Synthesis of Large Dynamic Concurrent Programs from Dynamic Specifications

Paul C. Attie [1]

College of Computer Science, Northeastern University
and
MIT Laboratory for Computer Science
attie@ccs.neu.edu
http://www.ccs.neu.edu/home/attie/Attie.html

November 11, 2003

Abstract

We present a tractable method for synthesizing arbitrarily large concurrent programs, for a shared memory model with common hardware-available primitives such as atomic registers, compare-and-swap, load-linked/store conditional, etc. The programs we synthesize are dynamic: new processes can be created and added at run-time, and so our programs are not finite-state, in general. Nevertheless, we successfully exploit automatic synthesis and model-checking methods based on propositional temporal logic. Our method is algorithmically efficient, with complexity polynomial in the number of component processes (of the program) that are "alive" at any time. Our method does not explicitly construct the automata-theoretic product of all processes that are alive, thereby avoiding *state explosion*. Instead, for each pair of processes which interact, our method constructs an automata-theoretic product (*pair-machine*) which embodies all the possible interactions of these two processes. From each pair-machine, we can synthesize a correct *pair-program* which coordinates the two involved processes as needed. We allow such pair-programs to be added dynamically at run-time. They are then "composed conjunctively" with the currently alive pair-programs to re-synthesize the program as it results after addition of the new pair-program. We are thus able to add new behaviors, which result in new properties being satisfied, at run-time. This "incremental composition" step has complexity independent of the total number of processes, it only requires the mechanical analysis of the two processes in the pair-program, and their immediate neighbors, i.e., the other processes which they interact directly with. 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 mechanically synthesizing a concurrent program consisting of a large, and dynamically varying, number of sequential processes executing in parallel. Our programs operate in shared memory, commonly available hardware primitives, such as using read and write operations on atomic registers, compare-and-swap, load-linked/store conditional. Even thought our synthesis method is largely mechanical, we only require that each process have a finite number of actions, and that the data referred to in action guards be finite. Underlying data that processes operate on, and which does not affect action guards, can be infinite. Also, since the number of processes can increase without limit, the synthesized program as a whole is not finite-state. In addition, our method is computationally efficient, it does not explicitly construct the automata-theoretic product of a large number of processes (e.g., all processes that are "alive" at some point) and is therefore not susceptible to the *state-explosion problem*, i.e., the exponential growth of the number of global

[1]Supported in part by NSF Grant CCR-0204432

states with the number of processes, which is widely acknowledged to be the primary impediment to large-scale application of mechanical verification methods.

Rather than build a global product, our method constructs the product of small numbers of sequential processes, and in particular, the product of each pair of processes that interact, thereby avoiding the exponential complexity in the number of processes that are "alive" at any time. The product of each pair of interacting processes, or *pair-machine*, is a Kripke structure which embodies the interaction of the two processes. The pair-machines can be constructed manually, and then efficiently model-checked (since it is small) to verify *pair-properties*: behavioral properties of the interaction of the two processes, when viewed in isolation from the remaining processes. Alternatively, the pair-properties can be specified first, and the pair-machine automatically synthesized from the pair-properties by the use of mechanical synthesis methods such as [EC82, MW84, KV97]. Again this is efficient since the pair-machines are small.

Corresponding to each pair-machine is a *pair-program*, a syntactic realization of the pair-machine, which generates the pair-machine as its global-state transition diagram. Finally, we syntactically compose all of the pair-programs. This composition has a conjunctive nature: a process P_i can make a transition iff that transition is permitted by *all* of the pair-programs in which P_i participates. We allow such "pair-programs" to be added dynamically at run-time. They are then composed with the currently alive pair-programs to re-synthesize the program as it results after addition of the new pair-program. We are thus able to add new behaviors, which result in new properties being satisfied, at run-time. The use of pairwise composition greatly facilitates this, since the addition of a new pair-program does not disturb the correctness properties which are satisfied by the currently present pair-programs. We establish a "large model" theorem which shows that the synthesized large program inherits correctness properties from the pair-programs.

Since the pair-machines are small, and since the composition step operates on syntax, i.e., the pair-programs themselves, and not their state-transition diagrams, our method is computationally efficient. In particular, the dynamic addition of a single pair-program requires a mechanical synthesis or model checking step whose complexity is independent of the total number of alive processes at the time, but which depends only on the checking products of the two processes involved in the pair-program, together with some of their neighbors, i.e., the processes which they immediately interact with. Our method thus overcomes the severe limitations previously imposed by state-explosion on the applicability of automatic synthesis methods, and extends these methods to the new domain of dynamic programs.

Our method can generate systems under arbitrary *process interconnection* schemes, e.g., fully connected, ring, star. In our model of parallel computation, two processes are interconnected if and only if either (1) one process can inspect the local state of the other process or (2) both processes read and/or write a common variable, or both.

The method requires the pair-programs to satisfy certain technical assumptions, thus it is not completely general. Nevertheless, it is applicable in many interesting cases. We illustrate our method by synthesizing a ring-based two phase commit protocol. Using the large model theorem, we show that correctness properties that two processes of the ring satisfy when interacting in isolation carry over when those processes are part of the ring. We then easily construct a correctness proof for the ring using these properties. We note that the ring can contain an arbitrarily large number of processes, i.e., we really synthesize a *family* of rings, one for each natural number.

A crucial aspect of our method is its soundness: which correctness properties can be established for our synthesized programs? We establish a "large model" theorem which shows that the synthesized program inherits all of the correctness properties of the pair-programs, i.e., the pair-properties. We express our pair-properties in the branching time temporal logic ACTL [GL94] minus the nexttime operator. In particular, propositional invariants and some temporal leads-to properties of any pair-program also hold of the synthesized 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.) In addition, we can use a suitable deductive system to combine the pair-properties to deduce correctness properties of the large program which are not directly expressible in pairwise fashion.

This paper extends our previous work [AE98] on the synthesis of large concurrent programs in four important directions:

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. In our method, every process can be nonisomorphic with every other process, and our method would still be computationally efficient.

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 the conditions are purely propositional) to formulae that can contain arbitrary nesting of temporal modalities.

3. It eliminates the requirement that the number of processes of the synthesized program be fixed: our previous work synthesized an infinite family of programs, each of which contains a large, but fixed, number of processes. By contrast, the current method produces a single program, in which the number of processes can dynamically increase at run-time.

4. It produces programs that do not require a large grain of atomicity: in [Att99, AE98], each process needed to atomically inspect the state of all of its neighbors (i.e., all processes with which it is composed in some pair-program) in a single transition. By contrast, the current method produces programs that operate using only hardware-available primitives for interprocess communication and synchronization.

To demonstrate the utility of our method, we apply it to synthesize a two-phase commit protocol, and a replicated data service.

Related work. Previous synthesis methods [AM94, DWT90, EC82, KMTV00, KV97, MW84, PR89a, PR89b] 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 $O(N)$ local states, requires building the global state transition diagram of size $O(N^K)$. There are a number of methods proposed for verifying correctness properties of an infinite family of finite-state processes [APR$^+$01, CGB86, EK00, EN96, PRZ01, SG92]. All of these deal with an infinite family of concurrent programs, where each program consists of a possibly large, but *fixed* set of processes. No method to date can verify or synthesize a *single* concurrent program in which processes can be dynamically created *at run time*. Furthermore, all methods to date that deal with large concurrent programs, apart from our own previous work [Att99, AE98] make the "parametrized system" assumption: the processes can be partitioned into a small number of "equivalence classes," within each of which all processes are isomorphic. Hence, in eliminating these two significant restrictions, our method is a significant improvement over the previous literature, and moves automated synthesis methods close to the realm of practical distributed algorithms. We illustrate this point by using our method to synthesize a replicated data service based on the algorithms of [FGL$^+$99, LLSG92]. Our algorithm is actually more flexible, since it permits the dynamic addition of more replicas at run time. Some synthesis method in the literature synthesize "open systems," or "reactive modules," which interact with an environment, and are required to satisfy a specification regardless of the environment's behavior. The main argument for open systems synthesis is that open systems can deal with any "input" which the environment presents. We can achieve this effect by using the "exists nexttime" (EX) modality of the temporal logic CTL [EC82, Eme90]. We illustrate this in our replicated data service example, where we specify that a client can submit operations at any time.

The rest of the paper is as follows. Section 2 presents our model of concurrent computation. Section 3 discusses temporal logic and fairness. Section 4 presents a restricted version of the method, which is only applicable to static concurrent programs: those with a fixed set of processes. This approach simplifies the development and exposition of our method, Section 5 establishes the soundness of the synthesis method for static programs. Section 6 presents the two phase commit example, which can be treated with the restricted method. Section 7 presents the general synthesis method, which can produce dynamic concurrent programs. Section 8 shows that the general method is sound. Section 9 outlines how the synthesized programs can be implemented using atomic registers. In Section 10 we use our method to synthesize an eventually-serializable replicated data service. Section 11 discusses further work and concludes.

2 Model of Concurrent Computation

We assume the existence of a possibly infinite, universal set Pids of unique process indices. A concurrent program P consists of a finite, unbounded, and possibly varying number of sequential processes $P_i, i \in \mathsf{Pids}$ running in parallel, i.e., $P = P_1 \| \cdots \| P_K$ where $P_1, \ldots, P_K$ execute in parallel and are the processes that have been "created" so far. For technical convenience, we do not allow processes to be "destroyed" in our model. Process destruction can be easily emulated by having a process enter a "sink" state, from which it has no enabled actions.

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.

As stated above, we compose a dynamically varying number of pair-programs to synthesize the overall program. To define the syntax and semantics of the pair-programs, we use the *synchronization skeleton* model of [EC82]. 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 $\oplus_{\ell\in[n]} B_\ell \to A_\ell$,[2] where each $B_\ell \to A_\ell$ is a guarded command [Dij76], and $\oplus$ is guarded command "disjunction," i.e., the arc is equivalent to n arcs, between the same pair of nodes, each labeled with one of the $B_\ell \to A_\ell$. Let $\hat{P}_i$ denote the synchronization skeleton of process i with all the arc labels removed.

