Synthesis of Large Concurrent Programs via Pairwise Composition

Paul C. Attie [*]

School of Computer Science, Florida International University, Miami, FL, USA
`http://www.cs.fiu.edu/scspage/professor/Attie.html`

Abstract. We present a tractable method for synthesizing arbitrarily large concurrent programs from specifications expressed in temporal logic. Our method does not explicitly construct the global state transition diagram of the program to be synthesized, and thereby avoids *state explosion*. Instead, it constructs a state transition diagram for each pair of component processes (of the program) that interact. This "pair-program" embodies all possible interactions of the two processes. Our method proceeds in two steps. First, we construct a pair-program for every pair of "connected" processes, and analyze these pair-programs for desired correctness properties. We then take the "pair processes" of the pair-programs, and "compose" them in a certain way to synthesize the large concurrent program. We establish a "large model" theorem which shows that the synthesized large program inherits correctness properties from the pair-programs.

1 Introduction

We exhibit a method of automatically synthesizing a concurrent program consisting of K sequential processes executing in parallel, from a temporal logic specification, where K is an arbitrarily large natural number. Previous synthesis methods [1, 9, 11, 14–16] all rely on some form of exhaustive state space search, and thus suffer from the *state explosion problem*: synthesizing a concurrent program consisting of K sequential processes, each with about N local states, requires building the global state transition diagram of size at least N^K, in general. We show how to synthesize a large concurrent program by only constructing the product of small numbers of processes, and in particular, the product of a pair of processes, thereby avoiding the exponential complexity in K.

Our method is a significant improvement over the previous literature. For example, the solutions synthesized in [11] and [14] for the mutual exclusion problem were only for two processes; consideration of just three processes made the problem infeasible for hand computation. Also, the examples given in [15, 16] are reactive modules containing only two single-bit variables. Therefore, we are able to overcome the severe limitations previously imposed by state explosion on the applicability of automatic temporal logic synthesis methods.

A crucial aspect of our method is its soundness: what correctness properties of the pair-programs are preserved by the synthesized program? We show that

[*] Supported in part by NSF CAREER Grant CCR-9702616 and AFOSR Grant F49620-96-1-0221.

any formula of the branching time temporal logic ACTL [13] that is expressed over two processes, and contains no nexttime operator, is preserved. In particular, propositional invariants and some temporal leads-to properties of any pair-program also hold of the synthesized large program. (A temporal leads-to property has the following form: if condition 1 holds now, then condition 2 eventually holds. ACTL can express temporal leads-to if condition 1 is purely propositional.)

This paper extends the work of [4], in two important ways: (1) it eliminates the requirement that all pair-programs be isomorphic to each other, which in effect constrains the synthesized program to contain only one type of interaction amongst its component processes, and (2) it extends the set of correctness properties that are preserved from propositional invariants and propositional temporal leads-to properties (i.e., leads-to properties where both conditions are purely propositional) to formulae that can contain arbitrary nesting of temporal modalities. Our examples will demonstrate the utility of this greater generality.

The rest of the paper is as follows. Section 2 presents our model of concurrent computation and Section 3 discusses temporal logic. Section 4 presents the synthesis method, and Section 5 establishes the method's soundness. In Sections 6 and 7 we synthesize solutions to the readers-writers and two-phase commit problems respectively. Section 8 discusses further work and concludes.

2 Model of Concurrent Computation

A concurrent program $P = P_1 \| \cdots \| P_K$ consists of a finite number of fixed sequential processes $P_1, \ldots, P_K$ running in parallel. With every process P_i, we associate a single, unique index, namely i. Two processes are *similar* if and only if one can be obtained from the other by swapping their indices. Intuitively, this corresponds to concurrent algorithms where a single "generic" indexed piece of code gives the code body for all processes.

We use the *synchronization skeleton* model of [11]. The synchronization skeleton of a process P_i is a state-machine where each state represents a region of code that performs some sequential computation and each arc represents a conditional transition (between different regions of sequential code) used to enforce synchronization constraints. For example, a node labeled C_i may represent the critical section of P_i. While in C_i, P_i may increment a single variable, or it may perform an extensive series of updates on a large database. In general, the internal structure and intended application of the regions of sequential code are unspecified in the synchronization skeleton. The abstraction to synchronization skeletons thus eliminates all steps of the sequential computation from consideration.

Formally, the synchronization skeleton of each process P_i is a directed graph where each node s_i is a unique *local state* of P_i, and each arc has a label of the form $\oplus_{\ell \in [1:n]} B_\ell \rightarrow A_\ell$,[1] where each $B_\ell \rightarrow A_\ell$ is a guarded command [7], and $\oplus$ is guarded command "disjunction." For example, in Figure 2 the arc of

[1] $[1:n]$ denotes the integers from 1 to n inclusive.

process WP_j from N_j to T_j is labeled with $N_k \lor C_k \to \mathit{skip} \oplus T_k \to x_{jk} := k$. Roughly, the operational semantics of $\oplus_{\ell \in [1:n]} B_\ell \to A_\ell$ is that if one of the B_ℓ evaluates to true, then the corresponding body A_ℓ can be executed. If none of the B_ℓ evaluates to true, then the command "blocks," i.e., waits until one of the B_ℓ holds.[2] Each node must have at least one outgoing arc, i.e., a skeleton contains no "dead ends," and two nodes are connected by at most one arc in each direction. A *global state* is a tuple of the form $(s_1, \ldots, s_K, v_1, \ldots, v_m)$ where each s_i is the current local state of P_i, and $v_1, \ldots, v_m$ is a list giving the current values of all the shared variables, $x_1, \ldots, x_m$ (we assume these are ordered in a fixed way, so that $v_1, \ldots, v_m$ specifies a unique value for each shared variable). A guard B is a predicate on states, and a body A is a parallel assignment statement that updates the values of the shared variables. If B is omitted from a command, it is interpreted as *true*, and we write the command as A. If A is omitted, the shared variables are unaltered, and we write the command as B.

We model parallelism in the usual way by the nondeterministic interleaving of the "atomic" transitions of the individual synchronization skeletons of the processes P_i. Hence, at each step of the computation, some process with an "enabled" arc is nondeterministically selected to be executed next. Assume that the current state is $s = (s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m)$ and that P_i contains an arc from s_i to s'_i labeled by the command $B \to A$. If B is true in s, then a permissible next state is $(s_1, \ldots, s'_i, \ldots, s_K, v'_1, \ldots, v'_m)$ where $v'_1, \ldots, v'_m$ is the list of updated values for the shared variables produced by executing A in state s. The arc from s_i to s'_i is said to be *enabled* in state s. An arc that is not enabled is *disabled*, or *blocked*. A *(computation) path* is any sequence of states where each successive pair of states is related by the above next-state relation.

3 Temporal Logic

CTL* is a propositional branching time temporal logic [10] whose formulae are built up from atomic propositions, propositional connectives, the universal (A) and existential (E) path quantifiers, and the linear-time modalities nexttime (by process j) X_j, and strong until U. The logic CTL [11] results from restricting CTL* so that every linear-time modality is paired with a path quantifier, and vice-versa. The logic ACTL [13] results from CTL by restricting negation to propositions, and eliminating the existential path quantifier. The linear-time temporal logic PTL [14] results from removing the path quantifiers from CTL*.

