Theoretical Computer Science 619 (2016) 1–31

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

# Finite-state concurrent programs can be expressed in pairwise normal form

Paul C. Attie

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

ARTICLE INFO

*Article history:*
Received 13 March 2014
Received in revised form 30 October 2015
Accepted 20 November 2015
Available online 3 December 2015
Communicated by R. van Glabbeek

*Keywords:*
Finite-state concurrent programs
Expressive completeness
Atomic registers
State-explosion

ABSTRACT

We show that any finite-state shared-memory concurrent program can be transformed into *pairwise normal form*: all variables are shared between exactly two processes, and the guards on transitions are conjunctions of conditions over this pairwise shared state. Specifically, if $P$ is a finite-state shared-memory concurrent program, then there exists a finite-state shared-memory concurrent program $\mathcal{P}$ expressed in pairwise normal form such that $\mathcal{P}$ is strongly bisimilar to $P$. Our result is constructive: we give an algorithm for producing $\mathcal{P}$, given $P$.



## 1. Introduction

It is well-known that synchrony increases the power of concurrent programs, e.g., with respect to the ability to solve consensus in the presence of some number and type of faults [15]. In this paper, we investigate the power of synchrony with respect to shared registers (i.e., shared variables) in a concurrent program. We show that, by using a larger grain of synchronous atomic action, we can decrease the power of the shared registers that are used. Specifically, we can replace any number of $K$-reader $K$-writer variables by (a larger number of) 1-reader 1-writer variables.

We present a transformation that starts with a concurrent program $P = P_1 \| \cdots \| P_K$ and produces an equivalent (strongly bisimilar) concurrent program $\mathcal{P} = \mathcal{P}_1 \| \cdots \| \mathcal{P}_K$ in *pairwise normal form*: (1) $\mathcal{P}$ uses only 1-reader 1-writer shared variables, and (2) every process $\mathcal{P}_i$ in $\mathcal{P}$ shares and updates state with other processes on a pairwise basis. That is, $\mathcal{P}_i$ shares and updates state with $\mathcal{P}_j$, and also with $\mathcal{P}_k$. The actions of $\mathcal{P}_i$ are "conjunctions" of "pairwise" actions. Each pairwise action inspects and updates the 1-reader 1-writer variables that are shared between $\mathcal{P}_i$ and one other process $\mathcal{P}_j$. A different pairwise action inspects and updates the 1-reader 1-writer variables that are shared between $\mathcal{P}_i$ and $\mathcal{P}_k$. An action of $\mathcal{P}_i$ is a "conjunction" of such pairwise actions. The global state transition diagram of $\mathcal{P}$ is strongly bisimilar to the global state transition diagram of $P$. Thus, in particular, our transformation does not introduce any extraneous waiting, i.e., waiting that is not originally present in $P$.

Our result has implications for the program synthesis and verification methods developed in [1,3,4]. Those papers present methods for synthesizing concurrent programs in pairwise normal form, and for verifying their deadlock-freedom, safety, and liveness properties. These methods exploit pairwise normal form to avoid *state-explosion*; they work by analyzing the global state transition diagrams of small subsystems of processes (two processes for safety and liveness, and three processes for deadlock-freedom).

*E-mail address:* paul.attie@aub.edu.lb.

http://dx.doi.org/10.1016/j.tcs.2015.11.032
0304-3975/

In principle, one could verify a concurrent program $P$ by using the result of this paper to convert $P$ to $\mathcal{P}$, and then apply the methods of [1,3,4] to $\mathcal{P}$. However, there is an exponential blowup in going from $P$ to $\mathcal{P}$, as $\mathcal{P}$ has size on the order of the size of the global state transition diagram of $P$, and so state-explosion is not avoided here. In practice, and as shown by the examples in [1,3,4], concurrent programs in pairwise normal form would be written manually and then verified, or would be synthesized from temporal logic specifications of the form $\bigwedge_{ij} f_{ij}$, i.e., conjunctions over pairs of processes.

We therefore emphasize that our result is an *expressiveness* result per se, and does not provide the basis for an efficient verification method for concurrent programs. Our result shows that the restriction to pairwise normal form does not, in principle, incur a loss of expressiveness. Hence the methods of [1,3,4] are quite general with respect to finite-state concurrent programs.

Pairwise normal form has two major benefits: (1) it enables the use of the results of [1,3,4] to verify the correctness of concurrent programs without state-explosion, and (2) it provides a notation that is modular and compositional: the interaction between two processes $\mathcal{P}_i$ and $\mathcal{P}_j$ is expressed as a set of "pairwise actions" that are separate from the rest of $\mathcal{P} = \mathcal{P}_1 \| \cdots \| \mathcal{P}_K$. This provides code locality and modifiability [22], since the interaction between $\mathcal{P}_i$ and $\mathcal{P}_j$ can be inspected and modified separately from the other pairwise interactions in $\mathcal{P}$.