Roughly, the operational semantics of $\oplus_{\ell\in[n]} B_\ell \to A_\ell$ 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.[3] 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 $(s_1, \dots, s_K, v_1, \dots, v_m)$ where each s_i is the current local state of P_i, and $v_1, \dots, v_m$ is a list giving the current values of all the shared variables, $x_1, \dots, x_m$ (we assume these are ordered in a fixed way, so that $v_1, \dots, v_m$ 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 $s = (s_1, \dots, s_i, \dots, s_K, v_1, \dots, v_m)$ 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 $(s_1, \dots, s'_i, \dots, s_K, v'_1, \dots, v'_m)$ where $v'_1, \dots, v'_m$ 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. If the number of processes is fixed, then the concurrent program can be written as $P_1 \| \cdots \| P_K$, where K is fixed. In this case, we also specify a a set S_0 of global states in which execution is permitted to start. These are the *initial states*. The program is then written as $(S_0, P_1 \| \cdots \| P_K)$. An initialized (computation) path is a computation path whose first state is an initial state. A state is *reachable* iff it lies along some initialized path.

3 Temporal Logic and Fairness

CTL* is a propositional branching time temporal logic [Eme90] 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 sublogic ACTL* [GL94] is the "universal fragment" of CTL*: it results from CTL by restricting negation to propositions, and eliminating the existential path quantifier E. The sublogic CTL [EC82] results from restricting CTL* so that every linear-time modality is paired with a path quantifier, and vice-versa. The sublogic ACTL [GL94] results from restricting ACTL* in the same way. The linear-time temporal logic PTL [MW84] results from removing the path quantifiers from CTL*.

We have the following syntax for CTL*. We inductively define a class of state formulae (true or false of states) using rules (S1)–(S3) below and a class of path formulae (true or false of paths) using rules (P1)–(P3) below:

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

[3] This interpretation was proposed by [Dij82].

(S1) The constants *true* and *false* are state formulae. p is a state formulae for any atomic proposition p.

(S2) If f, g are state formulae, then so are $f \wedge g$, $\neg f$.

(S3) If f is a path formula, then $\mathsf{A}f$ is a state formula.

(P1) Each state formula is also a path formula;

(P2) If f, g are path formulae, then so are $f \wedge g$, $\neg f$.

(P3) If f, g are path formulae, then so are $\mathsf{X}_j f$, $f \mathsf{U} g$.

The linear-time temporal logic PTL [MW84] consists of the set of path formulae generated by rules (S1) and (P1)–(P3). We also introduce some additional modalities as abbreviations: $\mathsf{F}f$ (eventually) for $[true \mathsf{U} f]$, $\mathsf{G}f$ (always) for $\neg\mathsf{F}\neg f$, $[f \mathsf{U_w} g]$ (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 $\mathsf{FG}f$.

Likewise, we have the following syntax for ACTL*.

(S1) The constants *true* and *false* are state formulae. p and $\neg p$ are state formulae for any atomic proposition p.

(S2) If f, g are state formulae, then so are $f \wedge g$, $f \vee g$.

(S3) If f is a path formula, then $\mathsf{A}f$ is a state formula.

(P1) Each state formula is also a path formula;

(P2) If f, g are path formulae, then so are $f \wedge g$, $f \vee g$.

(P3) If f, g are path formulae, then so are $\mathsf{X}_j f$, $f \mathsf{U} g$, and $f \mathsf{U_w} g$.

The logic ACTL [GL94] is obtained by replacing rules (S3),(P1)–(P3) by (S3'):

(S3') If f, g are state formulae, then so are $\mathsf{AX}_j f$, $\mathsf{A}[f \mathsf{U} g]$, and $\mathsf{A}[f \mathsf{U_w} g]$.

The set of state formulae generated by rules (S1)–(S3) and (P0) forms ACTL. The logic ACTL$^-$ is the logic ACTL without the AX_j modality. We define the logic ACTL$^* - X$ to be the logic ACTL* without the X_j modality, and 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 $\mathcal{AP}_i \cup \mathcal{AP}_j$.

Formally, we define the semantics of CTL* formulae with respect to a structure $M = (S, R)$ consisting of

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

Here $\mathcal{AP} = \bigcup_{i \in \mathsf{Pids}} \mathcal{AP}_i$, 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 process i can modify these propositions (which collectively define the local state of process i).

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. A fullpath $(s_1, s_2, \ldots)$ is infinite unless for some s_k there is no s_{k+1} such that $(s_k, s_{k+1}) \in R$. We use the convention (1) that $\pi = (s_1, s_2, \ldots)$ denotes a fullpath and (2) that π^i denotes the suffix $(s_i, s_{i+1}, s_{i+2}, \ldots)$ of π, provided $i \leq |\pi|$, where $|\pi|$, the length of π, is ω when π is infinite and k when π is finite and of the form $(s_1, \ldots, s_k)$; otherwise π^i is undefined. We also use the usual notation to indicate truth in a structure: $M, s_1 \models f$ (respectively $M, \pi \models f$) means that f is true in structure M at state s_1 (respectively of fullpath π). In addition, we use $M, S \models f$ to mean $\forall s \in S : (M, s \models f)$, where S is a set of states. We define $\models$ inductively:

(S1) $M, s_1 \models \mathit{true}$ and $M, s_1 \not\models \mathit{false}$. $M, s_1 \models p$ iff $s_1(p) = \mathit{true}$. $M, s_1 \models \neg p$ iff $s_1(p) = \mathit{false}$.

(S2) $M, s_1 \models f \wedge g$ iff $M, s_1 \models f$ and $M, s_1 \models g$
$M, s_1 \models f \vee g$ iff $M, s_1 \models f$ or $M, s_1 \models g$

(S3) $M, s_1 \models \mathsf{A}f$ iff for every fullpath $\pi = (s_1, s_2, \ldots)$ in M: $M, \pi \models f$

(P1) $M, \pi \models f$ iff $M, s_1 \models f$

(P2) $M, \pi \models f \wedge g$ iff $M, \pi \models f$ and $M, \pi \models g$
$M, \pi \models f \vee g$ iff $M, \pi \models f$ or $M, \pi \models g$

(P3) $M, \pi \models \mathsf{X}_j f$ iff π^2 is defined and $(s_1, s_2) \in R_j$ and $M, \pi^2 \models f$
$M, \pi \models f \mathsf{U} g$ iff there exists $i \in [1 : |\pi|]$ such that
$M, \pi^i \models g$ and for all $j \in [1 : (i-1)]$: $M, \pi^j \models f$
$M, \pi \models f \mathsf{U_w} g$ iff for all $i \in [1 : |\pi|]$
if $M, \pi^j \not\models g$ for all $j \in [1 : i]$, then $M, \pi^i \models f$

When the structure M is understood from context, it may be omitted (e.g., $M, s_1 \models p$ is written as $s_1 \models p$). Since the other logics are all sublogics of CTL*, the above definition provides semantics for them as well. We refer the reader to [Eme90] for details in general, and to [GL94] for details of ACTL.

3.1 Fairness

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 [EL87] as follows:

(S3-fair) $M, s_1 \models_\Phi \mathsf{A}f$ 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 Synthesis of Static Concurrent Programs

To simplify the development and exposition of our method, we first present a restricted case, where we synthesize *static* concurrent programs, i.e., those with a fixed set of processes. We extend the method to dynamic concurrent programs in Section 7 below.

As stated earlier, our aim is to synthesize a large concurrent program $P = P_{i_1} \parallel \ldots \parallel P_{i_K}$ without explicitly generating its global state transition diagram, and thereby incurring time and space complexity exponential in the number of component processes of P. We achieve this by breaking the synthesis problem down into two steps:

1. For every pair of processes in P that interact directly, synthesize a *pair-program* that describes their interaction.

2. Combine all the pair-programs to produce P.

When we say P_i and P_j interact directly, we mean that each process can read the other processe's atomic propositions (which, recall, encode the processe's local state), and that they have a set $\mathcal{SH}_{ij}$ of shared variables that they both read and write. We define the *interconnection relation* $I \subseteq \{i_1, \ldots, i_K\} \times \{i_1, \ldots, i_K\} \times \mathrm{ACTL}^-$ as follows: $(i, j, f_{ij}) \in I$ iff P_i and P_j interact directly, and f_{ij} is an ACTL$^-$ formula specifying this interaction. In the sequel we let $spec_{ij}$ denote the specification associated with i, j, and we say that $\{i_1, \ldots, i_K\}$ is the domain of I. We introduce the "spatial modality" $\bigwedge_{ij}$ which quantifies over all pairs (i, j) such that i and j are related by I. Thus, $\bigwedge_{ij} spec_{ij}$ is equivalent to $\forall (i, j, spec_{ij}) \in I : spec_{ij}$. We stipulate that I is "irreflexive," that is, $(i, i, f_{ij}) \notin I$ for all i, f_{ij}, and that every process interacts directly with at least one other process: $\forall i \in \{i_1, \ldots, i_K\} : (\exists j, f_{ij} : (i, j, f_{ij}) \in I \vee (j, i, f_{ij}) \in I)$. Furthermore, for any pair of process indices i, j, I contains at most one pair $(k, \ell, f_{k\ell})$ such that $k \in \{i, j\}$ and $\ell \in \{i, j\}$. In the sequel, we say that i and j are *neighbors* when $(i, j, f_{ij}) \in I$ or $(j, i, f_{ij}) \in I$, for some f_{ij}. We shall sometimes abuse notation and write $(i, j) \in I$ (or $i\, I\, j$) for $\exists f_{ij} : ((i, j, f_{ij}) \in I \vee (j, i, f_{ij}) \in I)$. We also introduce the following abbreviations: $I(i)$ denotes the set $\{j \mid i\, I\, j\}$; and $\hat{I}(i)$ denotes the set $\{i\} \cup \{j \mid i\, I\, j\}$. Since the interconnection relation I embodies a complete specification, we shall refer to a program that has been synthesized from I as an *I-program*, and to its component processes as *I-processes.*

Since our focus in this article is on avoiding state-explosion, we shall not explicitly address step 1 of the synthesis method outlined above. Any method for deriving concurrent programs from temporal logic specifications can be used to generate the required pair-programs, e.g., the synthesis method of [EC82]. Since a pair-program has only $O(N^2)$ states (where N is the size of each sequential process), the problem of deriving a pair-program from a specification is considerably easier than that of deriving an I-program from the specification. Hence, the contribution of this article, namely the second step above, is to reduce the more difficult problem (deriving the I-program) to the easier problem (deriving the pair-programs). We proceed as follows.