Formally, we define the semantics of CTL* formulae with respect to a (K-process) structure $M = (S, R_{i_1}, \ldots, R_{i_K})$ consisting of

- S, a countable set of states. Each state is a mapping from the set $\mathcal{AP}$ of atomic propositions into $\{\mathit{true}, \mathit{false}\}$, and
- $R_i \subseteq S \times \{i\} \times S$, a binary relation on S giving the transitions of process i.

Here $\mathcal{AP} = \{\mathcal{AP}_{i_1}, \ldots, \mathcal{AP}_{i_K}\}$, where $\mathcal{AP}_i$ is the set of atomic propositions that "belong" to process i. Other processes can read propositions in $\mathcal{AP}_i$, but only

[2] This interpretation was proposed by [8].

process i can modify these propositions (which collectively define the local state of process i). We define the logic ACTL$^-$ to be ACTL without the AX_j modality, and the logic ACTL$_{ij}^-$ to be ACTL$^-$ where the atomic propositions are drawn only from $\mathcal{AP}_i \cup \mathcal{AP}_j$.

Let $R = R_{i_1} \cup \cdots \cup R_{i_K}$. A *path* is a sequence of states $(s_1, s_2 \ldots)$ such that $\forall i : (s_i, s_{i+1}) \in R$, and a *fullpath* is a maximal path. $M, s_1 \models f$ (respectively $M, \pi \models f$) means that f is true in structure M at state s_1 (respectively of fullpath π). Also, $M, S \models f$ means $\forall s \in S : M, s \models f$, where S is a set of states. For the full definition of $\models$, see [10,13]. For example, $M, s_1 \models \mathsf{A}f$ iff for every fullpath $\pi = (s_1, s_2, \ldots)$ in M: $M, \pi \models f$; and $M, \pi \models f \mathsf{U} g$ iff there exists i such that $M, \pi^i \models g$ and for all $j \in [1 : (i-1)]$: $M, \pi^j \models f$ (π^i is the suffix starting at the i'th state of π).

We also introduce some additional modalities as abbreviations: $\mathsf{F}f$ (eventually) for $[\mathit{true}\mathsf{U}f]$, $\mathsf{G}f$ (always) for $\neg\mathsf{F}\neg f$, $[f\mathsf{U}_\mathsf{w}g]$ (weak until) for $[f\mathsf{U}g] \vee \mathsf{G}f$, $\overset{\infty}{\mathsf{F}}f$ (infinitely often) for $\mathsf{GF}f$, and $\overset{\infty}{\mathsf{G}}f$ (eventually always) for $\mathsf{FG}f$. We refer the reader to [10] for details in general, and to [13] for details of ACTL.

To guarantee liveness properties of the synthesized program, we use a form of weak fairness. Fairness is usually specified as a linear-time logic (i.e., PTL) formula Φ, and a fullpath is fair iff it satisfies Φ. To state correctness properties under the assumption of fairness, we relativize satisfaction ($\models$) so that only fair fullpaths are considered. The resulting notion of satisfaction, $\models_\Phi$, is defined by [12] as follows: $M, s_1 \models_\Phi \mathsf{A}f$ iff for every Φ-fair fullpath $\pi = (s_1, s_2, \ldots)$ in M: $M, \pi \models f$. Effectively, path quantification is only over the paths that satisfy Φ.

4 The Synthesis Method

We aim to synthesize a large concurrent program $P_{i_1} \| \cdots \| P_{i_K}$ without explicitly generating its global state transition diagram of size exponential in the number of processes K. The specification for a large concurrent program consists of:

1. a binary, irreflexive "interconnection" relation $I_c \subseteq \{i_1, \ldots, i_K\} \times \{i_1, \ldots, i_K\}$ over the set $\{i_1, \ldots, i_K\}$ of process indices, and
2. a mapping *spec* which maps each pair $(i, j) \in I$ to a formula of ACTL$_{ij}^-$ (we use spec_{ij} rather than $\mathit{spec}((i,j))$ to denote this "pair-specification").

We use I to denote the pair (I_c, spec) and abuse terminology by sometimes referring to I as the interconnection relation. Given a specification I, we synthesize an *I-program* $P^I = (S_I^0, P_{i_1}^I \| \ldots \| P_{i_K}^I)$ as follows:

1. For every pair of process indices $(i, j) \in I$, synthesize a *pair-program* $(S_{ij}^0, P_i^j \| P_j^i)$ using spec_{ij} as the specification.
2. "Compose" all the pair-programs to produce P^I.

Since our focus in this article is on avoiding state-explosion, we shall not explicitly address step 1 above. Any synthesis method that produces concurrent programs in the synchronization skeleton notation can be used, e.g., [2,3,11].

S_{ij}^0 is the set of initial states, and P_i^j, P_j^i are the synchronization skeletons for processes i, j, in the pair-program $(S_{ij}^0, P_i^j \| P_j^i)$. We refer to the component processes of a pair-program as *pair-processes.* Note that P_i^j and P_j^i interact by reading each other's local state and by reading/writing a set (call it $\mathcal{SH}_{ij}$) of shared variables.[3] S_I^0 is the set of initial states, and P_i^I is the synchronization skeleton for process i, in the I-program $(S_I^0, P_{i_1}^I \| \dots \| P_{i_K}^I)$. We refer to the component processes P_i^I of an I-program as *I-processes.* We say that P_i^I and P_j^I are *neighbors* when $(i,j) \in I$. We require that every process has at least one neighbor: $\forall i \in \{i_1, \dots, i_K\} : (\exists j : (i,j) \in I)$. We also define $I(i) = \{j \mid (i,j) \in I\}$.

$spec_{ij}$ is the specification for the pair-program $(S_{ij}^0, P_i^j \| P_j^i)$, and defines the interaction of processes i and j. Thus, $spec_{ij}$ is (initially) interpreted and verified over the structure induced by $(S_{ij}^0, P_i^j \| P_j^i)$ *executing in isolation.* Once $(S_{ij}^0, P_i^j \| P_j^i)$ has been composed with all the other pair-programs to yield the I-program, we will show that $spec_{ij}$ also holds for the I-program. Unlike [4], $spec_{ij}$ and $spec_{k\ell}$ (where $\{k,\ell\} \neq \{i,j\}$) can be completely different formulae, whereas in [4] these formulae had to be "similar," i.e., one was obtained from the other by substituting process indices.

Our synthesis method requires that the pair-programs induce the same *local structure* on all common processes. That is, for pair-programs $(S_{ij}^0, P_i^j \| P_j^i)$ and $(S_{ik}^0, P_i^k \| P_k^i)$, we require $\hat{P}_i^j = \hat{P}_i^k$, where $\hat{P}_i^j, \hat{P}_i^k$ result from removing all arc labels from P_i^j, P_i^k respectively.[4] We assume, in the sequel, that this condition holds. Also, all results quoted from [4] have been reverified to hold in our setting, i.e., when the similarity assumptions of [4] are dropped.

We compose pair-programs as follows. Consider first $I = \{(i,j),(j,k),(k,i)\}$, i.e., three pairwise interconnected processes i, j, k, With respect to process i, the proper interaction (i.e., that required to satisfy $spec_{ij}$) between process i and process j is captured by the commands that label the arcs of P_i^j. Likewise, the proper interaction between process i and process k is captured by the arc labels of P_i^k. Hence, in the three-process program P^I (consisting of processes i, j, k), the proper interaction for process i with processes j and k is captured as follows: when process i traverses an arc, the command which labels that arc in P_i^j is executed "simultaneously" with the command which labels the corresponding arc in P_i^k. For example, taking as our specification the mutual exclusion problem, if process i executes a mutual exclusion protocol with respect to both processes j and k, then, when process i enters its critical section, both processes j and k must be outside their own critical sections.

