Information Processing Letters 123 (2017) 8–13

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

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

ARTICLE INFO

Article history:
Received 14 July 2016
Received in revised form 17 January 2017
Accepted 16 February 2017
Available online xxxx
Communicated by Krishnendu Chatterjee

Keywords:
Concurrency
Finite-state concurrent programs
Expressive completeness

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.

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 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] $B_i \rightarrow A_i$ consisting of a guard B_i and corresponding action A_i. I write such an arc as the tuple $(s_i, B_i \rightarrow A_i, s_i')$, where s_i is the source node and s_i' is the target node.

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

http://dx.doi.org/10.1016/j.ipl.2017.02.004
0020-0190/

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 $P = (St_P, P_1 \| \cdots \| P_K)$ 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 $P = (St_P, P_1 \| \cdots \| P_K)$ 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 $(s_i, \bigotimes_{j,k\,:\,\mathsf{T}(i,j,k)} B_i^{jk} \to A_i^{jk}, s_i')$, where $B_i^{jk} \to A_i^{jk}$ is a guarded command,