For sake of argument, let us first assume that all the pair-programs are actually isomorphic to each other. Let $i\, I\, j$. We denote the pair-program for processes i and j by $(S^0_{ij}, P^j_i \parallel P^i_j)$, where S^0_{ij} is the set of initial states, P^j_i is the synchronization skeleton for process i in this pair-program, and P^i_j is the synchronization skeleton for process j. We take $(S^0_{ij}, P^j_i \parallel P^i_j)$ and generalize it in a natural way to an I-program. We show that our generalization preserves a large class of correctness properties. Roughly the idea is as follows. Consider first the generalization to three pairwise interconnected processes i, j, k, i.e., $I = \{(i, j), (j, k), (k, i)\}$[4]. With respect to process i, the proper interaction (i.e., the interaction required to satisfy the specification) between process i and process j is captured by the synchronization commands that label the arcs of P^j_i. Likewise, the proper interaction between process i and process k is captured by the arc labels of P^k_i. Therefore, in the three-process program consisting of processes i, j, k executing concurrently, (and where process i is interconnected to both process j and process k), the proper interaction for process i with processes j and k is captured as follows: when process i traverses an arc, the synchronization command which labels that arc in P^j_i is executed "simultaneously" with the synchronization command which labels the corresponding arc in P^k_i. For example, taking as our specification the mutual exclusion problem, if P_i executes the mutual exclusion protocol with respect to both P_j and P_k, then, when P_i enters its critical section, both P_j and P_k must be outside their own critical sections.

Based on the above reasoning, we determine that the synchronization skeleton for process i

[4]Note the abuse of notation: we have omitted the ACTL$^-$ formulae.

in the aforementioned three-process program (call it P_i^{jk}) has the same basic graph structure as P_i^j and P_i^k, and an arc label in P_i^{jk} is a "composition" of the labels of the corresponding arcs in P_i^j and P_i^k. In addition, the initial states S_{ijk}^0 of the three-process program are exactly those states that "project" onto initial states of all three pair-programs ($(S_{ij}^0, P_i^j \parallel P_j^i)$, $(S_{ik}^0, P_i^k \parallel P_k^i)$, and $(S_{jk}^0, P_j^k \parallel P_k^j)$).

Generalizing the above to the case of an arbitrary interconnection relation I, we see that the skeleton for process i in the I-program (call it P_i) has the same basic graph structure as P_i^j, and a transition label in P_i is a "composition" of the labels of the corresponding transitions in $P_i^{j_1}, \ldots, P_i^{j_n}$, where $\{j_1, \ldots, j_n\} = I(i)$, i.e., processes $j_1, \ldots, j_n$ are all the I-neighbors of process i. Likewise the set S_I^0 of initial states of the I-program is exactly those states all of whose "projections" onto all the pairs in I give initial states of the corresponding pair-program.

We now note that the above discussion does not use in any essential way the assumption that pair-programs are isomorphic to each other. In fact, the above argument can still be made if pair-programs are not isomorphic, provided that they induce the same *local structure* on all common processes. That is, for pair-programs $(S_{ij}^0, P_i^j \parallel P_j^i)$ and $(S_{ik}^0, P_i^k \parallel P_k^i)$, we require that $graph(P_i^j) = graph(P_i^k)$, where $graph(P_i^j)$, $graph(P_i^k)$ result from removing all arc labels from P_i^j, P_i^k respectively. Also, the initial state sets of all the pair-programs must be so that there is at least one I-state that projects onto some initial state of every pair-program (and hence the initial state set of the I-program will be nonempty). We assume, in the sequel, that these conditions hold. Also, all quoted results from [AE98] have been reverified to hold in our setting, i.e., when the similarity assumptions of [AE98] are dropped.

Before formally defining our synthesis method, we need some technical definitions.

Since P_i^j and P_i have the same local structure, they have the same nodes (remember that P_i^j and P_i are synchronization skeletons). A node of P_i^j, P_i is a mapping of $\mathcal{AP}_i$ to $\{true, false\}$. We will refer to such nodes as i-states. A state of the pair-program $(S_{ij}^0, P_i^j \parallel P_j^i)$ 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 $P_i^j \parallel P_j^i$ as ij-states. An ij-state 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.

We now turn to I-programs. If interconnection relation I has domain $\{i_1, \ldots, i_K\}$, then we denote an I-program by $(S_I^0, P_{i_1}^I \parallel \ldots \parallel P_{i_K}^I)$. S_I^0 is the set of initial states, and P_i is the synchronization skeleton for process i ($i \in \{i_1, \ldots, i_K\}$) in this I-program. A state of $(S_I^0, P_{i_1}^I \parallel \ldots \parallel P_{i_K}^I)$ 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 (we assume some fixed ordering of these variables, so that the values assigned to them are uniquely determined by the list $v^1, \ldots, v^n$). We refer to states of an I-program as I-states. An I-state inherits the assignments defined by its component i-states ($i \in \{i_1, \ldots, i_K\}$): $s_{ij}(p_i) = s_i(p_i)$, where $s = (s_{i_1}, \ldots, s_{i_K}, v^1, \ldots, v^n)$, and p_i is an arbitrary atomic proposition in $\mathcal{AP}_i$ ($i \in \{i_1, \ldots, i_K\}$). We shall usually use s, t, u to denote I-states. If $J \subseteq I$, then we define a J-program exactly like an I-program, but using interconnection relation J instead of I. J-state is similarly defined.

Let s_i be an i-state. We define a state-to-formula operator $\{\!| s_i |\!\}$ that takes an i-state s_i as an argument and returns a propositional formula that characterizes s_i in that $s_i \models \{\!| s_i |\!\}$, and $s_i' \not\models \{\!| s_i |\!\}$ for all i-states s_i' such that $s_i' \neq s_i$: $\{\!| s_i |\!\} = (\bigwedge_{s_i(p_i)=true} p_i) \ \wedge \ (\bigwedge_{s_i(p_i)=false} \neg p_i)$, where p_i ranges over the members of $\mathcal{AP}_i$. $\{\!| s_{ij} |\!\}$ is defined similarly. We define the **state projection**

operator $\upharpoonright$. This operator has several variants. First of all, 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 $s_{ij} = (s_i, s_j, v^1_{ij}, \ldots, v^m_{ij})$, then $s_{ij}{\upharpoonright}i = s_i$. This gives the i-state corresponding to the I-state s, ij-state s_{ij}, respectively. 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 $s{\upharpoonright}ij = (s_i, s_j, v^1_{ij}, \ldots, v^m_{ij})$, where $v^1_{ij}, \ldots, v^m_{ij}$ are those values from $v^1, \ldots, v^n$ that denote values of variables in $\mathcal{SH}_{ij}$. This gives the ij-state corresponding to the I-state s, and is well defined only when $i\,I\,j$. We also define projection onto the shared variables in $\mathcal{SH}_{ij}$ from both ij-states and I-states: if $s_{ij} = (s_i, s_j, v^1_{ij}, \ldots, v^m_{ij})$, then $s_{ij}{\upharpoonright}\mathcal{SH}_{ij} = (v^1_{ij}, \ldots, v^m_{ij})$, and if $s = (s_{i_1}, \ldots, s_{i_K}, v^1, \ldots, v^n)$, then $s{\upharpoonright}\mathcal{SH}_{ij} = (v^1_{ij}, \ldots, v^m_{ij})$, where $v^1_{ij}, \ldots, v^m_{ij}$ are those values from $v^1, \ldots, v^n$ that denote values of variables in $\mathcal{SH}_{ij}$. 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 $s{\upharpoonright}J = (s_{j_1}, \ldots, s_{j_L}, v^1_J, \ldots, v^m_J)$, where $\{j_1, \ldots, j_L\}$ is the domain of J, and $v^1_J, \ldots, v^m_J$ are those values from $v^1, \ldots, v^n$ that denote values of variables in $\bigcup_{(i,j)\in J} \mathcal{SH}_{ij}$. This gives the J-state (defined analogously to an I-state) corresponding to the I-state s and is well defined only when $J \subseteq I$.

To define projection for paths, we first extend the definition of path (and fullpath) to include the index of the process making the transition, e.g., each transition is labeled by an index denoting this process. For example, a path in M_I would be represented as $s^1 \xrightarrow{d_1} s^2 \cdots s^n \xrightarrow{d_n} s^{n+1} \xrightarrow{d_{n+1}} s^{n+2} \cdots$, where $\forall m \geq 1 : (d_m \in \mathit{dom}(I))$. Let π be an arbitrary path in M_I. For any J such that $J \subseteq I$, define a J-*block* (cf. [CGB86] and [BCG88]) of π to be a maximal subsequence of π that starts and ends in a state and does not contain a transition by any P_i such that $i \in \mathit{dom}(J)$. Thus we can consider π to be a sequence of J-blocks with successive J-blocks linked by a single P_i-transition such that $i \in \mathit{dom}(J)$ (note that a J-block can consist of a single state). It also follows that $s{\upharpoonright}J = t{\upharpoonright}J$ for any pair of states s, t in the same J-block. This is because a transition that is not by some P_i such that $i \in \mathit{dom}(J)$ cannot affect any atomic proposition in $\bigcup_{i\in \mathit{dom}(J)} \mathcal{AP}_i$, nor can it change the value of a variable in $\bigcup_{(i,j)\in J} \mathcal{SH}_{ij}$; and a J-block contains no such P_i transition. Thus, if B is a J-block, we define $B{\upharpoonright}J$ to be $s{\upharpoonright}J$ for some state s in B. We now give the formal definition of path projection. We use the same notation ($\upharpoonright$) as for state projection. Let B^n denote the nth J-block of π.