Based on the above, we determine that the synchronization skeleton for process i in P^I (call it P_i^I) has the same basic graph structure as P_i^j and P_i^k, and an arc label in P^I is a "conjunction" of the labels of the corresponding arcs in P_i^j and P_i^k.

[3] The shared variable sets of different pair-programs are disjoint: $\mathcal{SH}_{ij} \cap \mathcal{SH}_{i'j'} = \emptyset$ if $\{i,j\} \neq \{i',j'\}$.

[4] Contrast this with the much more restrictive "process similarity assumption" of [4] which requires that P_i^j can be obtained from $P_{i'}^{j'}$ by substituting i for i' and j for j'. In effect, <u>all</u> processes must have isomorphic local structure <u>and</u> isomorphic arc labels. Thus, all pair-programs are isomorphic—Proposition 6.2.1 of [4].

Generalizing to an arbitrary interconnection relation I, P_i^I has the same basic graph structure as P_i^j, $(\hat{P_i^I} = \hat{P_i^j})$, and an arc label in P_i^I is a "conjunction" of the labels of the corresponding arcs in $P_i^{j_1}, \ldots, P_i^{j_n}$, where $\{j_1, \ldots, j_n\} = I(i)$ are all the neighbors of process i.

We now make some technical definitions. A node (i.e., local state) of P_i^j, P_i^I is a mapping of $\mathcal{AP}_i$ to $\{\mathit{true}, \mathit{false}\}$. We refer to such nodes as i-states. A state of the pair-program $(S_{ij}^0, P_i^j \,\|\, P_j^i)$ is a tuple $(s_i, s_j, v_{ij}^1, \ldots, v_{ij}^m)$ where s_i, s_j are i-states, j-states, respectively, and $v_{ij}^1, \ldots, v_{ij}^m$ give the values of all the variables in $\mathcal{SH}_{ij}$. We refer to states of $(S_{ij}^0, P_i^j \,\|\, P_j^i)$ as ij-states. An ij-state s_{ij} inherits the assignments defined by its component i- and j-states: $s_{ij}(p_i) = s_i(p_i)$, $s_{ij}(p_j) = s_j(p_j)$, where $s_{ij} = (s_i, s_j, v_{ij}^1, \ldots, v_{ij}^m)$, and p_i, p_j are arbitrary atomic propositions in $\mathcal{AP}_i$, $\mathcal{AP}_j$, respectively. A state of $(S_I^0, P_{i_1}^I \,\|\, \ldots \,\|\, P_{i_K}^I)$ is a tuple $(s_{i_1}, \ldots, s_{i_K}, v^1, \ldots, v^n)$, where s_i, $(i \in \{i_1, \ldots, i_K\})$ is an i-state and $v^1, \ldots, v^n$ give the values of all the shared variables of the I-program (i.e., those in $\bigcup_{(i,j)\in I} \mathcal{SH}_{ij}$). We refer to states of an I-program as I-states. An I-state s inherits the assignments defined by its component i-states $(i \in \{i_1, \ldots, i_K\})$: $s(p_i) = s_i(p_i)$, where $s = (s_{i_1}, \ldots, s_{i_K}, v^1, \ldots, v^n)$, and p_i is any atomic proposition in $\mathcal{AP}_i$ $(i \in \{i_1, \ldots, i_K\})$. If $J \subseteq I$, then define J-program, J-state exactly like I-program, I-state resp. but using interconnection relation J instead of I.

The state-to-formula operator $\{\!\mid s_i \mid\!\}$ takes an i-state s_i as argument and returns a propositional formula: $\{\!\mid s_i \mid\!\} = (\bigwedge_{s_i(p_i)=true} p_i) \wedge (\bigwedge_{s_i(p_i)=false} \neg p_i)$, where p_i ranges over the members of $\mathcal{AP}_i$. $\{\!\mid s_i \mid\!\}$ characterizes s_i in that $s_i \models \{\!\mid s_i \mid\!\}$, and $s_i' \not\models \{\!\mid s_i \mid\!\}$ for all $s_i' \neq s_i$. $\{\!\mid s_{ij} \mid\!\}$ is defined similarly (but note that the variables in $\mathcal{SH}_{ij}$ must be accounted for). We define the *state projection operator* $\upharpoonright$. This operator has several variants. First, we define projection onto a single process from both I-states and ij-states: if $s = (s_{i_1}, \ldots, s_{i_K}, v^1, \ldots, v^n)$, then $s\!\upharpoonright\! i = s_i$, and if $s_{ij} = (s_i, s_j, v_{ij}^1, \ldots, v_{ij}^m)$, then $s_{ij}\!\upharpoonright\! i = s_i$. Next we define projection of an I-state onto a pair-program: if $s = (s_{i_1}, \ldots, s_{i_K}, v^1, \ldots, v^n)$, then $s\!\upharpoonright\! ij = (s_i, s_j, v_{ij}^1, \ldots, v_{ij}^m)$, where $v_{ij}^1, \ldots, v_{ij}^m$ are those values from $v^1, \ldots, v^n$ that denote values of variables in $\mathcal{SH}_{ij}$. $s\!\upharpoonright\! ij$ is well defined only when $i\,I\,j$ (i.e., $(i,j) \in I$). Finally, we define projection of an I-state onto a J-program. If $s = (s_{i_1}, \ldots, s_{i_K}, v^1, \ldots, v^n)$, then $s\!\upharpoonright\! J = (s_{j_1}, \ldots, s_{j_L}, v_J^1, \ldots, v_J^m)$, where $\{j_1, \ldots, j_L\}$ is the domain of J, and $v_J^1, \ldots, v_J^m$ are those values from $v^1, \ldots, v^n$ that denote values of variables in $\bigcup_{(i,j)\in J} \mathcal{SH}_{ij}$. $s\!\upharpoonright\! J$ is well defined only when $J \subseteq I$.

Let π be a computation path of P^I. Then, the *path-projection* of π onto $J \subseteq I$ (denoted $\pi\!\upharpoonright\! J$) is obtained as follows. Replace every state s along π by $s\!\upharpoonright\! J$, and then remove all transitions in π that are not by some process in J, coalescing the source and target states of all such transitions (which must be the same, since if $s \xrightarrow{i} t$ and $i \notin \{j_1, \ldots, j_L\}$, then $s\!\upharpoonright\! J = t\!\upharpoonright\! J$).

The above discussion leads to the following definition for our synthesis method, which derives an I-process P_i^I of the I-program $(S_I^0, P_{i_1}^I \,\|\, \ldots \,\|\, P_{i_K}^I)$ from the pair-processes $\{P_i^j \mid j \in I(i)\}$ of the pair-programs $\{(S_{ij}^0, P_i^j \,\|\, P_j^i) \mid j \in I(i)\}$:

Definition 1 (Pairwise Synthesis). *An* I-process P_i^I $(i \in \{i_1, \ldots, i_K\})$ *is derived from the pair-processes* P_i^j, $j \in I(i)$, *as follows:*

P_i^I *contains an arc from* s_i *to* t_i *with label* $\otimes_{j \in I(i)} \oplus_{\ell \in [1:n_j]} B_{i,\ell}^j \rightarrow A_{i,\ell}^j$
iff
$\forall j \in I(i) : P_i^j$ *contains an arc from* s_i *to* t_i *with label* $\oplus_{\ell \in [1:n_j]} B_{i,\ell}^j \rightarrow A_{i,\ell}^j$.

