Finite-state concurrent programs can be expressed succinctly in triple normal form

Paul C. Attie

Department of Computer Science, American University of Beirut, Beirut, Lebanon

Abstract

I show that any finite-state shared-memory concurrent program P can be transformed into *triple normal form*: all variables are shared between exactly three processes, and the guards on actions are conjunctions of conditions over this triple-shared state. My result is constructive, since the transformation that I present is syntactic, and is easily implemented.

If (1) action guards are in disjunctive normal form, or are short, i.e., of size logarithmic in the size of P, and (2) the number of shared variables is logarithmic in the size of P, then the triple normal form program has size polynomial in the size of P, and the transformation is computable in polynomial time.

Keywords: finite-state concurrent programs, expressive completeness, atomic registers

1. Introduction

I present a transformation that starts with a finite-state shared-memory concurrent program P and produces a strongly bisimilar concurrent program $\mathcal{P}$ that is in *triple normal form*: (1) $\mathcal{P}$ uses only 3-process shared variables, and (2) every process $\mathcal{P}_i$ in $\mathcal{P}$ shares and updates state with other processes on a triple-by-triple basis. That is, $\mathcal{P}_i$ shares and updates state with $\mathcal{P}_j$ and $\mathcal{P}_k$, and also with $\mathcal{P}_{j'}$ and $\mathcal{P}_{k'}$. The overall actions of $\mathcal{P}_i$ are "conjunctions" of actions over $\mathcal{P}_i, \mathcal{P}_j, \mathcal{P}_k$ on one hand, and $\mathcal{P}_i, \mathcal{P}_{j'}, \mathcal{P}_{k'}$ on the other hand. Likewise for all other triples that $\mathcal{P}_i$ is involved in.

The transformation preserves the structure of P, both syntactically and semantically. Each action in $\mathcal{P}$ is derived directly from a particular action in P, and the global state transition diagram of $\mathcal{P}$ is strongly bisimilar to the global state transition diagram of P. The transformation requires that action guards be first rewritten in disjunctive normal form, and so may incur exponential complexity in the size of the guards. In practice however, guards in concurrent

Email address: paul.attie@aub.edu.lb (Paul C. Attie)

programs tend to be short. Also, the transformation is exponential in the number m of shared variables of P, and so is polynomial only if m is logarithmic in the size of P. This limitation on the number of shared variables may seem restrictive, but it applies only to variables that are both read and written by more than one process. A variable that is written by one process and read by several is not "shared" in my model, and I implement it as part of the local state of the process which writes to it. For example, many mutual exclusion algorithms have a single shared "turn" variable, and many local "flag" variables.

2. Model of concurrent computation

A finite-state shared-memory concurrent program $P = P_1 \| \cdots \| P_K$ consists of a finite number K of fixed sequential processes $P_1, \ldots, P_K$ running in parallel. With every process P_i, $1 \leqslant i \leqslant K$, associate a single unique index i. Each P_i is a *synchronization skeleton* [4], i.e., a directed multigraph where each node is a *local state* of P_i, which is labeled by a unique name s_i, and where each arc is labeled with a *guarded command* [3] $B_i \to A_i$ consisting of a guard B_i and corresponding action A_i. I write such an arc as the tuple $(s_i, B_i \to A_i, s_i')$, where s_i is the source node and s_i' is the target node.

Let S_i denote the set of local states of P_i. With each P_i, associate a finite set AP_i of *atomic propositions*, and a mapping $V_i : S_i \to (AP_i \to \{true, false\})$ from local states of P_i to boolean valuations over AP_i: for $p_i \in AP_i$, $V_i(s_i)(p_i)$ is the value of p_i in s_i. Without loss of generality, assume $V_i(s_i) \neq V_i(s_i')$ when $s_i \neq s_i'$, i.e., different local states have different valuations. As P_i executes transitions and changes its local state, the atomic propositions in AP_i are updated, since the valuation changes. Atomic propositions are not shared: $AP_i \cap AP_j = \emptyset$ when $i \neq j$. Any process P_j, $j \neq i$, can read (via guards) but not update the atomic propositions in AP_i. Define the set of all atomic propositions $AP = AP_1 \cup \cdots \cup AP_K$. There is also a finite set $SH = \{x_1, \ldots, x_m\}$ of shared variables, which can be read and written by every process. Each x_ℓ, $1 \leqslant \ell \leqslant m$, takes values from some finite domain D_ℓ. For any arc $(s_i, B_i \to A_i, s_i')$ of process P_i, the guard B_i is a propositional formula over atomic propositions in $AP - AP_i$ and shared variable tests of the form $x_\ell = c$ where $c \in D_\ell$ is a constant. The atomic propositions in AP_i are referenced implicitly by the choice of start state s_i. The action A_i is a multiple assignment that updates the shared variables.