Definition 1 (Path projection) *Let π be $B^1 \xrightarrow{d_1} \cdots B^n \xrightarrow{d_n} B^{n+1} \cdots$ where B^m is a J-block for all $m \geq 1$. Then the* Path Projection Operator *$\upharpoonright J$ is given by: $\pi{\upharpoonright}J = B^1{\upharpoonright}J \xrightarrow{d_1} \cdots B^n{\upharpoonright}J \xrightarrow{d_n} B^{n+1}{\upharpoonright}J \cdots$*

Thus there is a one-to-one correspondence between J-blocks of π and states of $\pi{\upharpoonright}J$, with the nth J-block of π corresponding to the nth state of $\pi{\upharpoonright}J$ (note that path projection is well defined when π is finite).

The above discussion leads to the following definition of the synthesis method, which shows how an I-process P_i of the I-program $(S^0_I, P^I_{i_1} \| \ldots \| P^I_{i_K})$ is derived from the pair-processes $\{P^j_i \mid j \in I(i)\}$ of the the pair-programs $\{(S^0_{ij}, P^j_i \| P^i_j) \mid j \in I(i)\}$:

Definition 2 (Pairwise synthesis) *An I-*process *P_i is derived from the pair-processes P^j_i, for all $j \in I(i)$ as follows:*

P_i contains a move from s_i to t_i with label $\otimes_{j\in I(i)} \oplus_{\ell\in[1:n]} B^j_{i,\ell} \to A^j_{i,\ell}$

iff

for every j in $I(i)$: P^j_i contains a move from s_i to t_i with label $\oplus_{\ell\in[1:n]} B^j_{i,\ell} \to A^j_{i,\ell}$.

The initial state set *S^0_I of the I-program is derived from the initial state S^0_{ij} of the pair-program as* *follows:*

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

Here $\oplus$ and $\otimes$ are guarded command "disjunction" and "conjunction," respectively. Roughly, the operational semantics of $B_{i,1}^j \to A_{i,1}^j \oplus B_{i,2}^j \to A_{i,2}^j$ is that if one of the guards $B_{i,1}^j, B_{i,2}^j$ evaluates to true, then the corresponding body $A_{i,1}^j, A_{i,2}^j$ respectively, can be executed. If neither $B_{i,1}^j$ nor $B_{i,2}^j$ evaluates to true, then the command "blocks," i.e., waits until one of $B_{i,1}^j, B_{i,2}^j$ evaluates to true.[5] We call an arc whose label has the form $\oplus_{\ell \in [1:n]} B_{i,\ell}^j \to A_{i,\ell}^j$ a *pair-move*. In compact notation, a pair-process has at most one move between any pair of local states.

The operational semantics of $B_{i,1}^j \to A_{i,1}^j \otimes B_{i,2}^j \to A_{i,2}^j$ is that if both of the guards $B_{i,1}^j, B_{i,2}^j$ evaluate to true, then the bodies $A_{i,1}^j, A_{i,2}^j$ can be executed in parallel. If at least one of $B_{i,1}^j$, $B_{i,2}^j$ evaluates to false, then the command "blocks," i.e., waits until both of $B_{i,1}^j, B_{i,2}^j$ evaluate to true. We call an arc whose label has the form $\otimes_{j \in I(i)} \oplus_{\ell \in [1:n]} B_{i,\ell}^j \to A_{i,\ell}^j$ an *I-move*. In compact notation, an I-process has at most one move between any pair of local states.

The above definition 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 $K = |\{i_1, \ldots, i_K\}|$.

Let M_{ij}, M_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. The technical definitions are given below, and follow the operational semantics given in Section 2.

Definition 3 (Pair-structure) *Let $i\,I\,j$. The semantics of $(S_{ij}^0, P_i^j \parallel P_j^i)$ is given by the* pair-structure *$M_{ij} = (S_{ij}^0, S_{ij}, R_{ij})$ where*

1. *S_{ij} is a set of ij-states,*
2. *$S_{ij}^0 \subseteq S_{ij}$ gives the initial states of $(S_{ij}^0, P_i^j \parallel P_j^i)$, and*
3. *$R_{ij} \subseteq S_{ij} \times \{i,j\} \times S_{ij}$ is a transition relation giving the transitions of $(S_{ij}^0, P_i^j \parallel P_j^i)$. A transition (s_{ij}, h, t_{ij}) by $P_h^{\bar{h}}$ is in R_{ij} if and only if all of the following hold:*
   - *(a) $h \in \{i,j\}$,*
   - *(b) s_{ij} and t_{ij} are ij-states, and*
   - *(c) there exists a move $(s_{ij}{\upharpoonright}h, \oplus_{\ell \in [1:n]} B_{h,\ell}^{\bar{h}} \to A_{h,\ell}^{\bar{h}}, t_{ij}{\upharpoonright}h)$ in $P_h^{\bar{h}}$ such that there exists $m \in [1:n]$:*
     - *(i) $s_{ij}(B_{h,m}^{\bar{h}}) = true$,*
     - *(ii) $< s_{ij}{\upharpoonright}\mathcal{SH}_{ij} > A_{h,m}^{\bar{h}} < t_{ij}{\upharpoonright}\mathcal{SH}_{ij} >$, and*
     - *(iii) $s_{ij}{\upharpoonright}\bar{h} = t_{ij}{\upharpoonright}\bar{h}$.*

   Here $\bar{h} = i$ if $h = j$ and $\bar{h} = j$ if $h = i$.

[5]This interpretation was proposed by [Dij82].

In a transition (s_{ij}, h, t_{ij}), we say that s_{ij} is the *start* state and that t_{ij} is the *finish* state. The transition (s_{ij}, h, t_{ij}) is called a $P_h^{\bar{h}}$-transition. In the sequel, we use $s_{ij} \xrightarrow{h} t_{ij}$ as an alternative notation for the transition (s_{ij}, h, t_{ij}). $< s_{ij}\lceil\mathcal{SH}_{ij} > A < t_{ij}\lceil\mathcal{SH}_{ij} >$ is Hoare triple notation [Hoa69] for total correctness, which in this case means that execution of A always terminates,[6] and, when the shared variables in $\mathcal{SH}_{ij}$ have the values assigned by s_{ij}, leaves these variables with the values assigned by t_{ij}. $s_{ij}(B_h^{\bar{h}}) = true$ states that the value of guard $B_h^{\bar{h}}$ in state s_{ij} is *true*.[7] We consider that $(S_{ij}^0, P_i^j \,\|\, P_j^i)$ possesses a correctness property expressed by an CTL* formula f_{ij} if and only if $M_{ij}, S_{ij}^0 \models f_{ij}$.

The semantics of $(S_I^0, P_{i_1}^I \,\|\ldots\|\, P_{i_K}^I)$ is given by the global state transition diagram M_I generated by its execution. We call the global state transition diagram of an I-system an *I-structure*.

Definition 4 (I-structure) *The semantics of* $(S_I^0, P_{i_1}^I \,\|\ldots\|\, P_{i_K}^I)$ *is given by the* I-structure $M_I = (S_I^0, S_I, R_I)$ *where*

1. *S_I is a set of I-states,*
2. *$S_I^0 \subseteq S_I$ gives the initial states of $(S_I^0, P_{i_1}^I \,\|\ldots\|\, P_{i_K}^I)$, and*
3. *$R_I \subseteq S_I \times dom(I) \times S_I$ is a transition relation giving the transitions of $(S_I^0, P_{i_1}^I \,\|\ldots\|\, P_{i_K}^I)$. A transition (s, i, t) by P_i is in R_I if and only if*
   (a) *$i \in dom(I)$,*
   (b) *s and t are I-states, and*
   (c) *there exists a move $(s\lceil i, \otimes_{j\in I(i)} \oplus_{\ell\in[1:n]} B_{i,\ell}^j \to A_{i,\ell}^j, t\lceil i)$ in P_i such that all of the following hold:*
      (i) *for all j in $I(i)$, there exists $m \in [1:n]$:*
      $s\lceil ij(B_{i,m}^j) = true$ *and* $< s\lceil\mathcal{SH}_{ij} > A_{i,m}^j < t\lceil\mathcal{SH}_{ij} >$,
      (ii) *for all j in $dom(I) - \{i\}$: $s\lceil j = t\lceil j$, and*
      (iii) *for all j, k in $dom(I) - \{i\}$, $j\, I\, k$: $s\lceil\mathcal{SH}_{jk} = t\lceil\mathcal{SH}_{jk}$.*

In a transition (s, i, t), we say that s is the *start* state, and t is the *finish* state. The transition (s, i, t) is called a P_i-transition. In the sequel, we use $s \xrightarrow{i} t$ as alternative notation for the transition (s, i, t). Also, if I is set to $\{\{i, j\}\}$ in Definition 4, then the result is, as expected, the pair-structure definition (3). In other words, the two definitions are consistent. Furthermore, the semantics of a J-system,$J \subseteq I$ is given by the J-structure $M_J = (S_J^0, S_J, R_J)$, which is obtained by using J for I in Definition 4.

As M_I gives the semantics of $(S_I^0, P_{i_1}^I \,\|\ldots\|\, P_{i_K}^I)$, we consider that $(S_I^0, P_{i_1}^I \,\|\ldots\|\, P_{i_K}^I)$ possesses a correctness property expressed by a formula $\bigwedge_{k\ell} f_{k\ell}$ if and only if $M_I, S_I^0 \models \bigwedge_{k\ell} f_{k\ell}$, i.e., $M_I, S_I^0 \models \forall (i,j) \in I : (f_{ij})$.