The initial state set S_I^0 *of the* I*-program is derived from the pair-program initial state sets* S_{ij}^0, $(i,j) \in I$, *as follows:*

$$S_I^0 = \{s \mid \forall (i,j) \in I : s{\uparrow}ij \in S_{ij}^0\}.$$

Here $\otimes$ is guarded command "conjunction." The operational semantics of $B_1 \rightarrow A_1 \otimes B_2 \rightarrow A_2$ is that if both the guards B_1, B_2 evaluate to true, then the bodies A_1, A_2 can be executed in parallel. If at least one of B_1, B_2 evaluates to false, then the command "blocks," i.e., waits until both of B_1, B_2 evaluate to true. See [4] for complete definitions of $\oplus, \otimes$. Note that S_I^0 consists of exactly those I-states whose "projections" onto all the pairs in I give initial states of the corresponding pair-program. We assume that the initial-state sets of all the pair-programs are so that there is at least one such I-state, and so S_I^0 is nonempty.

Definition 1 is, in effect, a *syntactic transformation* that can be carried out in linear time and space (in both $(S_{ij}^0, P_i^j \parallel P_j^i)$ and I). In particular, we avoid explicitly constructing the global state transition diagram of $(S_I^0, P_{i_1}^I \parallel \ldots \parallel P_{i_K}^I)$, which is of size exponential in $K = |\{i_1, \ldots, i_K\}|$.

5 Soundness of the Synthesis Method

Let $M_{ij} = (S_{ij}^0, S_{ij}, R_{ij})$ and $M_I = (S_I^0, S_I, R_I)$ be the global state transition diagrams of $(S_{ij}^0, P_i^j \parallel P_j^i)$, $(S_I^0, P_{i_1}^I \parallel \ldots \parallel P_{i_K}^I)$, respectively. S_{ij}^0, S_I^0 are the sets of initial states of M_{ij}, M_I respectively, and S_{ij}, S_I are the sets of all states of M_{ij}, M_I respectively, and $R_{ij} \subseteq S_{ij} \times \{i,j\} \times S_{ij}$, $R_I \subseteq S_I \times \{i_1, \ldots, i_K\} \times S_I$, are the sets of transitions of M_{ij}, M_I respectively. The technical definitions of M_{ij}, M_I in terms of $(S_{ij}^0, P_i^j \parallel P_j^i)$, $(S_I^0, P_{i_1}^I \parallel \ldots \parallel P_{i_K}^I)$ are straightforward and are omitted (Section 2 describes the relevant operational semantics). M_{ij} and M_I can be interpreted as ACTL structures. M_{ij} gives the semantics of $(S_{ij}^0, P_i^j \parallel P_j^i)$ *executing in isolation*, and M_I gives the semantics of $(S_I^0, P_{i_1}^I \parallel \ldots \parallel P_{i_K}^I)$. Our main soundness result below (the large model theorem) relates the ACTL formulae that hold in M_I to those that hold in M_{ij}. We characterize transitions in M_I as compositions of transitions in all the relevant M_{ij}:

Lemma 1. [4] *For all* I*-states* $s, t \in S_I$ *and* $i \in \{i_1, \ldots, i_K\}$, $s \xrightarrow{i} t \in R_I$ iff :

$$\forall j \in I(i) : s{\uparrow}ij \xrightarrow{i} t{\uparrow}ij \in R_{ij} \text{ and}$$
$$\forall j, k \in \{i_1, \ldots, i_K\} - \{i\}, j\,I\,k : s{\uparrow}jk = t{\uparrow}jk.$$

Lemma 2. [4] *Let* $J \subseteq I$. *If* π *is a path in* M_I, *then* $\pi{\uparrow}J$ *is a path in* M_J.

In particular, when $J = \{(i,j)\}$, Lemma 2 forms the basis for our soundness proof, since it relates computations of the synthesized I-program to computations of the pair-programs.

5.1 Deadlock-Freedom and the Wait-For-Graph

The *wait-for-graph* in a particular I-state s contains as nodes every I-process, and every arc whose start state is a component of s. These arcs have an outgoing edge to every I-process which blocks them.

Definition 2 (Wait-For-Graph $W_I(s)$). *Let s be an arbitrary I-state. The* wait-for-graph $W_I(s)$ *of s is a directed bipartite graph, where*

1. *the nodes of $W_I(s)$ are*
   (a) *the I-processes $\{P_i^I \mid i \in \{i_1, \dots, i_K\}\}$, and*
   (b) *the arcs $\{a_i^I \mid i \in \{i_1, \dots, i_K\}$ and $a_i^I \in P_i^I$ and $s{\uparrow}i = a_i^I.start\}$*
2. *there is an edge from P_i^I to every node of the form a_i^I in $W_I(s)$, and*
3. *there is an edge from a_i^I to P_j^I in $W_I(s)$ if and only if $(i,j) \in I$ and $a_i^I \in W_I(s)$ and $s{\uparrow}ij(a_i^I.guard_j) = false$.*

Here $a_i^I.guard_j$ is the conjunct of the guard of arc a_i^I which references the state shared by P_i and P_j (in effect, $\mathcal{AP}_j$ and $\mathcal{SH}_{ij}$). We characterize a deadlock as the occurrence in the wait-for-graph of a *supercycle*:

Definition 3 (Supercycle). *SC is a* supercycle *in $W_I(s)$ if and only if:*

1. *SC is nonempty,*
2. *if $P_i^I \in SC$ then for all a_i^I such that $a_i^I \in W_I(s)$, $P_i^I \longrightarrow a_i^I \in SC$, and*
3. *if $a_i^I \in SC$ then there exists P_j^I such that $a_i^I \longrightarrow P_j^I \in W_I(s)$ and $a_i^I \longrightarrow P_j^I \in SC$.*

Note that this definition implies that SC is a subgraph of $W_I(s)$. In [4], we give a criterion, the *wait-for-graph assumption*, which is evaluated over the product of a small number of processes, thereby avoiding state-explosion. We show there that if the wait-for-graph assumption holds, then $W_I(s)$ cannot contain a supercycle for any reachable state s of M_I. Furthermore, if $W_I(s)$ does not contain a supercycle, then, in state s, there exists at least one enabled arc. These results extend to the setting of this paper.

5.2 Liveness

To assure liveness properties of the synthesized I-program, we assume a form of weak fairness. Let $CL(f)$ be the set of all subformulae of f, including f itself. Let ex_i be an assertion that is true along a transition in a structure iff that transition results from executing process i. Let en_i hold in an I-state s iff P_i^I has some arc that is enabled in s. Our fairness criterion is the conjunction of *weak blocking fairness* and *weak eventuality fairness* (given below) and is defined as a formula of the linear time temporal logic PTL [14].

Definition 4 (Sometimes-Blocking, blk_i^j, blk_i). *An i-state s_i is* sometimes-blocking *in M_{ij} if and only if:*

$$\exists s_{ij}^0 \in S_{ij}^0 : M_{ij}, s_{ij}^0 \models \mathsf{EF}(\ \{\!|s_i|\!\} \wedge (\exists a_j^i \in P_j^i : (\{\!|a_j^i.start|\!\} \wedge \neg a_j^i.guard))\ ).$$

Also, $blk_i^j \overset{\mathrm{df}}{=} (\bigvee \{\!|s_i|\!\} : s_i$ is sometimes-blocking in $M_{ij})$, and $blk_i \overset{\mathrm{df}}{=} \bigvee_{j \in I(i)} blk_i^j$.