A *global state* s is a tuple of the form $s = (s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m)$ where s_i is the current local state of P_i and $v_1, \ldots, v_m$ is a list giving the values of $x_1, \ldots, x_m$ in s, respectively. For a propositional formula B, define $s(B)$ as usual: $s(\text{"}x = c\text{"}) = true$ iff $s(x) = c$, $s(B1 \wedge B2) = s(B1) \wedge s(B2)$, $s(\neg B1) = \neg s(B1)$. If $s(B) = true$, write $s \models B$. Suppose that P_i contains an arc $(s_i, B_i \to A_i, s_i')$ and that $s \models B_i$. Then, a possible 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 $x_1, \ldots, x_m$ resulting from the execution of action A_i. The set of all (and only) such triples (s, i, s') constitutes the *next-state relation* of program P. In this case, we say that $(s_i, B_i \to A_i, s_i')$ is *enabled* in s. Thus, at each step of the computation, a process with an enabled arc is nondeterministically selected to

be executed next, i.e., I model parallelism by nondeterministic interleaving of the "atomic" transitions of the individual processes P_i. Atomic transitions have a large grain of atomicity; evaluation of B_i, execution of A_i, and change of local state of P_i from s_i to s_i', must all occur as a single indivisible transition.

Definition 1. *Let $s = (s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m)$ be a global state. For atomic proposition $p_i \in AP_i$, $1 \leqslant i \leqslant K$, $s(p_i) \triangleq V_i(s_i)(p_i)$, and for shared variable x_ℓ, $1 \leqslant \ell \leqslant m$, $s(x_\ell) \triangleq v_\ell$. Also $s\restriction i \triangleq s_i$, i.e., $s\restriction i$ is the local state of P_i in s, and $s\restriction AP \triangleq \{p \in AP \mid s(p) = true\}$ i.e., $s\restriction AP$ is the set of atomic propositions that are true in state s.*

Let St_P be a given set of initial ("start") states in which computations of P can begin. A *computation path* of P is a sequence of states whose first state is in St_P and where each successive pair of states (together with some process index i) are related by the next-state relation. A state is *reachable* iff it lies on a computation path. I re-define a concurrent program $P = (St_P, P_1 \parallel \cdots \parallel P_K)$ to be the parallel composition of K sequential processes, $P_1, \ldots, P_K$, together with a set St_P of initial states.

Definition 2 (Global state transition diagram). *The* global state transition diagram *generated by concurrent program $P = (St_P, P_1 \parallel \cdots \parallel P_K)$ is a Kripke structure $M = (St_P, S, R)$ as follows:*

1. *S is the set of all reachable global states of P.*
2. *R is the next-state relation given above, and restricted to S.*

In the sequel, I use "GSTD" for "global state transition diagram". The semantics of a concurrent program is given by its GSTD, and I define two concurrent programs to be strongly bisimilar iff their GSTD's are.

Definition 3 (Strong bisimulation). *Let $M = (St, S, R)$ and $M' = (St', S', R')$ be two Kripke structures with the same underlying set AP of atomic propositions. A relation $B \subseteq S \times S'$ is a* strong bisimulation *between M and M' iff, whenever $B(s, s')$, then (1) $s\restriction AP = s'\restriction AP$, (2) if $(s, i, u) \in R$ then $\exists u' : (s', i, u') \in R' \wedge B(u, u')$, and (3) if $(s', i, u') \in R'$ then $\exists u : (s, i, u) \in R \wedge B(u, u')$. Define $M \sim M'$, (M and M' are* strongly bisimilar*) iff there exists a strong bisimulation $B \subseteq S \times S'$ between M and M' 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. Triple normal form

Let G_1, G_2 be guarded commands, and let $\otimes$ be a binary infix operator on guarded commands. The operational semantics of $G_1 \otimes G_2$ is that both G_1 and G_2 are executed, that is, the guards of both G_1 and G_2 hold at the same time, and the actions of G_1 and G_2 are executed simultaneously, as a single parallel assignment statement. $\otimes$ is idempotent: $G_1 \otimes G_1 = G_1$. When $G_1 \neq G_2$, the semantics of $G_1 \otimes G_2$ is well-defined only if there are no conflicting assignments to shared variables in G_1 and G_2. This is always the case for the programs that I consider. As $\otimes$ is clearly commutative and associative, I use an indexed version $\bigotimes$ of $\otimes$. See [2] for a detailed discussion of $\otimes$.

Process index set notation. I use $[K]$ for the set $\{1, \ldots, K\}$, and i, j, k, ℓ and primed variants as process indices ranging implicitly over $[K]$. Other restrictions on the range (i.e., in quantifications $\bigwedge, \bigvee, \bigotimes$) are given explicitly, e.g., $\bigwedge_{j\,:\,j \neq \ell}$ also restricts j to be not equal to ℓ. Define $\mathsf{T}(i,j,k) \triangleq i \in [K] \wedge j \in [K] \wedge k \in [K] \wedge i \neq j \wedge j \neq k \wedge k \neq i$, i.e., $\mathsf{T}(i,j,k)$ is the set of triples in $[K]$ with distinct elements. For example, in $\bigotimes_{j\,:\,\mathsf{T}(i,j,\ell)}$, j ranges over all indices in $[K]$ that are different than i and ℓ (given $i \neq \ell$), and in $\bigotimes_{j,k\,:\,\mathsf{T}(i,j,k)}$, j and k range over all pairs of indices in $[K]$ that are different than i and also different than each other. Also, I use $j, k \neq \ell$ to abbreviate $j \neq \ell \wedge k \neq \ell$.