M_{ij} and M_I can be interpreted as CTL* structures. We call M_{ij} a *pair-structure*, since it gives the semantics of a pair-program, ad M_I an *I-structure*, since it gives the semantics of an I-program. We state our main soundness result below by relating the ACTL formulae that hold in M_I to those that hold in M_{ij}.

[6]Termination is obvious, since the right-hand side of A is a list of constants.

[7]$s_{ij}(B_h^{\bar{h}})$ is defined by the usual inductive scheme: $s_{ij}(``x_{ij} = h_{ij}") = true$ iff $s_{ij}(x_{ij}) = h_{ij}$, $s_{ij}(B1_h^{\bar{h}} \wedge B2_h^{\bar{h}}) = true$ iff $s_{ij}(B1_h^{\bar{h}}) = true$ and $s_{ij}(B2_h^{\bar{h}}) = true$, $s_{ij}(\neg B1_h^{\bar{h}}) = true$ iff $s_{ij}(B1_h^{\bar{h}}) = false$.

This characterization of transitions in the I-program as compositions of transitions in all the relevant pair-programs is formalized in the transition mapping lemma:

Lemma 1 (Transition mapping [AE98]) *For all I-states $s, t \in S_I$ and $i \in dom(I)$, $s \xrightarrow{i} t \in R_I$* iff :

$$\begin{aligned}
&\forall j \in I(i) : (s{\upharpoonright}ij \xrightarrow{i} t{\upharpoonright}ij \in R_{ij}) \text{ and} \\
&\forall j \in \{i_1, \ldots, i_K\} - \hat{I}(i) : (s{\upharpoonright}j = t{\upharpoonright}j) \text{ and} \\
&\forall j, k \in \{i_1, \ldots, i_K\} - \{i\}, j\, I\, k : (s{\upharpoonright}\mathcal{SH}_{jk} = t{\upharpoonright}\mathcal{SH}_{jk}).
\end{aligned}$$

Proof. This was established in [AE98] as Lemma 6.4.1. The proof there did not assume that the M_{ij} are isomorphic. Hence, it carries over to the setting of this paper. □

In similar manner, we establish:

Corollary 2 (Transition mapping [AE98]) *Let $J \subseteq I$ and $i \in dom(J)$. If $s \xrightarrow{i} t \in R_I$, then $s{\upharpoonright}J \xrightarrow{i} t{\upharpoonright}J \in R_J$.*

By applying the transition-mapping corollary to every transition along a path π in M_I, we show that $\pi{\upharpoonright}J$ is a path in M_J. Again, the proof carries over from [AE98].

Lemma 3 (Path mapping [AE98]) *Let $J \subseteq I$. If π is a path in M_I, then $\pi{\upharpoonright}J$ is a path in M_J.*

In particular, when $J = \{(i, j, spec_{ij})\}$, Lemma 3 forms the basis for our soundness proof, since it relates computations of the synthesized program to computations of the pair-programs.

Since every reachable state lies at the end of some initialized path, we can use the path-mapping corollary to relate reachable states in M_I to their projections in M_J:

Corollary 4 (State mapping [AE98]) *Let $J \subseteq I$. If t is a reachable state in M_I, then $t{\upharpoonright}J$ is a reachable state in M_J.*

5 Soundness of the Method for Static Programs

5.1 Deadlock-freedom

As we showed in [AE98], it is possible for the synthesized program P to be deadlock-prone even though all the pair-programs are deadlock-free. To ensure deadlock-freedom of P, we imposed a condition on the "blocking behavior" of processes: after a process executes a move, it must either have another move enabled, or it must not be blocking any other process. In general, any behavioral condition which prevents the occurrence of certain patterns of blocking ("supercycles") is sufficient.

We formalize our notion of blocking behavior by the notion of *wait-for-graph*. The wait-for-graph in a particular I-state s contains as nodes all the processes, and all the moves whose start state is a component of s. These moves have an outgoing edge to every process which blocks them.

Definition 5 (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* $\{P_i \mid i \in dom(I)\}$, *and*
   (b) *the moves* $\{a_i^I \mid i \in dom(I) \text{ and } a_i^I \in P_i \text{ and } s{\upharpoonright}i = a_i^I.start\}$
2. *there is an edge from* P_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 *in* $W_I(s)$ *if and only if* $i\,I\,j$ *and* $a_i^I \in W_I(s)$ *and* $s{\upharpoonright}ij(a_i^I.guard_j) = false$.

Here $a_i^I.guard_j$ is the conjunct of the guard of move a_i^I which is evaluated over the (pairwise) shared state with P_j. We characterize a deadlock as the occurrence in the wait-for-graph of a graph-theoretic construct that we call a *supercycle*:

Definition 6 (Supercycle) *SC is a supercycle in $W_I(s)$ if and only if all of the following hold:*

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

Note that this definition implies that SC is a subgraph of $W_I(s)$.

Our conditions will be stated over "small" programs, i.e,. programs that result from compositing a small number of processes together. To then infer that the large program P has similar behavior, we use the following proposition.

Proposition 5 (Wait-for-graph projection) *Let $J \subseteq I$ and $i\,J\,j$. Furthermore, let s_I be an arbitrary I-state. Then*

1. *$P_i \longrightarrow a_i^I \in W_I(s_I)$ iff $P_i \longrightarrow a_i^J \in W_J(s_I{\upharpoonright}J)$, and*
2. *$a_i^I \longrightarrow P_j \in W_I(s_I)$ iff $a_i^J \longrightarrow P_j \in W_J(s_I{\upharpoonright}J)$.*

Proof. By assumption, $i\,J\,j$ and $J \subseteq I$. Hence $i\,I\,j$.

Proof of clause (1). By the wait-for-graph definition (5), $P_i \longrightarrow a_i^I \in W_I(s_I)$ iff $s_I{\upharpoonright}i = a_i^I.start$. Since $i \in dom(J)$, we have $(s_I{\upharpoonright}J){\upharpoonright}i = s_I{\upharpoonright}i$ by definition of ${\upharpoonright}J$. Thus $s_I{\upharpoonright}i = a_i^I.start$ iff $(s_I{\upharpoonright}J){\upharpoonright}i = a_i^J.start$ (since $a_i^I.start = a_i^J.start = s_i$). Finally, by the wait-for-graph definition (5) and $i\,J\,j$, $(s_I{\upharpoonright}J){\upharpoonright}i = a_i^J.start$ iff $P_i \longrightarrow a_i^J \in W_J(s_I{\upharpoonright}J)$. These three equivalences together yield clause (1) (using transitivity of equivalence).

Proof of clause (2). By the wait-for-graph definition (5), $a_i^I \longrightarrow P_j \in W_I(s_I)$ iff $s{\upharpoonright}ij \not\models a_i^I.guard_j$. Since $i\,J\,j$, we have $(s_I{\upharpoonright}J){\upharpoonright}ij = s_I{\upharpoonright}ij$ by definition of ${\upharpoonright}J$. Also, $a_i^I.guard_j = a_i^J.guard_j = \bigvee_{\ell \in [1:n]} B_{i,\ell}^j$. Thus $s_I{\upharpoonright}ij \not\models a_i^I.guard_j$ iff $(s_I{\upharpoonright}J){\upharpoonright}ij \not\models a_i^J.guard_j$ Finally, by the wait-for-graph definition (5) and $i\,J\,j$, $(s_I{\upharpoonright}J){\upharpoonright}ij \not\models a_i^J.guard_j$ iff $a_i^J \longrightarrow P_j \in W_J(s_I{\upharpoonright}J)$. These three equivalences together yield clause (2), (using transitivity of equivalence, and noting that $s \not\models B$ and $s(B) = false$ have identical meaning). $\square$

5.1.1 The Wait-for-graph Condition

In [AE98], we give a criterion, the wait-for-graph assumption, which can be 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. The wait-for-graph condition embodies the requirement that, after a process executes a move, it must either have another move enabled, or it must not be blocking any other process.

Definition 7 (Static wait-for-graph condition) *Let t_k be an arbitrary reachable local state of P_k^ℓ in $M_{k\ell}$ for all $\ell \in I(k)$, and let $n = |t_k.moves|$. Also let J be an arbitrary interconnection relation such that $J \subseteq I$ and J has the form $\{(j,k,spec_{jk}), (k,\ell_1,spec_{k\ell_1}), \dots, (k,\ell_n,spec_{k\ell_n})\}$, where $k \notin \{j,\ell_1,\dots,\ell_n\}$. Then, for every reachable J-state t_J in M_J such that $t_J{\upharpoonright}k = t_k$ and $s_J \xrightarrow{k} t_J \in R_J$ for some reachable J-state s_J, we have*

$$\forall a_j^J : (a_j^J \longrightarrow P_k \notin W_J(t_J))$$

or

$$\exists a_k^J \in W_J(t_J) : (\forall \ell \in \{\ell_1,\dots,\ell_n\} : a_k^J \longrightarrow P_\ell \notin W_J(t_J)).$$

Theorem 6 (Static supercycle-free wait-for-graph) *If the wait-for-graph condition holds, and $W_I(s_I^0)$ is supercycle-free for every initial state $s_I^0 \in S_I^0$, then for every reachable state t of M_I, $W_I(t)$ is supercycle-free.*

Proof. Let t be an arbitrary reachable state of M_I, and let s be an arbitrary reachable state of M_I such that $s \xrightarrow{k} t$ for some $k \in dom(I)$. We shall establish that

$$\text{if } W_I(t) \text{ is supercyclic, then } W_I(s) \text{ is supercyclic.} \qquad \text{(P1)}$$

The contrapositive of P1 together with the assumption that $W_I(s_I^0)$ is supercycle-free for all $s_I^0 \in S_I^0$ is sufficient to establish the conclusion of the theorem (by induction on the length of a path from some $s_I^0 \in S_I^0$ to t).