Thus, a sometimes-blocking state is an i-state s_i such that there exists a reachable ij-state s_{ij} in M_{ij} satisfying $s_{ij}{\uparrow}i = s_i$ and in which P_i blocks some arc a^i_j of P_j. $a^i_j.start$ is the start state of a^i_j, and $a^i_j.guard$ is its guard.

Definition 5 (Weak Blocking Fairness Φ_b).

$$\Phi_b \overset{\text{df}}{=} \bigwedge_{i \in \{i_1,\ldots,i_K\}} \overset{\infty}{\mathsf{G}}(blk_i \wedge en_i) \Rightarrow \overset{\infty}{\mathsf{F}} ex_i.$$

Weak blocking fairness requires that a process that is continuously enabled and in a sometimes-blocking state is eventually executed.

Definition 6 (Pending Eventuality, pnd_{ij}). *Let $(i,j) \in I$. An ij-state s_{ij} has a* pending eventuality *if and only if:*

$$\exists f_{ij} \in CL(spec_{ij}) : M_{ij}, s_{ij} \models \neg f_{ij} \wedge \mathsf{AF} f_{ij}.$$

Also, $pnd_{ij} \overset{\text{df}}{=} (\bigvee \;\{\!|s_{ij}|\!\} : s_{ij}$ has a pending eventuality$)$.

In other words, s_{ij} has a pending eventuality if there is a subformula of the pair-specification $spec_{ij}$ which does not hold in s_{ij}, but is guaranteed to eventually hold along every fullpath of M_{ij} that starts in s_{ij}.

Definition 7 (Weak Eventuality Fairness Φ_ℓ).

$$\Phi_\ell \overset{\text{df}}{=} \bigwedge_{(i,j)\in I}(\overset{\infty}{\mathsf{G}} en_i \vee \overset{\infty}{\mathsf{G}} en_j) \wedge \overset{\infty}{\mathsf{G}} pnd_{ij} \Rightarrow \overset{\infty}{\mathsf{F}}(ex_i \vee ex_j).$$

Weak eventuality fairness requires that if an eventuality is continuously pending, and one of P^I_i or P^I_j is continuously enabled, then eventually one of them will be executed. Our overall fairness notion Φ is then the conjunction of weak blocking and weak eventuality fairness: $\Phi \overset{\text{df}}{=} \Phi_b \wedge \Phi_\ell$.

Definition 8 (Liveness Condition). *For every $(i,j) \in I$:*

$$M_{ij}, S^0_{ij} \models \mathsf{AGA}(\mathsf{G} ex_i \Rightarrow \overset{\infty}{\mathsf{G}} aen_j),$$

where $aen_j \overset{\text{df}}{=} \forall a^i_j \in P^i_j \;:\; (\{\!|a^i_j.start|\!\} \Rightarrow a^i_j.guard)$.

aen_j means that every arc of P^i_j whose start state is a component of the current global state s is also enabled in s. The liveness condition requires, for every pair-program $(S^0_{ij}, P^j_i \,\|\, P^i_j)$, when executing in isolation, that if P^j_i can execute continuously along some path, then there exists a suffix of that path along which P^j_i does not block any arc of P^i_j. Given the liveness condition and the absence of deadlocks and the use of Φ-fair scheduling, we can show that one of P^I_i or P^I_j is guaranteed to be executed from any state of the I-program whose ij-projection has a pending eventuality.

Lemma 3 (Progress). *Let $(i,j) \in I$, and let s be an arbitrary reachable I-state. If*

1. *the liveness condition holds, and*
2. *for every reachable I-state u, $W_I(u)$ is supercycle-free, and*
3. *$M_{ij}, s{\uparrow}ij \models \neg h_{ij} \wedge \mathsf{AF} h_{ij}$ for some $h_{ij} \in CL(spec_{ij})$, then*

$$M_I, s \models_\Phi \mathsf{AF}(ex_i \vee ex_j).$$

5.3 The Large Model Theorem

The large model theorem establishes the soundness of our synthesis method. It states that any subformula of pair-specification $spec_{ij}$ which holds in the ij-projection of an I-state s also holds in s itself. That is, correctness properties satisfied by a pair-program executing in isolation also hold in the I-program.

Theorem 1 (Large Model). *Suppose $M_{ij}, S^0_{ij} \models spec_{ij}$ for some $(i,j) \in I$. Let $f_{ij} \in CL(spec_{ij})$, and let s be an arbitrary reachable I-state. If the liveness condition holds, and $W_I(u)$ is supercycle-free for every reachable I-state u, then*

$$M_{ij}, s{\uparrow}ij \models f_{ij} \text{ implies } M_I, s \models_\Phi f_{ij}.$$

The correctness properties we are usually interested in are those that hold in all initial states. The large model corollary states that if all pair-specifications hold in all initial states of their respective pair-programs, then all pair-specifications also hold in all initial states of the I-program. The "spatial modality" $\bigwedge_{ij}$ quantifies over all pairs $(i,j) \in I$: $\bigwedge_{ij} spec_{ij}$ is equivalent to $\bigwedge_{(i,j)\in I} spec_{ij}$.

Corollary 1 (Large Model). *If the liveness condition holds, and $W_I(u)$ is supercycle-free for every reachable I-state u, then*

$$(\forall (i,j) \in I : M_{ij}, S^0_{ij} \models spec_{ij}) \text{ implies } M_I, S^0_I \models_\Phi \bigwedge_{ij} spec_{ij}.$$

6 Example—Readers Writers

In the readers-writers problem [6] a set of reader processes and a set of writer processes contend for access to a shared file. Mutual exclusion of access between readers and writers, and also between two writers, is required. Also, all requests by writers for access must eventually be granted ("absence of starvation"), and a writer's request takes priority over a reader's request. We specify the readers-writers problem in ACTL as follows:

Local structure of both readers and writers (P_i is a reader or a writer):

N_i: P_i is initially in its noncritical region

$\mathsf{AG}(N_i \Rightarrow (\mathsf{AX}_i T_i \wedge \mathsf{EX}_i T_i)) \wedge \mathsf{AG}(T_i \Rightarrow \mathsf{AX}_i C_i) \wedge \mathsf{AG}(C_i \Rightarrow (\mathsf{AX}_i N_i \wedge \mathsf{EX}_i N_i))$: P_i moves from N_i to T_i to C_i and back to N_i. Furthermore, P_i can always move from N_i to T_i and from C_i to N_i

$\mathsf{AG}((N_i \equiv \neg(T_i \vee C_i)) \wedge (T_i \equiv \neg(N_i \vee C_i)) \wedge (C_i \equiv \neg(N_i \vee T_i)))$: P_i is always in exactly one of the states N_i (noncritical), T_i (trying), or C_i (critical)

Reader-writer pair-specification (RP_i is reader, WP_j is writer):

Local structure: The above local structure specification for both RP_i and WP_j

$\mathsf{AG}(T_i \Rightarrow \mathsf{AF}(C_i \vee \neg N_j))$: absence of starvation for readers provided no writer requests access

$\mathsf{AG}(T_j \Rightarrow \mathsf{AF}C_j)$: absence of starvation for writers

$\mathsf{AG}((T_i \wedge T_j) \Rightarrow \mathsf{A}[T_i \mathsf{U} C_j])$: priority of writers over readers for outstanding requests to enter the critical region

$\mathsf{AG}(\neg(C_i \wedge C_j))$: mutual exclusion of access between a reader and a writer

Writer-writer pair-specification (WP_j, WP_k are writers):
Local structure: The above local structure specification for WP_j and WP_k
$\mathsf{AG}(T_j \Rightarrow \mathsf{AF}C_j) \wedge \mathsf{AG}(T_k \Rightarrow \mathsf{AF}C_k)$: absence of starvation for writers
$\mathsf{AG}(\neg(C_j \wedge C_k))$: mutual exclusion of access between two writers

Interconnection Relation I: Let K_R, K_W be the desired number of readers, writers respectively. Then I is given by $RW \cup WW$, where $RW = \{(RP_i, WP_j) \mid i \in [1:K_R], j \in [1:K_W]\}$ gives the interconnection between readers and writers, and $WW = \{(WP_j, WP_k) \mid j,k \in [1:K_W], j \neq k\}$ gives the interconnection between writers and writers. There is no interconnection between readers and readers.