Definition 4 (Triple normal form). *A concurrent program $P = (St_P, P_1 \| \cdots \| P_K)$ is in* triple normal form *iff the following four conditions all hold:*

1. *every arc of every process P_i has the form $(s_i, \bigotimes_{j,k\,:\,\mathsf{T}(i,j,k)} B_i^{jk} \rightarrow A_i^{jk}, s_i')$, where $B_i^{jk} \rightarrow A_i^{jk}$ is a guarded command.*
2. *variables are shared in a three-way manner, i.e., for each i, j, k such that $\mathsf{T}(i,j,k)$, there is some set SH_{ijk} of shared variables*
3. *B_i^{jk} can reference only variables in SH_{ijk} and atomic propositions in $AP_j \cup AP_k$, and*
4. *A_i^{jk} can update only variables in SH_{ijk}.*

In the above, the order of subscripts in SH_{ijk} does not matter, i.e., SH_{ijk} and SH_{kji} are the same. Also, the order of superscripts in A_i^{jk}, B_i^{jk} does not matter, i.e., A_i^{jk} and A_i^{kj} are the same, as are B_i^{jk} and B_i^{kj}.

4. The transformation into triple normal form

In the sequel, fix two concurrent programs P and $\mathcal{P}$. $P = (St_P, P_1 \| \cdots \| P_K)$ is a finite-state shared-memory concurrent program as defined in Section 2 above, and $\mathcal{P} = (St_{\mathcal{P}}, \mathcal{P}_1 \| \cdots \| \mathcal{P}_K)$ is the result of applying to P the transformation to triple normal form that I present below.

4.1. Syntactic restrictions

For simplicity, assume that P has exactly one shared variable x, and that arcs in every process P_i of P are labeled with guarded commands in one of these forms: (1) $f \wedge x = c \;\rightarrow\; x := d$, (2) $f \wedge x = c \;\rightarrow\; \mathit{skip}$, (3) $f \;\rightarrow\; x := d$, and (4) $f \;\rightarrow\; \mathit{skip}$, where f is a propositional formula over the atomic propositions in $AP - AP_i$. Call f the *propositional component* of the guard.

4.2. The transformation

The key idea is that $\mathcal{P}$ emulates the operations which P executes on the shared variable x, by using a set of 3-process shared variables x_{ijk} and a set of "last writer" variables lw_{ijk}, where $\mathsf{T}(i,j,k)$. The last writer variables indicate the index of the last process to update x. This enables a process $\mathcal{P}_i$ of $\mathcal{P}$ to find an up-to-date x_{ijk}, whose value is the current value of x. $\mathcal{P}_i$ then executes

an arc, using this x_{ijk}, which emulates a corresponding arc of process P_i of P. Each arc of P_i is thereby emulated by a set of arcs in $\mathcal{P}_i$.

Let AR be some arc in P_i, so AR has one of the forms given above. Let f be the propositional component of AR's guard. Rewrite f in disjunctive normal form (DNF): $f^1 \vee \cdots \vee f^n$, and replace AR in P_i by $AR^1, \ldots, AR^n$, where AR^j is AR with f replaced by f^j. The resulting program generates exactly the same GSTD as P, and so I take it to be P without further elaboration. That is, I assume in the sequel that the propositional component of guards in P is always a conjunction of literals. Rewriting f in DNF can incur an exponential increase in the size of f, but, in practice, guards on actions in concurrent programs tend to be quite short, and so this should not be an issue for realistic programs.

I transform P into $\mathcal{P}$ such that (1) $\mathcal{P}$ is in triple normal form, and (2) the GSTD's of $\mathcal{P}$ and P are strongly bisimilar. The transformation applies to each P_i separately, and actually to each arc in each P_i. That is, I transform every arc of P_i into a corresponding set of arcs in triple normal form. The result is $\mathcal{P}_i$. I also transform each start state in St_P into a corresponding start state for $\mathcal{P}$, giving the set $St_{\mathcal{P}}$ of start states for $\mathcal{P}$. Then $\mathcal{P} = (St_{\mathcal{P}}, \mathcal{P}_1 \| \cdots \| \mathcal{P}_K)$. To effect this transformation, I use the following state variables in $\mathcal{P}$.

- **The atomic propositions in AP_i, $i \in [K]$.** These are written by $\mathcal{P}_i$ and read by all processes. They enable $\mathcal{P}_i$ to emulate the local state of P_i, which is defined by the same set AP_i of atomic propositions. $\mathcal{P}_i$ uses the same propositional valuation mapping V_i that P_i does.