We say that an edge is *k-incident* iff at least one of its vertices is P_k or a_k^I. The following (P2) will be useful in proving P1

$$\text{if edge } e \text{ is not } k\text{-incident, then } e \in W_I(t) \text{ iff } e \in W_I(s). \qquad \text{(P2)}$$

Proof of P2. If e is not k-incident, then, by the wait-for-graph definition (5), either $e = P_h \longrightarrow a_h^I$, or $e = a_h^I \longrightarrow P_\ell$, for some h, ℓ such that $h \neq k, \ell \neq k$. From $h \neq k, \ell \neq k$ and $s \xrightarrow{k} t \in R_I$, we have $s{\upharpoonright}h = t{\upharpoonright}h$ and $s{\upharpoonright}h\ell = t{\upharpoonright}h\ell$ by the wait-for-graph definition (5). Since $e \in W_I(t), e \in W_I(s)$ are determined solely by $t{\upharpoonright}h\ell, s{\upharpoonright}h\ell$ respectively, (see the wait-for-graph definition (5), P2 follows. (End proof of P2.)

Let v be a vertex in a supercycle SC. We define $depth_{SC}(v)$ to be the length of the longest backward path in SC which starts in v. If there exists an infinite backward path (i.e., one that traverses a cycle) in SC starting in v, then $depth_{SC}(v) = \omega$ (ω for "infinity"). We now establish that

$$\text{every supercycle } SC \text{ contains at least one cycle.} \qquad \text{(P3)}$$

Proof of P3. Suppose P3 does not hold, and SC is a supercycle containing no cycles. Therefore, all backward paths in SC are finite, and so by definition of $depth_{SC}$ all vertices of SC have finite depth. Thus, there is at least one vertex v in SC with maximal depth. But, by definition of $depth_{SC}$, v has no successors in SC, which, by the supercycle definition (6), contradicts the assumption that SC is a supercycle. (End proof of P3.)

Our final prerequisite for the proof of P1 is

if SC is a supercycle in $W_I(s)$, then the graph SC' obtained from SC by removing all vertices of finite depth from SC (along with incident edges) is also a supercycle in $W_I(s)$. (P4)

Proof of P4. By P3, $SC' \neq \emptyset$. Thus SC' satisfies clause (1) of the supercycle definition (6). Let v be an arbitrary vertex of SC'. Thus $v \in SC$ and $depth_{SC}(v) = \omega$ by definition of SC'. Let w be an arbitrary successor of v in SC. $depth_{SC}(w) = \omega$ by definition of *depth*. Hence $w \in SC'$. Furthermore, w is a successor of v in SC', by definition of SC'. Thus every vertex v of SC' is also a vertex of SC, and the successors of v in SC' are the same as the successors of v in SC Now since SC is a supercycle, every vertex v in SC has enough successors in SC to satisfy clauses (2) and (3) of the supercycle definition (6). It follows that every vertex v in SC' has enough successors in SC' to satisfy clauses (2) and (3) of the supercycle definition (6). (End proof of P4.)

We now present the proof of (P1). We assume the antecedent of P1 and establish the consequent. Let SC be some supercycle in $W_I(t)$. Let SC' be the graph obtained from SC by removing all vertices of finite depth from SC (along with incident edges). We now show that $P_k^I \notin SC'$ and that SC' contains no move vertex of the form a_k^I. There are two cases.

Case 1: $P_k \notin SC$. Then obviously $P_k \notin SC'$. Now suppose some node of the form a_k^I is in SC'. By definition of SC', we have $a_k^I \in SC$ and $depth_{SC}(a_k^I) = \omega$. Hence, by definition of *depth*, there exists an infinite backward path in SC starting in a_k^I. Thus a_k^I must have a predecessor in SC. By the supercycle definition (6), P_k is the only possible predecessor of a_k^I in SC, and hence $P_k \in SC$, contrary to the case assumption. We therefore conclude that SC' contains no vertices of the form a_k^I. (End of case 1.)

Case 2: $P_k \in SC$. By the supercycle definition (6),

$$\forall a_k^I \in W_I(t) : (\exists \ell : (a_k^I \longrightarrow P_\ell \in W_I(t))). \tag{a}$$

Since there are exactly n moves a_k^I of process P_k^I in $W_I(t)$ ($n = |t_k.moves|$), we can select $\ell_1, \ldots, \ell_n$ (where $\ell_1, \ldots, \ell_n$ are not necessarily pairwise distinct) such that

$$\forall a_k^I \in W_I(t) : (\exists \ell \in \{\ell_1, \ldots, \ell_n\} : (a_k^I \longrightarrow P_\ell \in W_I(t))). \tag{b}$$

Now let $J = \{\{j,k\}, \{k,\ell_1\}, \ldots, \{k,\ell_n\}\}$ where j is an arbitrary element of $I(k)$. Applying the wait-for-graph projection proposition (5) to (b) gives us

$$\forall a_k^J \in W_J(t{\restriction}J) : (\exists \ell \in \{\ell_1, \ldots, \ell_n\} : (a_k^J \longrightarrow P_\ell \in W_J(t{\restriction}J))). \tag{c}$$

Now $s \xrightarrow{k} t \in R_I$ by assumption. Hence $s{\restriction}J \xrightarrow{k} t{\restriction}J \in R_J$ by the transition-mapping corollary (2). Also, by the state-mapping corollary (4) $s{\restriction}J$ is reachable in M_J, since s is reachable in M_I. Thus we can apply the wait-for-graph assumption to $t{\restriction}J$ to get

$$\forall a_j^J : (a_j^J \longrightarrow P_k \notin W_J(t{\restriction}J))$$

or

$$\exists a_k^J \in W_J(t{\restriction}J) : (\forall \ell \in \{\ell_1, \ldots, \ell_n\} : (a_k^J \longrightarrow P_\ell \notin W_J(t{\restriction}J))). \tag{d}$$

Now (c) contradicts the second disjunct of (d). Hence

$$\forall a_j^J : (a_j^J \longrightarrow P_k \notin W_J(t{\restriction}J)),$$

and applying the wait-for-graph projection proposition (5) to this gives us

$$\forall a_j^J : (a_j^I \longrightarrow P_k \notin W_I(t)).$$

Since j is an arbitrary element of $I(k)$, we conclude that P_k has no incoming edges in $W_I(t)$. Thus, by definition of *depth*, $depth_{SC}(P_k) = 0$, and so $P_k \notin SC'$.

Now suppose some node of the form a_k^I is in SC'. By definition of SC', we have $a_k^I \in SC$ and $depth_{SC}(a_k^I) = \omega$. Hence, by definition of *depth*, there exists an infinite backward path in SC starting in a_k^I. Thus a_k^I must have a predecessor in SC. By the supercycle definition (6), P_k is the only possible predecessor of a_k^I in SC, and hence there exists an infinite backward path in SC starting in P_k^I. Thus $depth_{SC}(P_k^I) = \omega$ by definition of *depth*. But we have established $depth_{SC}(P_k) = 0$, so we conclude that SC' contains no vertices of the form a_k^I. (End of case 2.)

In both cases, $P_k^I \notin SC'$, and SC' contains no move vertex of the form a_k^I. Thus every edge of SC' is not k-incident. Hence, by P2, every edge of SC' is an edge of $W_I(s)$ (since $SC' \subseteq W_I(t)$). By P4, SC' is a supercycle, so $W_I(s)$ is supercyclic. Thus P1 is established, which establishes the theorem. □

5.1.2 Establishing Deadlock-freedom

We show that the absence of supercycles in the wait-for-graph of a state implies that there is at least one enabled move in that state.

Proposition 7 (Supercycle [AE98]) *If $W_I(s)$ is supercycle-free, then some move a_i^I has no outgoing edges in $W_I(s)$.*

Proof. We establish the contrapositive. Since every local state of a process has at least one outgoing arc (Section 2), there exists at least one move of the form a_i^I for every $i \in dom(I)$ in $W_I(s)$. Suppose that every such move has at least one outgoing edge in $W_I(s)$. Consider the subgraph SC of $W_I(s)$ consisting of these edges together with all edges of the form $P_i \longrightarrow a_i^I$ in $W_I(s)$. By the wait-for-graph definition (5), and the supercycle definition (6), it is clear that SC is a supercycle in $W_I(s)$. Thus $W_I(s)$ is not supercycle-free. □

Proposition 8 (Move enablement) *Let s be an arbitrary I-state such that $s{\upharpoonright}i = a_i^I.start$. If a_i^I has no outgoing edges in $W_I(s)$, then a_i^I can be executed in state s.*

Proof. If a_i^I has no outgoing edges in $W_I(s)$, then by the wait-for-graph definition (5), $s{\upharpoonright}ij(a_i^I.guard_j) = true$ for all $j \in I(i)$. Hence, by the I-structure definition (4), a_i^I can be executed in state s. □

Theorem 9 (Deadlock freedom [AE98]) *If, for every reachable state s of M_I, $W_I(s)$ is supercycle-free, then $M_I, S_I^0 \models \mathsf{AGEX}true$.*

Proof. Let s be an arbitrary reachable state of M_I. By the antecedent, $W_I(s)$ is supercycle-free. Hence, by the supercycle proposition (7), some move a_i^I has no outgoing edges in $W_I(s)$. By Proposition 8, a_i^I can be executed in state s. Since s is an arbitrary reachable state of M_I, we conclude that every reachable state of M_I has at least one enabled move a_i^I, (where, in general, a_i^I depends on s). Hence $M_I, S_I^0 \models \mathsf{AGEX}true$. □

5.2 Liveness

To assure liveness properties of the synthesized programs, we need to 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. We give our fairness criterion as a formula of the linear time temporal logic PTL [MW84].

Definition 8 (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, $blk_i \overset{\text{df}}{=} (\bigvee \{\!\!\{s_i\}\!\!\} : s_i$ is sometimes-blocking in $M_{ij})$, and $blk_i \overset{\text{df}}{=} \bigvee_{j \in I(i)} blk_i^j$.