For each pair-specification, we synthesize a pair-program satisfying it, using the synthesis method of [11]. Figures 1, 2 display the pair-programs for the reader-writer pair-specification, writer-writer pair-specification, respectively. Finally, we apply Definition 1 to synthesize the I-program with K_R readers and K_W writers, which is shown in Figure 3. Correctness of the I-program follows immediately from Corollary 1, since the conjunction of the pair-specifications gives us the desired correctness properties (formulae of the forms $\mathsf{AG}(p_i \Rightarrow \mathsf{AX}_i q_i)$, $\mathsf{AG}(p_i \Rightarrow \mathsf{EX}_i q_i)$ are not in ACTL^-_{ij}, but were shown to be preserved in [4], and the proof given there still applies here).

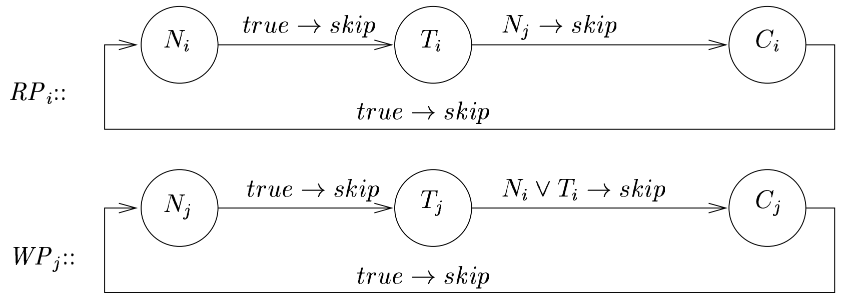


Fig. 1. Reader-writer pair-program $RP_i \parallel WP_j$.

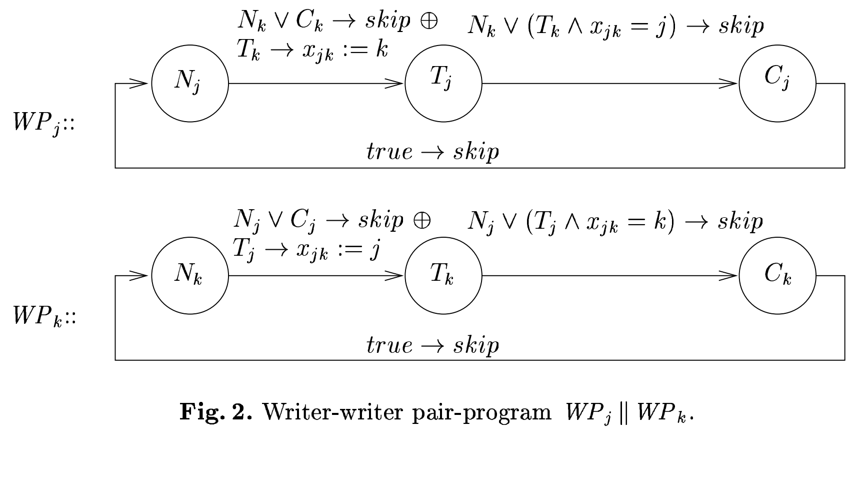
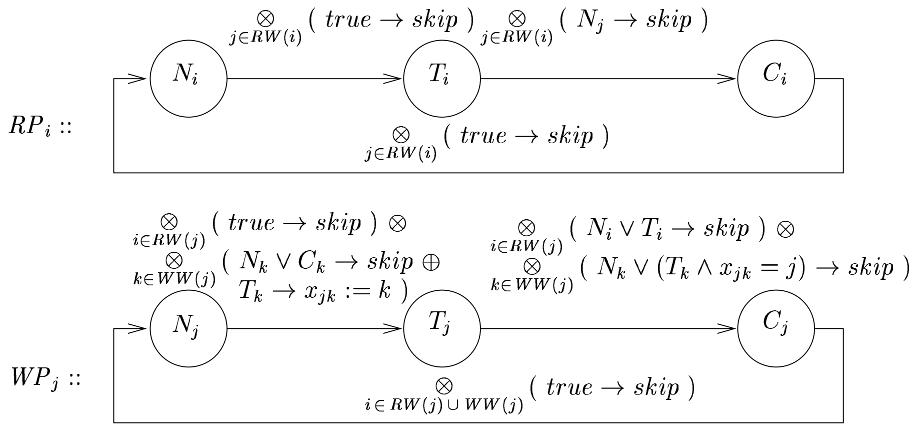

WP_j::
$N_k \vee C_k \rightarrow skip \oplus$
$T_k \rightarrow x_{jk} := k$
$N_k \vee (T_k \wedge x_{jk} = j) \rightarrow skip$
N_j
T_j
C_j
$true \rightarrow skip$
WP_k::
$N_j \vee C_j \rightarrow skip \oplus$
$T_j \rightarrow x_{jk} := j$
$N_j \vee (T_j \wedge x_{jk} = k) \rightarrow skip$
N_k
T_k
C_k
$true \rightarrow skip$

Fig. 2. Writer-writer pair-program $WP_j \parallel WP_k$.



$\bigotimes_{j \in RW(i)} (\ true \to skip\ )$
$\bigotimes_{j \in RW(i)} (\ N_j \to skip\ )$
N_i
T_i
C_i
$RP_i ::$
$\bigotimes_{j \in RW(i)} (\ true \to skip\ )$
$\bigotimes_{i \in RW(j)} (\ true \to skip\ ) \otimes \bigotimes_{k \in WW(j)} (\ N_k \vee C_k \to skip \oplus T_k \to x_{jk} := k\ )$
$\bigotimes_{i \in RW(j)} (\ N_i \vee T_i \to skip\ ) \otimes \bigotimes_{k \in WW(j)} (\ N_k \vee (T_k \wedge x_{jk} = j) \to skip\ )$
N_j
T_j
C_j
$WP_j ::$
$\bigotimes_{i \in RW(j) \cup WW(j)} (\ true \to skip\ )$

Fig. 3. Many readers, many writers program $(\|_{1 \leq i \leq K_R} RP_i) \| (\|_{1 \leq j \leq K_W} WP_j)$.