- **A variable lw_{ijk} for every i, j, k such that $\mathsf{T}(i, j, k)$.** These are written and read by $\mathcal{P}_i, \mathcal{P}_j, \mathcal{P}_k$. lw_{ijk} has value in $\{i, j, k\}$, and records which of $\mathcal{P}_i$, $\mathcal{P}_j$, and $\mathcal{P}_k$ most recently assigned to x, i.e., the "last writer" to x. Hence every process $\mathcal{P}_i$ can observe, using the lw variables that it has access to, the relative order of writing to x of each pair $\mathcal{P}_j, \mathcal{P}_k$. This allows $\mathcal{P}_i$ to determine the index ℓ of the last writer to x.

- **A shared variable x_{ijk} for every i, j, k such that $\mathsf{T}(i, j, k)$.** These are written and read by $\mathcal{P}_i, \mathcal{P}_j, \mathcal{P}_k$. The "last writer" $\mathcal{P}_\ell$ emulates $x := d$ by writing d into all variables $x_{ij\ell}$, for all i, j such that $\mathsf{T}(i, j, \ell)$. Any $\mathcal{P}_i$ ($i \neq \ell$) reads all the $x_{ij\ell}$ (as j varies such that $\mathsf{T}(i, j, \ell)$) to find the correct emulated value d of x.[1] If $i = \ell$ (i.e., $\mathcal{P}_i$ itself is the last writer, since it executes two transitions in a row) then $\mathcal{P}_i$ reads all the x_{ijk} (as j, k vary such that $\mathsf{T}(i, j, k)$).

For brevity, I say that "$\mathcal{P}_i$ writes to x" rather than "$\mathcal{P}_i$ emulates a write to x". The order of subscripts does not matter, e.g., x_{ijk} and x_{kji} are the same variable, as are lw_{ijk} and lw_{kji}. Recall that the propositional component f of guards in P_i is a conjunction of literals over $AP - AP_i$, and so write $f = \bigwedge_{j\,:\,j \neq i} f_j$ where f_j is a conjunction of literals over AP_j.

[1] It is simpler to have $\mathcal{P}_i$ read all such $x_{ij\ell}$, as j varies, rather than choose one arbitrarily. They will all have the same value d.

Definition 5 (Transformation of P to $\mathcal{P}$). *For each arc (s_i, G, s_i') of P_i, I add arcs to $\mathcal{P}_i$ as follows:*

1. *if G is $f \wedge x = c \;\rightarrow\; x := d$, then, for all $\ell \neq i$, add arc $(s_i, \mathcal{G}^\ell, s_i')$ to $\mathcal{P}_i$, where $\mathcal{G}^\ell$ is*

$$\bigotimes_{j\,:\,\mathsf{T}(i,j,\ell)} \big(f_\ell \wedge f_j \wedge lw_{ij\ell} = \ell \wedge x_{ij\ell} = c \;\rightarrow\; x_{ij\ell}, lw_{ij\ell} := d, i\big) \;\otimes$$
$$\bigotimes_{j,k\,:\,\mathsf{T}(i,j,k) \wedge j,k \neq \ell} \big(f_j \wedge f_k \;\rightarrow\; x_{ijk}, lw_{ijk} := d, i\big)$$

   Also add arc $(s_i, \mathcal{G}^i, s_i')$ to $\mathcal{P}_i$, where $\mathcal{G}^i$ is

$$\bigotimes_{j,k\,:\,\mathsf{T}(i,j,k)} \big(f_j \wedge f_k \wedge lw_{ijk} = i \wedge x_{ijk} = c \;\rightarrow\; x_{ijk}, lw_{ijk} := d, i\big)$$

2. *if G is $f \wedge x = c \;\rightarrow\; skip$, then, for all $\ell \neq i$, add arc $(s_i, \mathcal{G}^\ell, s_i')$ to $\mathcal{P}_i$, where $\mathcal{G}^\ell$ is*

$$\bigotimes_{j\,:\,\mathsf{T}(i,j,\ell)} \big(f_\ell \wedge f_j \wedge lw_{ij\ell} = \ell \wedge x_{ij\ell} = c \;\rightarrow\; skip\big)$$

   Also add arc $(s_i, \mathcal{G}^i, s_i')$ to $\mathcal{P}_i$, where $\mathcal{G}^i$ is

$$\bigotimes_{j,k\,:\,\mathsf{T}(i,j,k)} \big(f_j \wedge f_k \wedge lw_{ijk} = i \wedge x_{ijk} = c \;\rightarrow\; skip\big)$$

3. *if G is $f \;\rightarrow\; x := d$, then add arc $(s_i, \mathcal{G}, s_i')$ to $\mathcal{P}_i$, where $\mathcal{G}$ is*

$$\bigotimes_{j,k\,:\,\mathsf{T}(i,j,k)} \big(f_j \wedge f_k \;\rightarrow\; x_{ijk}, lw_{ijk} := d, i\big)$$