Note that $a^i_j.start$ is the start state of the two-process move a^i_j, and $a^i_j.guard$ is its guard.

Definition 9 (Weak blocking fairness Φ_b) $\Phi_b \overset{\text{df}}{=} \bigwedge_{i \in dom(I)} \overset{\infty}{\mathsf{G}}(blk_i \wedge en_i) \Rightarrow \overset{\infty}{\mathsf{F}} ex_i$.

Definition 10 (Pending eventuality, pnd_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 11 (Weak eventuality fairness, Φ_ℓ)

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

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 12 (Liveness condition for static programs) *For every reachable state s_{ij} in M_{ij},*
$M_{ij}, s_{ij} \models \mathsf{A}(\mathsf{G} ex_i \Rightarrow \overset{\infty}{\mathsf{G}} aen_j)$,
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 move of P^i_j whose start state is a component of the current global state is also enabled in the current global state. The liveness condition requires, in every pair-program $(S^0_{ij}, P^j_i \,\|\, P^i_j)$, 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 move of P^i_j.

Lemma 10 (Progress for static programs) *If*

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

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

Proof. By assumption 2 and Theorem 9, $M_I, S^0_I \models \mathsf{AGEX}\mathit{true}$. Hence every fullpath in M_I is infinite. Let π be an arbitrary Φ-fair fullpath starting in s. If $M_I, \pi \models \mathsf{F}(ex_i \vee ex_j)$, then we are done. Hence we assume

$$\pi \models \mathsf{G}(\neg ex_i \wedge \neg ex_j) \qquad (*)$$

in the remainder of the proof. Now define $\psi_{inf} \overset{\text{df}}{=} \{k \mid \pi \models \overset{\infty}{\mathsf{F}} ex_k\}$ and $\psi_{fin} \overset{\text{df}}{=} \{k \mid \pi \models \overset{\infty}{\mathsf{G}} \neg ex_k\}$.

Let ρ be a suffix of π such that no process in ψ_{fin} executes along ρ, and let t be the first state of ρ. Note that, by (*), $i \in \psi_{fin}$, $j \in \psi_{fin}$.

Let W be the portion of $W_I(t)$ induced by starting in P_i, P_j and following wait-for edges that enter processes in ψ_{fin} or their moves. By assumption 2, W is supercycle-free. Hence, there exists a process P_k in W such that P_k has some move a_k^I with no wait-for edges to any process in W, by Proposition 7. Hence, in state $t{\upharpoonright}k\ell$, a_k^ℓ is enabled in all pair-machines $M_{k\ell}$ such that $\ell \in \psi_{fin}$, i.e., $t{\upharpoonright}k\ell \models \{\!|a_k^\ell.start|\!\} \wedge en(a_k^\ell)$. Also, $k \in \psi_{fin}$, by definition of W. Since t is the first state of ρ and no process in ψ_{fin} executes along ρ, we have from above, that $\bigwedge \ell \in \psi_{fin} \cap I(k) : \rho{\upharpoonright}k\ell \models \mathsf{G}en(a_k^\ell)$.

Now consider a pair-machine $M_{k\ell}$ such that $\ell \in \psi_{inf}$ (if any). Hence $\rho \models \overset{\infty}{\mathsf{F}}ex_\ell \wedge \mathsf{G}\neg ex_k$, since $k \in \psi_{fin}$. Hence $\rho{\upharpoonright}k\ell \models \mathsf{G}ex_\ell \wedge \mathsf{G}\neg ex_k$. By Lemma 3, $\rho{\upharpoonright}k\ell$ is a path in $M_{k\ell}$. Since ρ is an infinite path and $\rho \models \overset{\infty}{\mathsf{F}}ex_\ell$, $\rho{\upharpoonright}k\ell$ is an infinite path. Hence $\rho{\upharpoonright}k\ell$ is a fullpath in $M_{k\ell}$. By the liveness condition for static programs (Definition 12), $\rho{\upharpoonright}k\ell \models \overset{\infty}{\mathsf{G}}aen_k$. Now $t{\upharpoonright}k\ell \models \{\!|a_k^\ell.start|\!\}$. Since $\rho{\upharpoonright}k\ell \models \mathsf{G}\neg ex_k$, P_k's local state does not change along $\rho{\upharpoonright}k\ell$. Hence $\rho{\upharpoonright}k\ell \models \mathsf{G}\{\!|a_k^\ell.start|\!\}$. Hence, by definition of aen_k, $\rho{\upharpoonright}k\ell \models \overset{\infty}{\mathsf{G}}en(a_k^\ell)$. Since ℓ is an arbitrary element of $\psi_{inf} \cap I(k)$, we have $\bigwedge \ell \in \psi_{inf} \cap I(k) : \rho{\upharpoonright}k\ell \models \overset{\infty}{\mathsf{G}}en(a_k^\ell)$. Since $(\psi_{inf} \cap I(k)) \cup (\psi_{fin} \cap I(k)) = I(k)$, we conclude $\bigwedge \ell \in I(k) : \rho{\upharpoonright}k\ell \models \overset{\infty}{\mathsf{G}}en(a_k^\ell)$. By Definitions 1 and 2, we have $\rho \models \overset{\infty}{\mathsf{G}}en(a_k^I)$. Hence, we conclude

$$\rho \models \overset{\infty}{\mathsf{G}}en_k. \qquad \text{(a)}$$

Assume $k \notin \{i, j\}$. Then, by definition of W, in state t P_k blocks some move a_ℓ^k of some process P_ℓ, i.e., $t \models \{\!|a_\ell^k.start|\!\} \wedge \neg a_\ell^k.guard$. By Definition 8, $t{\upharpoonright}k$ is sometimes-blocking in $M_{k\ell}$ (since t is reachable, so is $t{\upharpoonright}k$, by [AE98, Corollary 6.4.5]). Hence $t{\upharpoonright}k \models blk_k^\ell$, and so $t \models blk_k^\ell$. Now $\rho \models \mathsf{G}\neg ex_k$. Since t is the first state of ρ, this means that $t{\upharpoonright}k = u{\upharpoonright}k$ for any state u of ρ, i.e., the local state of P_k does not change along ρ. Thus, $\rho \models \mathsf{G}blk_k^\ell$, since $t \models blk_k^\ell$. Thus $\rho \models \mathsf{G}blk_k$, by definition of blk_k. From this and (a), we have $\rho \models \overset{\infty}{\mathsf{G}}(blk_k \wedge en_k)$. Hence, by weak blocking fairness, (Definition 9), $\rho \models \overset{\infty}{\mathsf{F}}ex_k$, which contradicts $\rho \models \mathsf{G}\neg ex_k$. Hence the assumption $k \notin \{i, j\}$ does not hold, and so $k \in \{i, j\}$.

Since $\pi \models \mathsf{G}(\neg ex_i \wedge \neg ex_j)$, by assumption (*), and $s = first(\pi)$, we have $u{\upharpoonright}ij = s{\upharpoonright}ij$ for every state u along π. Now $M_{ij}, s{\upharpoonright}ij \models \neg h_{ij} \wedge \mathsf{AF}h_{ij}$ for some $h_{ij} \in CL(spec_{ij})$ by assumption 3. Hence $M_{ij}, u{\upharpoonright}ij \models \neg h_{ij} \wedge \mathsf{AF}h_{ij}$ for all u along π. Hence $M_{ij}, u{\upharpoonright}ij \models pnd_{ij}$ for all u along π by Definition 10. Hence, $M_I, u \models pnd_{ij}$ for all u along π, since pnd_{ij} is purely propositional, and so $M_I, \pi \models \mathsf{G}pnd_{ij}$. Since ρ is a suffix of π and $k \in \{i, j\}$, we conclude from (a) that $\pi \models \overset{\infty}{\mathsf{G}}en_i \vee \overset{\infty}{\mathsf{G}}en_j$. Hence $M_I, \pi \models (\overset{\infty}{\mathsf{G}}en_i \vee \overset{\infty}{\mathsf{G}}en_j) \wedge \overset{\infty}{\mathsf{G}}pnd_{ij}$. By weak eventuality fairness (Definition 11), $\pi \models \overset{\infty}{\mathsf{F}}(ex_i \vee ex_j)$. This contradicts the assumption (*), which is therefore false. Hence $\pi \models \mathsf{F}(ex_i \vee ex_j)$. Since π is an arbitrary Φ-fair fullpath starting in s, the lemma follows. □

5.3 The Large Model Theorem for Static Programs

Theorem 11 (Large model) *Let $(i, j, spec_{ij}) \in I$, where $spec_{ij} \in \mathrm{ACTL}_{ij}^-$, and let s be an arbitrary reachable I-state. If*

1. *the liveness condition for static programs holds,*

2. $W_I(u)$ is supercycle-free for every reachable I-state u, and

3. $M_{ij}, s{\upharpoonright}ij \models f_{ij}$ for some $f_{ij} \in CL(spec_{ij})$,

then

$$M_I, s \models_\Phi f_{ij}.$$

Proof. The proof is by induction on the structure of f_{ij}. Throughout, let $s_{ij} = s{\upharpoonright}ij$.

$f_{ij} = p_i$, or $f_{ij} = \neg p_i$, where $p_i \in \mathcal{AP}_i$, i.e., p_i is an atomic proposition.
By definition of ${\upharpoonright}ij$, s and $s{\upharpoonright}ij$ agree on all atomic propositions in $\mathcal{AP}_i \cup \mathcal{AP}_j$. The result follows.

$f_{ij} = g_{ij} \wedge h_{ij}$. The antecedent is $M_{ij}, s_{ij} \models g_{ij} \wedge h_{ij}$. So, by CTL* semantics, $M_{ij}, s_{ij} \models g_{ij}$ and $M_{ij}, s_{ij} \models h_{ij}$. Since $f_{ij} \in CL(spec_{ij})$, we have $g_{ij} \in CL(spec_{ij})$ and $h_{ij} \in CL(spec_{ij})$. Hence, applying the induction hypothesis, we get $M_I, s \models_\Phi g_{ij}$ and $M_I, s \models_\Phi h_{ij}$. So by CTL* semantics we get $M_I, s \models_\Phi (g_{ij} \wedge h_{ij})$.