7 Example—Two Phase Commit

Our second example is a ring-based (non fault-tolerant) two-phase commit protocol $P^I = P_0^I \| P_1^I \| \cdots \| P_{n-1}^I$, where I specifies a ring. P_0^I is the *coordinator*, and $P_i^I, 1 \leq i < n$ are the participants: each participant represents a transaction. The protocol proceeds in two cycles around the ring. The coordinator initiates the first cycle, in which each participant decides to either submit its transaction or unilaterally abort. P_i^I can submit only after it observes that P_{i-1}^I has submitted. After the first cycle, the coordinator observes the state of P_{n-1}^I. If P_{n-1}^I has submitted, that means that all participants have submitted, and so the coordinator decides commit. If P_{n-1}^I has aborted, that means that some participant P_i^I unilaterally aborted, thereby causing all participants $P_j^I, i < j \leq n-1$ to abort. In that case, the coordinator decides abort. The second cycle relays the coordinators decision around the ring. The participant processes are all similar to each other, but not similar to the coordinator. Hence, there are three dissimilar pair-programs to consider: $P_{n-1}^0 \| P_0^{n-1}$, $P_0^1 \| P_1^0$, and $P_{i-1}^i \| P_i^{i-1}$ (which is replicated for each i from 2 to $n-1$ inclusive). The pair-specifications are as follows. For brevity, we omit the obvious local structure specifications (see the previous section for an example of these). The formula $f \to g$ abbreviates $\mathsf{A}[(f \Rightarrow \mathsf{AF}g)\mathsf{U_w}g]$, which means that if f holds at some point along a fullpath, then g holds at some (possibly different) point. There is no ordering on the times at which f and g hold. $f \leadsto g$ abbreviates temporal leads to: $\mathsf{AG}[f \Rightarrow \mathsf{AF}g]$.

Pair-specification for $P_{n-1}^0 \| P_0^{n-1}$:
$cm_0 \to sb_{n-1}$: the coordinator decides commit only if participant $n-1$ submits
$\mathsf{AF}(cm_0 \vee ab_0)$: the coordinator eventually decides

Pair-specification for $P_0^1 \| P_1^0$:
$\mathsf{AF}(cm_0 \vee ab_0)$: the coordinator eventually decides
$cm_1 \to cm_0$: participant 1 commits only if the coordinator decides commit

$ab_0 \rightarrow ab_1$: if the coordinator decides abort, then participant 1 aborts

$\mathsf{AG}(\neg cm_1 \vee \neg ab_1) \wedge \mathsf{AG}(cm_1 \Rightarrow \mathsf{AG}cm_1) \wedge \mathsf{AG}(ab_1 \Rightarrow \mathsf{AG}ab_1)$: participant 1 does not both commit and abort, and does not change its decision once made

$\mathsf{AG}(st_1 \Rightarrow \mathsf{EX}_1 ab_1)$: participant 1 can abort unilaterally from it's starting state

$\mathsf{AG}[sb_1 \Rightarrow \mathsf{A}[sb_1 \mathsf{U}(sb_1 \wedge (cm_0 \vee ab_0))]]$: once participant 1 submits, it does not decide until the coordinator first decides

Pair-specification for $P^i_{i-1} \,\|\, P^{i-1}_i$*, for* $2 \le i \le n-1$:

$sb_i \rightarrow sb_{i-1}$: participant i submits only if participant $i-1$ submits

$cm_i \rightarrow cm_{i-1}$: participant i commits only if participant $i-1$ commits

$(cm_{i-1} \wedge sb_i) \leadsto cm_i$: if participant i submits and participant $i-1$ commits, then participant i eventually commits

$ab_{i-1} \rightarrow ab_i$: if participant $i-1$ aborts, then so does participant i

$\mathsf{AG}(\neg cm_i \vee \neg ab_i) \wedge \mathsf{AG}(cm_i \Rightarrow \mathsf{AG}cm_i) \wedge \mathsf{AG}(ab_i \Rightarrow \mathsf{AG}ab_i)$: participant i does not both commit and abort, and does not change its decision once made

$\mathsf{AG}[sb_i \Rightarrow \mathsf{A}[sb_i \mathsf{U}(sb_i \wedge (cm_{i-1} \vee ab_{i-1}))]]$: once participant i submits, it does not decide until participant $i-1$ first decides

$\mathsf{AG}(st_i \Rightarrow \mathsf{EX}_i ab_i)$: participant i can abort unilaterally from it's starting state

The pair-programs synthesized from the above pair-specifications are given in Figures 4, 5, and 6, respectively, where $term_i \equiv cm_i \vee ab_i$, and an incoming arrow with no source indicates an initial local state. They satisfy the liveness condition and the wait-for-graph assumption, and so Theorem 1 is applicable. The synthesized two phase commit protocol P^I is given in Figure 7. We establish the correctness of P^I by the following deductive argument:

1.	$cm_0 \rightarrow sb_{n-1}$	LMT
2.	$\bigwedge_{2 \le i < n}(sb_i \rightarrow sb_{i-1})$	LMT
3.	$cm_0 \rightarrow \bigwedge_{1 \le i < n} sb_i$	1, 2
4.	$\bigwedge_{1 \le i < n}(cm_i \rightarrow cm_{i-1})$	LMT
5.	$\bigwedge_{0 \le i < n}(cm_i \rightarrow (\bigwedge_{1 \le j < n} sb_j))$	3, 4
6.	$\bigwedge_{1 \le i < n}((cm_{i-1} \wedge sb_i) \leadsto cm_i)$	LMT
7.	$\bigwedge_{0 \le i < n} \mathsf{AG}(\neg cm_i \vee \neg ab_i) \wedge \mathsf{AG}(cm_i \Rightarrow \mathsf{AG}cm_i)$	LMT
8.	$\bigwedge_{1 \le i < n} \mathsf{AG}[sb_i \Rightarrow \mathsf{A}[sb_i \mathsf{U}(sb_i \wedge (cm_{i-1} \vee ab_{i-1}))]]$	LMT
9.	$\bigwedge_{0 \le i < n-1}(cm_i \rightarrow \mathsf{A}[sb_{i+1} \mathsf{U}(sb_{i+1} \wedge cm_i)])$	5, 7, 8
10.	$\bigwedge_{1 \le i < n-1}((cm_{i-1} \wedge sb_i) \rightarrow (cm_i \wedge sb_{i+1}))$	6, 9
11.	$cm_0 \rightarrow \bigwedge_{1 \le i < n} cm_i$	3,6,7,9,10

The formulae in the above proof hold in all initial states of M_I, the global state transition diagram of P^I. The notation LMT means that the formula is a conjunct of the pair-specifications, and then we used Theorem 1 to deduce that the formula also holds in M_I (i.e., for the I-program). A notation of some formula numbers means that the formula was deduced from preceding formulae using an appropriate CTL deductive system [10]. Formula 11 gives a correctness property of two phase commit: if the coordinator commits, then so does every participant. Likewise, we establish $ab_0 \rightarrow \bigwedge_{1 \le i < n} ab_i$—if the coordinator aborts, then so does every participant. Finally, we establish $\mathsf{AF}(cm_0 \vee ab_0)$—the coordinator eventually decides—directly from the pair-specification for $P^1_0 \,\|\, P^0_1$ using

Theorem 1. Note that $\bigwedge_{1 \le i < n} \mathsf{AG}(st_i \Rightarrow \mathsf{EX}_i ab_i)$—every participant can abort unilaterally—also holds in M_I.

The deductive argument we used to establish $cm_0 \rightarrow \bigwedge_{1 \le i < n} cm_i$ required only five deductive steps (lines 3, 5, 9, 10, and 11). A completely manual correctness argument for a two-phase commit protocol would be much longer. Our vision is that the large model theorem, in combination with automatic synthesis or model checking [5] methods for verifying the correctness of pair-programs, performs most of the work in establishing behavioral properties of the synthesized I-program. Then, the use of a deductive system provides us with the flexibility needed to deduce the final desired correctness properties.