4. *if G is $f \;\rightarrow\; skip$, then add arc $(s_i, \mathcal{G}, s_i')$ to $\mathcal{P}_i$, where $\mathcal{G}$ is*

$$\bigotimes_{j,k\,:\,\mathsf{T}(i,j,k)} \big(f_j \wedge f_k \;\rightarrow\; skip\big)$$

Finally, remove duplicate arcs which result because the order of process indices does not matter. Also, for each state $s \in St_P$, add to $St_\mathcal{P}$ the state t, where (1) $\bigwedge_i t\restriction i = s\restriction i$, (2) $\bigwedge_{i,j,k\,:\,\mathsf{T}(i,j,k)} t(x_{ijk}) = s(x)$, and (3) $\bigwedge_{j,k\,:\,\mathsf{T}(1,j,k)} t(lw_{1jk}) = 1$. The remaining lw variables can have arbitrary values.

I accommodate several shared variables $x_1, \ldots, x_m$ by introducing a separate set of x_{ijk} and lw_{ijk} variables for each shared variable $x_\ell, 1 \leqslant \ell \leqslant m$. Now each arc in $\mathcal{P}$ must check for the last writer of every $x_1, \ldots, x_m$, leading to K^m possibilities, which makes the size of $\mathcal{P}$ exponential in m. The modifications to Def. 5 are straightforward, and left to the reader.

To illustrate Def. 5, consider a concurrent program $P = P_1 \parallel P_2 \parallel P_3 \parallel P_4$. Each P_i has atomic propositions $AP_i = \{N_i, C_i\}$ and two local states: a neutral state, in which N_i holds, and a critical state, in which C_i holds. Initially, all processes are in their neutral state. P_1, P_2 are "producers" and P_3, P_4 are "consumers". When P_1, P_2 are in their critical state, they produce data items (not modeled). P_3, P_4 consume these items by entering their critical state, and they take alternate turns to do so, except that the first of P_3, P_4 to consume is set by the producer (P_1, P_2) who was most recently in its critical state. There is a single shared variable x, which mediates the entry of P_3, P_4 to their critical states. P_1 has arcs: $(N_1, N_2 \wedge N_3 \wedge N_4 \to \mathit{skip}, C_1)$, $(C_1, \mathit{true} \to x := 3, N_1)$, $(C_1, \mathit{true} \to x := 4, N_1)$. P_2 is similar, with the obvious index substitutions. So P_1, P_2 enter their critical state if no other process is in its critical state, and upon exit, they nondeterministically set x to 3 or 4. P_3 has arcs: $(N_3, N_1 \wedge N_2 \wedge N_4 \wedge x = 3 \to x := 4, C_3)$, $(C_3, \mathit{skip}, N_3)$, and P_4 is similar to P_3. Thus P_3, P_4 enter their critical state if no other process is in its critical state, and the shared variable x gives one of them priority over the other. Upon entry to the critical state, they set priority to the other. For example, applying Def. 5 to the arc $(N_3, N_1 \wedge N_2 \wedge N_4 \wedge x = 3 \to x := 4, C_3)$ of P_3 results in the following arc, for the case when the last writer to x is P_2, i.e., $\ell = 2$:

$$\begin{gathered}\big(N_2 \wedge N_1 \wedge \mathit{lw}_{312} = 2 \wedge x_{312} = 3 \;\to\; x_{312}, \mathit{lw}_{312} := 4, 3\big) \;\otimes \\ \big(N_2 \wedge N_4 \wedge \mathit{lw}_{342} = 2 \wedge x_{342} = 3 \;\to\; x_{342}, \mathit{lw}_{342} := 4, 3\big) \;\otimes \\ \big(N_1 \wedge N_4 \;\to\; x_{314}, \mathit{lw}_{314} := 4, 3\big)\end{gathered}$$

4.3. Correctness of the transformation

In the sequel, fix $M_P = (St_P, S_P, R_P)$ and $M_{\mathcal{P}} = (St_{\mathcal{P}}, S_{\mathcal{P}}, R_{\mathcal{P}})$ to be the GSTD's of $P, \mathcal{P}$, respectively, as per Def. 2.

Proposition 1. *$\mathcal{P}$ is in triple normal form.*

Proof. Immediate from Def. 4 and Def. 5. □

Definition 6. *For $\ell \neq i$, define $\mathit{last}_i(\ell) \triangleq \bigwedge_{j\,:\,\mathsf{T}(i,j,\ell)} \mathit{lw}_{ij\ell} = \ell$. Also $\mathit{last}_i(i) \triangleq \bigwedge_{j,k\,:\,\mathsf{T}(i,j,k)} \mathit{lw}_{ijk} = i$. Finally, $\mathit{last}(\ell) \triangleq \bigwedge_i \mathit{last}_i(\ell)$.*

Then, $\mathit{last}_i(\ell)$ holds when $\mathcal{P}_i$ observes that $\mathcal{P}_\ell$ most recently wrote to x. $\mathit{last}(\ell)$ holds when all processes observe that $\mathcal{P}_\ell$ most recently wrote to x.

