number 1664 in LNCS. Springer-Verlag, Aug. 1999.

[2] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. ACM Trans. Program. Lang. Syst., 20(1):51–115, Jan. 1998.

[3] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems for an atomic read/write model of computation. ACM Trans. Program. Lang. Syst., 23(2):187–242, Mar. 2001. Extended abstract appears in Proceedings of the 15th ACM Symposium of Principles of Distributed Computing (PODC), Philadelphia, May 1996, 111–120.

[4] P. C. Attie. Synthesis of large dynamic concurrent programs from dynamic specifications. Technical report, Northeastern University, Boston, MA, 2003. Available at http://www.ccs.neu.edu/home/attie/pubs.html.
[5] P.C. Attie and H. Chockler. Efficiently verifiable sufficient conditions for deadlock-freedom of large concurrent programs. Technical report, Northeastern University, Boston, MA, 2004. Available at http://www.ccs.neu.edu/home/attie/pubs.html.

[6] H. Attiya and J. Welch. Distributed Computing. McGraw Hill, London, UK, 1998.

[7] E.M. Clarke, O. Grumberg, and D.A. Peled. Model Checking. MIT Press, Cambridge, MA, 1999.

[8] Dolev D. and Shavit N. Bounded concurrent time-stamping. SIAM J. Comput., 26(2):418–455, Apr. 1997.

[9] E. W. Dijkstra. A Discipline of Programming. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

[10] E. A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, Handbook of Theoretical Computer Science, volume B, Formal Models and Semantics. The MIT Press/Elsevier, Cambridge, Mass., 1990.

[11] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Program., 2:241 – 266, 1982.