Finally, we note the significant use of nested temporal modalities in both the above pair-specifications and the deductive proof (recall that the ACTL formula $f \rightarrow g$ is really an abbreviation for $\mathsf{A}[(f \Rightarrow \mathsf{AF}g)\mathsf{U_w}g]$, which nests AF inside $\mathsf{AU_w}$). This would not have been possible in the framework of [4].

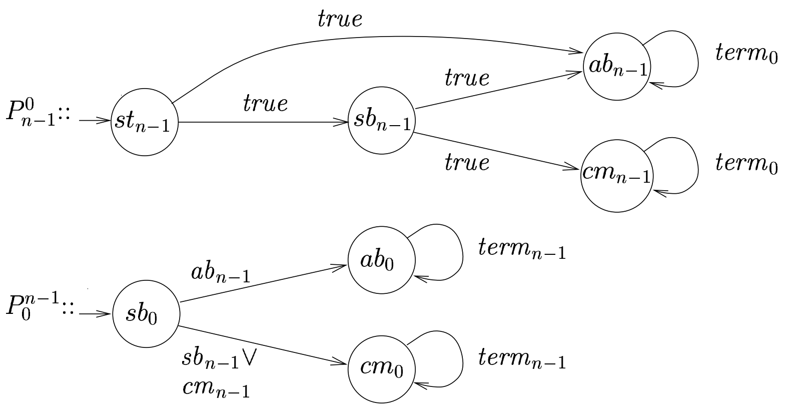


Fig. 4. Pair program $P^0_{n-1} \,\|\, P^{n-1}_0$.

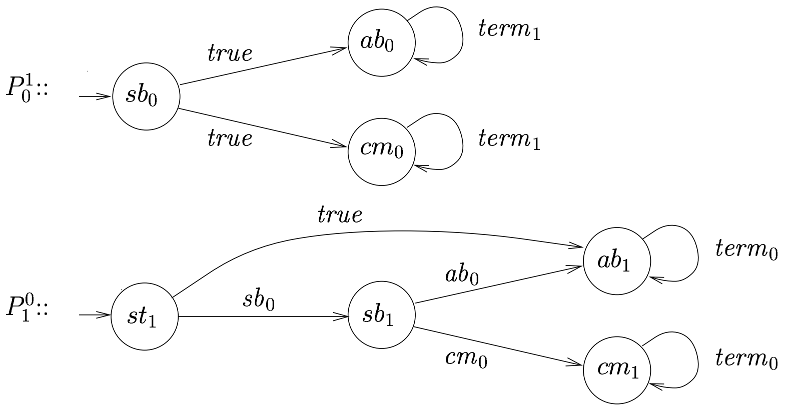


Fig. 5. Pair program $P^1_0 \,\|\, P^0_1$.

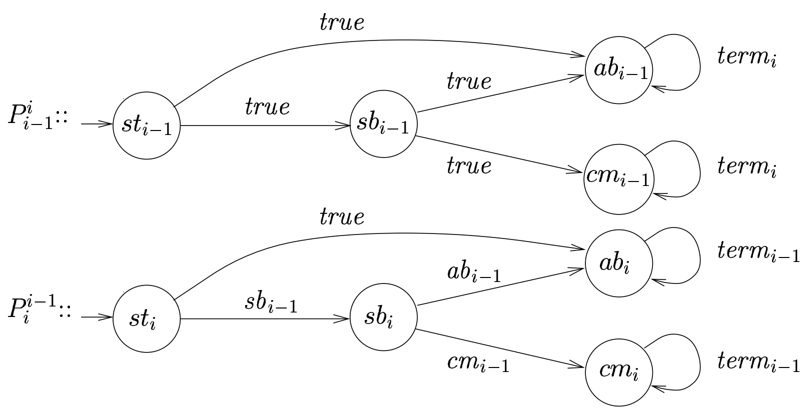


Fig. 6. Pair program $P_{i-1}^{i} \,\|\, P_i^{i-1}$.

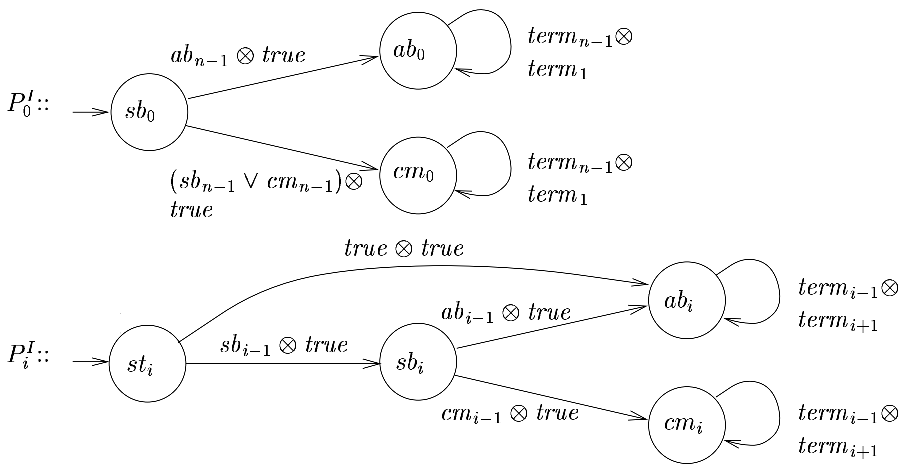


Fig. 7. The synthesized two phase commit protocol $P^I = P_0^I \,\|\, (\|_{1 \le i < n} P_i^I)$.

8 Conclusions and Further Work

We presented a synthesis method that deals with an arbitrary number of component processes without incurring the exponential overhead due to state explosion. Our method applies to any process interconnection scheme, does not make any assumption of similarity among the component processes, and preserves all pairwise and nexttime-free formulae of ACTL. We note that the method of implementing the synthesized programs on realistic distributed systems which was proposed in [4] is also applicable to the programs that our new method produces.

Further work includes dealing with fault-tolerance and real-time, and extending the method to a more expressive notation where the nodes of a synchronization skeleton denote sets of local states rather than individual local states.

References

1. A. Anuchitanukul and Z. Manna. Realizability and synthesis of reactive modules. In *Proc. 6th Intl. CAV Conference*, volume 818 of *LNCS*. Springer-Verlag, 1994.
2. A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent systems. In *Proc. 17'th Annual ACM Symposium on Principles of Distributed Computing*, pages 173 – 182, June 1998.
3. P. C. Attie and E. A. Emerson. Synthesis of concurrent systems for an atomic read/atomic write model of computation (extended abstract). In *Proc. 15'th ACM Symposium on Principles of Distributed Computing*, pages 111 – 120, May 1996.
4. P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst.*, 20(1):51–115, January 1998.
5. E. M. Clarke, E. A. Emerson, and P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.
6. P.J. Courtois, H. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667 – 668, 1971.
7. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall Inc., 1976.
8. E. W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*, pages 188–199. Springer-Verlag, New York, 1982.
9. D.L. Dill and H. Wong-Toi. Synthesizing processes and schedulers from temporal specifications. In *2'nd Intl. CAV Conference*, *LNCS* vol. 531. Springer-Verlag, 1990.
10. E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B. 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.
12. E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8:275–306, 1987.
13. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, May 1994.
14. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, January 1984.
15. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, New York, 1989. ACM.
16. A. Pnueli and R. Rosner. On the synthesis of asynchronous reactive modules. In *Proc. 16th ICALP*, volume 372 of *LNCS*, Berlin, 1989. Springer-Verlag.