Proposition 2. *Let $s \in S_{\mathcal{P}}$. Then for all $i, j \in [K]$, (a) $s \models \mathit{last}_i(\ell) \Rightarrow \mathit{last}_j(\ell)$, and (b) $s \models \mathit{last}_i(\ell) \equiv \mathit{last}(\ell)$.*

Proof. (b) follows from (a) and Def. 6. By Def. 2, s is reachable, so let π be a computation path ending in s. Assume $s \models \mathit{last}_i(\ell)$. Consider first $\ell \neq i \wedge \ell \neq j$. By Def. 6, $s \models \bigwedge_{k\,:\,\mathsf{T}(i,k,\ell)} \mathit{lw}_{ik\ell} = \ell$. By Def. 5, along π, either $\mathcal{P}_\ell$ writes last to x, i.e., after any other process (if any) that writes to x, or $\ell = 1$ and no process writes to x along π. Hence by Def. 5, $s \models \bigwedge_{k\,:\,\mathsf{T}(j,k,\ell)} \mathit{lw}_{jk\ell} = \ell$. Hence $s \models \mathit{last}_j(\ell)$, and (a) is established. The cases $\ell = i$, $\ell = j$ are similar; details are left to the reader. □

Proposition 3. *Let $s \in S_{\mathcal{P}}$. Then $s \models \bigvee_\ell \big(last(\ell) \wedge \bigwedge_{k\,:\,k\neq\ell} \neg last(k)\big)$. That is, $s \models last(\ell)$ for exactly one $\ell \in [K]$.*

Proof. By Def. 2, s is reachable, so let π be a finite computation path ending in s, and let $\mathcal{P}_\ell$ be the process that last wrote to x along π, if any, and if no process wrote to x along π, then let $\mathcal{P}_\ell$ be $\mathcal{P}_1$. By Def. 5, we easily verify $s \models last(\ell)$, since $\mathcal{P}_\ell$ sets $lw_{ij\ell}$ to ℓ for all i, j such that $\mathsf{T}(i, j, \ell)$. Hence $s(lw_{ij\ell}) = \ell$ for all such $lw_{ij\ell}$.

Now suppose that $s \models last(k)$ for some $k \neq \ell$. By Def. 6, $s(lw_{ijk}) = k$ for all i, j such that $\mathsf{T}(i, j, k)$. Hence $s(lw_{ik\ell}) = \ell$ and $s(lw_{ik\ell}) = k$ for all i such that $\mathsf{T}(i, k, \ell)$. This is a contradiction, and so $s \models \neg last(k)$. □

Theorem 4. *$M_P \sim M_{\mathcal{P}}$, i.e., M_P and $M_{\mathcal{P}}$ are strongly bisimilar.*

Proof. Let $s \in S_P, t \in S_{\mathcal{P}}$. Define $s \bowtie t$ iff (1) $s{\restriction}AP = t{\restriction}AP$ and (2) $s \models x = c$ iff $t \models \bigvee_\ell \big(last(\ell) \wedge \big(\bigwedge_{i,j\,:\,\mathsf{T}(i,j,\ell)} x_{ij\ell} = c\big)\big)$.

From Def. 5, for each initial state of P, i.e., $s \in St_P$, there is a $t \in St_{\mathcal{P}}$ such that $s \bowtie t$ (with $\ell = 1$), and vice-versa. It remains to show that $\bowtie$ is a bisimulation. Clause (1) of Def. 3 holds since $s{\restriction}AP = t{\restriction}AP$ by definition of $\bowtie$. Consider an arbitrary pair of states s of M_P and t of $M_{\mathcal{P}}$ such that $s \bowtie t$.

To establish Clause (2) of Def. 3, let (s, i, s') be some transition of M_P. This transition results from the execution of some arc $AR = (s_i, G, s'_i)$ in P_i, where $s_i = s{\restriction}i$ and $s'_i = s'{\restriction}i$. Consider first the case when G has the form $f \wedge x = c \;\rightarrow\; x := d$, where $f = \bigwedge_{j\,:\,j\neq i} f_j$, and each f_j is over AP_j only. Hence $s \models f \wedge x = c$, since AR is enabled in s.