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 shared-memory concurrent program $P$ can be transformed into a strongly-bisimilar finite-state shared-memory concurrent program $\mathcal{P}$ that is in pairwise normal form. A running example is included, for illustration. Section 5 presents another example. Section 6 discusses related work, and Section 7 concludes.

## 2. Technical preliminaries

### 2.1. Model of concurrent computation

We consider finite-state shared-memory concurrent programs of the form $P = P_1 \| \cdots \| P_K$ that consist of a finite number $K$ 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 $P_i$, which is labeled by a unique name $s_i$, and where each arc is labeled with a *guarded command* [8] $B_i \to A_i$ consisting of a guard $B_i$ and corresponding action $A_i$. We 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. Each node must have at least one outgoing arc, i.e., a synchronization skeleton contains no "dead ends." This is without loss of generality, since a dead end can be simulated by an arc labeled with the guarded command $\mathit{false} \to \mathit{skip}$. For $K \geq 1$, let $[K]$ denote the set $\{1, \ldots, K\}$ and $[K] \setminus i$ denote the set $\{1, \ldots, K\} - \{i\}$.

Let $S_i$ denote the set of local states of $P_i$. With each $P_i$, we associate a finite set $\mathcal{AP}_i$ of *atomic propositions*, and a mapping $V_i : S_i \to (\mathcal{AP}_i \to \{\mathsf{true}, \mathsf{false}\})$ from local states of $P_i$ to boolean valuations over $\mathcal{AP}_i$: for $p_i \in \mathcal{AP}_i$, $V_i(s_i)(p_i)$ is the value of atomic proposition $p_i$ in $s_i$. Hence, as $P_i$ executes transitions and changes its local state, the atomic propositions in $\mathcal{AP}_i$ are updated, since $V_i(s_i) \neq V_i(s_i')$ in general. Atomic propositions are not shared: $\mathcal{AP}_i \cap \mathcal{AP}_j = \emptyset$ when $i \neq j$. Any process $P_j$, $j \neq i$, can read (via guards) but not update the atomic propositions in $\mathcal{AP}_i$. We define the set of all atomic propositions $\mathcal{AP} = \mathcal{AP}_1 \cup \cdots \cup \mathcal{AP}_K$. There is also a finite set $\mathcal{SH} = \{x_1, \ldots, x_m\}$ of shared variables, which can be read and written by every process. Each $x_\ell$ takes values from some finite domain $D_\ell$, for $\ell = 1, \ldots, m$. These are updated by the action $A_i$ of an arc. Fig. 1 presents an example synchronization skeleton program (taken from [11]) for two-process mutual exclusion. Process $P_i$ $(i = 1, 2)$ has $\mathcal{AP}_i = \{N_i, T_i, C_i\}$, and each local state is labeled with the atomic propositions true in that state. $N_i$ indicates neutral: $P_i$ is engaged in local computation. $T_i$ indicates trying: $P_i$ has requested the critical resource. $C_i$ indicates critical: $P_i$ is in its critical section, and so holds the critical resource. A shared variable $x$ resolves contention when both processes have requested the critical resource and are in their trying states. An incoming arrow indicates the initial local states.

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. We write $s(x_\ell)$ for the value $v_\ell$ that $s$ assigns to $x_\ell$, $\ell = 1, \ldots, m$.

For any arc $(s_i, B_i \to A_i, s_i')$ of some process $P_i$, the guard $B_i$ is a formula which denotes a predicate on global states, and the action $A_i$ is any piece of terminating pseudocode that updates the shared variables. We do not further restrict the syntax of $B_i$ and $A_i$; any computable function and predicate over $\mathcal{AP}$ and $x_1, \ldots, x_m$ can be used. We do however, assume that "simple" functions and predicates are used, so that the cost of evaluating $B_i$ and executing $A_i$ is proportional to their length. This is not a restriction in practice. If $B_i$ holds in global state $s$, we write $s \models B_i$. We write just $A_i$ for $\mathsf{true} \to A_i$ and just $B_i$ for $B_i \to \mathit{skip}$, where *skip* is the empty assignment.

We model parallelism as usual by the nondeterministic interleaving of the "atomic" transitions of the individual processes $P_i$.

**Definition 1** *(Next-state relation).* 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 node $s_i'$ labeled with $B_i \to A_i$. If $B_i$ holds in $s$, 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 the shared variables 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 $P$.