By Prop. 3, $t \models last(\ell) \wedge \bigwedge_{k\,:\,k\neq\ell} \neg last(k)$ for some $\ell \in [K]$. Suppose $\ell \neq i$, and let $(s_i, \mathcal{G}^\ell, s'_i)$ be the arc in $\mathcal{P}_i$ generated for this ℓ from (s_i, G, s'_i), according to Def. 5. By $s \bowtie t$ and Prop. 3, I have $t \models last(\ell) \wedge \big(\bigwedge_{j\,:\,\mathsf{T}(i,j,\ell)} x_{ij\ell} = c\big)$, since $t \models \neg last(k)$ for all $k \neq \ell$. By $t \models last(\ell)$ and Def. 6, $t \models \bigwedge_{j\,:\,\mathsf{T}(i,j,\ell)} lw_{ij\ell} = \ell$. Since $s \models f$, I have $s \models \bigwedge_{j\,:\,j\neq i} f_j$. By $s \bowtie t$, I have $s{\restriction}AP = t{\restriction}AP$. Hence $t \models \bigwedge_{j\,:\,j\neq i} f_j$ since satisfaction of f_j depends only on $s{\restriction}AP$, $t{\restriction}AP$, respectively. I conclude $t \models \big(\bigwedge_{j\,:\,\mathsf{T}(i,j,\ell)} lw_{ij\ell} = \ell \wedge x_{ij\ell} = c\big) \wedge \bigwedge_{j\,:\,j\neq i} f_j$. Hence $(s_i, \mathcal{G}^\ell, s'_i)$ is enabled in t.

Let t' be the state resulting from execution of $(s_i, \mathcal{G}^\ell, s'_i)$ in t, so that (t, i, t') is a transition of $M_{\mathcal{P}}$. It remains to show $s' \bowtie t'$. $s'{\restriction}AP = t'{\restriction}AP$ follows from $s{\restriction}AP = t{\restriction}AP$ and the observation that both arcs change the local state of process i from s_i to s'_i. By construction of $(s_i, \mathcal{G}^\ell, s'_i)$, $t' \models last(i) \wedge \big(\bigwedge_{j,k\,:\,\mathsf{T}(i,j,k)} x_{ijk} = d\big)$. Since $s'(x) = d$, it follows that $s' \bowtie t'$. The case of $\ell = i$ is argued similarly, using the definition of $last_i(i)$. The other three forms for G are argued in a similar but simpler manner, since x is not read, or not written, or neither.

To establish Clause (3) of Def. 3, let (t, i, t') be some transition of $M_{\mathcal{P}}$. By Def. 5, (t, i, t') arises from the execution by $\mathcal{P}_i$ of some arc $(s_i, \mathcal{G}^\ell, s'_i)$, where $s_i = t{\restriction}i$, $s'_i = t'{\restriction}i$. Suppose that $\ell \neq i$. Let (s_i, G, s'_i) be the arc in P_i from which $(s_i, \mathcal{G}^\ell, s'_i)$ is generated, according to Def. 5. Consider first the case where G has the form $f \wedge x = c \;\rightarrow\; x := d$. By Def. 5, I have

$t \models \big(\bigwedge_{j\,:\,\mathsf{T}(i,j,\ell)} lw_{ij\ell} = \ell \wedge x_{ij\ell} = c\big) \wedge \bigwedge_{j\,:\,j\neq i} f_j$, since $(s_i, \mathcal{G}^\ell, s_i')$ is enabled in t. By Def. 6, $t \models last_i(\ell) \wedge \bigwedge_{j\,:\,\mathsf{T}(i,j,\ell)} x_{ij\ell} = c$. By Prop. 2, $t \models last(\ell)$. By Prop. 3, $t \models last(\ell) \wedge \bigwedge_{k\,:\,k\neq\ell} \neg last(k)$. From $s \bowtie t$ and $t \models last(\ell) \wedge \big(\bigwedge_{k\,:\,k\neq\ell} \neg last(k)\big) \wedge \big(\bigwedge_{j\,:\,\mathsf{T}(i,j,\ell)} x_{ij\ell} = c\big)$, I have $s \models x = c$. From $s \bowtie t$, I have $s{\restriction}AP = t{\restriction}AP$. Since $t \models \bigwedge_{j\,:\,j\neq i} f_j$, I have $s \models \bigwedge_{j\,:\,j\neq i} f_j$, and so $s \models f$. Hence $s \models f \wedge x = c$, and so the arc (s_i, G, s_i') is enabled in state s.

Let s' be the state resulting from execution of (s_i, G, s_i') in s. Since execution of $(s_i, \mathcal{G}^\ell, s_i')$ in t leads to t', I have $t' \models last(i) \wedge \big(\bigwedge_{j,k\,:\,\mathsf{T}(i,j,k)} x_{ijk} = d\big)$. Since execution of (s_i, G, s_i') in s leads to s', I have $s' \models x = d$. Also, $s'{\restriction}AP = t'{\restriction}AP$, since both arcs change the local state of process i from s_i to s_i'. Hence $s' \bowtie t'$. The case of $\ell = i$ is argued similarly. The other three forms for G are argued in a similar but simpler manner, since x is not read, or not written, or neither. □

It is simple to extend Def. 5 and Th. 4 to deal with a more general syntax for concurrent programs, in which shared variable tests are of the form $\mathsf{p}(x)$, a predicate that references x, constants c from the domain of x, and standard arithmetic relations and functions, e.g., $10 \leqslant x\%5 \leqslant 12$. This is because the bisimulation $\bowtie$ maintains correspondence of x and $x_{ij\ell}$, and so $s \bowtie t$ implies $s \models \mathsf{p}(x)$ iff $t \models \mathsf{p}(x_{ij\ell})$. The details are left to the reader.

Let $|P_i|$ be the size of the representation of P_i using a standard encoding, i.e., enumeration for sets, strings for guards and actions etc. Likewise define $|\mathcal{P}_i|$. Define $|P| \triangleq |St_P| + |P_1| + \cdots + |P_K|$, and $|\mathcal{P}| \triangleq |St_\mathcal{P}| + |\mathcal{P}_1| + \cdots + |\mathcal{P}_K|$.

Theorem 5. *If the propositional component of guards of arcs in P is a conjunction of literals, and P has a single shared variable, then $|\mathcal{P}| = O(K^3 * |P|)$.*

Proof. Def. 5 generates, for each arc AR in P_i, at most K arcs in $\mathcal{P}_i$. Each such arc has size at most about K^2 times the size of AR, since (in the worst case) it quantifies over all j, k such that $\mathsf{T}(i, j, k)$. Hence $|\mathcal{P}_i| = O(K^3 * |P_i|)$. Each initial state in St_P has $O(K^3)$ variables x_{ijk} and lw_{ijk} added to it to produce an initial state in $St_\mathcal{P}$, so $|St_\mathcal{P}| = O(K^3 * |St_P|)$, Th. 5 follows. □

If there are m shared variables $x_1, \ldots, x_m$, then we have $O(K^m)$ arcs in $\mathcal{P}_i$ for each arc in P_i, one arc for each set of possible last writers to $x_1, \ldots, x_m$. Hence $|\mathcal{P}| = O(m * K^{m+2} * |P|)$. So $|\mathcal{P}|$ is polynomial in $|P|$ if $m = O(\log |P|)$.

Corollary 6. *Let P be a finite-state shared-memory concurrent program where (1) every guard has a propositional component that, either is in disjunctive normal form, or has size logarithmic in $|P|$, and (2) the number of shared variables is logarithmic in $|P|$. Then there exists a finite-state shared-memory concurrent program $\mathcal{P}$ that is strongly bisimilar to P and is in triple normal form. Furthermore, $|\mathcal{P}|$ is polynomial in $|P|$, and $\mathcal{P}$ can be computed from P in time polynomial in $|P|$, via Def. 5.*

Proof. Immediate from Th. 4, Th. 5, Def. 5, and the above discussion. □

5. Related work and discussion

It has long been known that a multiple-reader multiple-writer atomic register can be implemented using a set of single-reader single-writer registers [5, 6, 7, 8,

9]. However, these atomic register constructions do not subsume my result since they do not respect triple normal form, and do not provide strong bisimulation. I presented in [1] a transformation of any finite-state concurrent program P into a program $\mathcal{P}$ in "pairwise normal form." However, $|\mathcal{P}|$ is exponential in $|P|$.

This paper shows that the power of shared variables can be traded off for synchrony: a program P with "global" guarded commands that read/write variables shared by all processes, can be transformed into a bisimilar program $\mathcal{P}$ in triple normal form: each command is a "synchronous conjunction" ($\otimes$) of several guarded commands, each of which reads/writes variables shared by three processes. Moreover, the transformation is simple, syntactic, and polynomial-time when guards are either short or in DNF (in which case they can be of any size), and the number of shared variables is small.

[1] Attie, P. C., 2016. Finite-state concurrent programs can be expressed in pairwise normal form. Theor. Comput. Sci. 619, 1–31.
URL http://dx.doi.org/10.1016/j.tcs.2015.11.032

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

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

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

[5] Haldar, S., Vidyasankar, K., 1996. Simple extensions of 1-writer atomic variable constructions to multiwriter ones. Acta Informatica 33 (2), 177–202.
URL http://dx.doi.org/10.1007/s002360050040

[6] Israeli, A., Shaham, A., 1992. Optimal multi-writer multi-reader atomic register. In: Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing. PODC '92. ACM, New York, NY, USA, pp. 71–82.
URL http://doi.acm.org/10.1145/135419.135435

[7] Li, M., Tromp, J., Vitányi, P. M. B., Jul. 1996. How to share concurrent wait-free variables. J. ACM 43 (4), 723–746.
URL http://doi.acm.org/10.1145/234533.234556

[8] Li, M., Vitányi, P. M. B., 1992. Optimality of wait-free atomic multiwriter variables. Information Processing Letters 43 (2), 107 – 112.
URL http://www.sciencedirect.com/science/article/pii/002001909290020V

[9] Singh, A. K., Anderson, J. H., Gouda, M. G., Mar. 1994. The elusive atomic register. J. ACM 41 (2), 311–339.
URL http://doi.acm.org/10.1145/174652.174657