$f_{ij} = g_{ij} \vee h_{ij}$. The antecedent is $M_{ij}, s_{ij} \models g_{ij} \vee h_{ij}$. So, by CTL* semantics, $M_{ij}, s_{ij} \models g_{ij}$ or $M_{ij}, s_{ij} \models h_{ij}$. Since $f_{ij} \in CL(spec_{ij})$, we have $g_{ij} \in CL(spec_{ij})$ and $h_{ij} \in CL(spec_{ij})$. Hence, applying the induction hypothesis, we get $M_I, s \models_\Phi g_{ij}$ or $M_I, s \models_\Phi h_{ij}$. So by CTL* semantics we get $M_I, s \models_\Phi (g_{ij} \vee h_{ij})$.

$f_{ij} = \mathsf{A}[g_{ij} \mathsf{U_w} h_{ij}]$. Let π be an arbitrary Φ-fair fullpath starting in s. We establish $\pi \models [g_{ij} \mathsf{U_w} h_{ij}]$. By Definition 1, $\pi{\upharpoonright}ij$ starts in $s{\upharpoonright}ij = s_{ij}$. Hence, by CTL semantics, $\pi{\upharpoonright}ij \models [g_{ij} \mathsf{U_w} h_{ij}]$ (note that this holds even if $\pi{\upharpoonright}ij$ is not a fullpath, i.e., is a finite path). We have two cases.

Case 1: $\pi{\upharpoonright}ij \models \mathsf{G} g_{ij}$. Let t be an arbitrary state along π. By Definition 1, $t{\upharpoonright}ij$ lies along $\pi{\upharpoonright}ij$. Hence $t{\upharpoonright}ij \models g_{ij}$. By the induction hypothesis, $t \models g_{ij}$. Hence $\pi \models \mathsf{G} g_{ij}$, since t was arbitrarily chosen. Hence $\pi \models [g_{ij} \mathsf{U_w} h_{ij}]$ by CTL*semantics.

Case 2: $\pi{\upharpoonright}ij \models [g_{ij} \mathsf{U} h_{ij}]$. Let $s_{ij}^{m'}$ be the first state along $\pi{\upharpoonright}ij$ that satisfies h_{ij}[8]. By Definition 1, there exists at least one state t along π such that $t{\upharpoonright}ij = s_{ij}^{m'}$. Let $s^{n'}$ be the first such state. By the induction hypothesis, $s^{n'} \models h_{ij}$. Let s^n be any state along π up to but not including $s^{n'}$ (i.e., $0 \le n < n'$). Then, by Definition 1, $s^n{\upharpoonright}ij$ lies along the portion of $\pi{\upharpoonright}ij$ up to, and possibly including, $s_{ij}^{m'}$. That is, $s^n{\upharpoonright}ij = s_{ij}^m$, where $0 \le m \le m'$. Now suppose $s^n{\upharpoonright}ij = s_{ij}^{m'}$ (i.e., $m = m'$). Then, by $s_{ij}^{m'} \models h_{ij}$ and the induction hypothesis, $s^n \models h_{ij}$, contradicting the fact that $s^{n'}$ is the first state along π that satisfies h_{ij}. Hence, $m \neq m'$, and so $0 \le m < m'$. Since $s_{ij}^{m'}$ is the first state along $\pi{\upharpoonright}ij$ that satisfies h_{ij}, and $\pi{\upharpoonright}ij \models [g_{ij} \mathsf{U} h_{ij}]$, we have $s_{ij}^m \models g_{ij}$ by CTL*semantics. From $s^n{\upharpoonright}ij = s_{ij}^m$ and the induction hypothesis, we get $s^n \models g_{ij}$. Since s^n is any state along π up to but not including $s^{n'}$, and $s^{n'} \models h_{ij}$, we have $\pi \models [g_{ij} \mathsf{U} h_{ij}]$ by CTL*semantics. Hence $\pi \models [g_{ij} \mathsf{U_w} h_{ij}]$ by CTL*semantics.

In both cases, we showed $\pi \models [g_{ij} \mathsf{U_w} h_{ij}]$. Since π is an arbitrary Φ-fair fullpath starting in s, we conclude $M_I, s \models_\Phi \mathsf{A}[g_{ij} \mathsf{U_w} h_{ij}]$.

$f_{ij} = \mathsf{A}[g_{ij} \mathsf{U} h_{ij}]$. Since $f_{ij} \in CL(spec_{ij})$, we have $g_{ij} \in CL(spec_{ij})$ and $h_{ij} \in CL(spec_{ij})$. Suppose $s_{ij} \models h_{ij}$. Hence $s \models h_{ij}$ by the induction hypothesis, and so $s \models \mathsf{A}[g_{ij} \mathsf{U} h_{ij}]$ and we are done.

[8]We use s_{ij}^n to denote the n'th state along $\pi{\upharpoonright}ij$, i.e., $\pi{\upharpoonright}ij = s_{ij}^0, s_{ij}^1, \ldots$, and we let $s_{ij} = s_{ij}^0$.

Hence we assume $s_{ij} \models \neg h_{ij}$ in the remainder of the proof. Since $s_{ij} \models \mathsf{A}[g_{ij} \mathsf{U} h_{ij}]$ by assumption, we have $s_{ij} \models \neg h_{ij} \wedge \mathsf{AF} h_{ij}$. Let π be an arbitrary Φ-fair fullpath starting in s. By Theorem 9, π is an infinite path. We now establish $\pi \models_{\Phi} \mathsf{F} h_{ij}$.

Proof of $\pi \models_{\Phi} \mathsf{F} h_{ij}$. Assume $\pi \models_{\Phi} \neg \mathsf{F} h_{ij}$, i.e., $\pi \models_{\Phi} \mathsf{G} \neg h_{ij}$. Let t be an arbitrary state along π. Let ρ be the segment of π from s to t. By Definition 1, $\rho{\upharpoonright}ij$ is a path from s_{ij} to $t{\upharpoonright}ij$. By Lemma 3, $\rho{\upharpoonright}ij$ is a path in M_{ij}. Suppose $\rho{\upharpoonright}ij$ contains a state u_{ij} such that $u_{ij} \models h_{ij}$. By Definition 1, there exists a state u along ρ such that $u{\upharpoonright}ij = u_{ij}$. By the induction hypothesis, we have $u \models_{\Phi} h_{ij}$, contradicting the assumption $\pi \models_{\Phi} \mathsf{G} \neg h_{ij}$. Hence $\rho{\upharpoonright}ij$ contains no state that satisfies h_{ij}. Since $s_{ij} \models \mathsf{AF} h_{ij}$ and $\rho{\upharpoonright}ij$ is a path from s_{ij} to $t{\upharpoonright}ij$ (inclusive) which contains no state satisfying h_{ij}, we must have $t{\upharpoonright}ij \models \neg h_{ij} \wedge \mathsf{AF} h_{ij}$ by CTL semantics. Let π' be the suffix of π starting in t. Since $t{\upharpoonright}ij \models \neg h_{ij} \wedge \mathsf{AF} h_{ij}$ and $h_{ij} \in CL(spec_{ij})$, we can apply the Progress Lemma to conclude $M_I, t \models_{\Phi} \mathsf{AF}(ex_i \vee ex_j)$. Since t is an arbitrary state along π, we conclude $M_I, \pi \models \overset{\infty}{\mathsf{F}}(ex_i \vee ex_j)$. Hence, by Definition 1, $\pi{\upharpoonright}ij$ is a fullpath. By Lemma 3, $\pi{\upharpoonright}ij$ is a fullpath in M_{ij}. Since $\pi{\upharpoonright}ij$ starts in $s_{ij} = s{\upharpoonright}ij$, and $s_{ij} \models \mathsf{AF} h_{ij}$, $\pi{\upharpoonright}ij$ must contain a state v_{ij} such that $v_{ij} \models h_{ij}$. By Definition 1, π contains a state v such that $v{\upharpoonright}ij = v_{ij}$. By the induction hypothesis and $v_{ij} \models h_{ij}$, we have $v \models_{\Phi} h_{ij}$. Hence $\pi \models_{\Phi} \mathsf{F} h_{ij}$, contrary to assumption, and we are done. (End of proof of $\pi \models_{\Phi} \mathsf{F} h_{ij}$).

By assumption, $s_{ij} \models \mathsf{A}[g_{ij} \mathsf{U} h_{ij}]$. Hence $s_{ij} \models \mathsf{A}[g_{ij} \mathsf{U_w} h_{ij}]$. From the above proof case for $\mathsf{A}[g_{ij} \mathsf{U_w} h_{ij}]$, we have $s \models_{\Phi} \mathsf{A}[g_{ij} \mathsf{U_w} h_{ij}]$. Hence $\pi \models_{\Phi} [g_{ij} \mathsf{U_w} h_{ij}]$, since π is a Φ-fair fullpath starting in s. From this and $\pi \models_{\Phi} \mathsf{F} h_{ij}$, we have $\pi \models_{\Phi} [g_{ij} \mathsf{U} h_{ij}]$ by CTL*semantics. Since π is an arbitrary Φ-fair fullpath starting in s, we have $s \models_{\Phi} \mathsf{A}[g_{ij} \mathsf{U} h_{ij}]$. □

Corollary 12 (Large model) *If the liveness condition for static programs holds, and $W_I(u)$ is supercycle-free for every reachable I-state u, then*

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

Unlike [AE98], $spec_{ij}$ and $spec_{k\ell}$, where $\{k, \ell\} \neq \{i, j\}$, can be completely different formulae, whereas in [AE98] these formulae had to be "similar," i.e., one was obtained from the other by substituting process indices.

6 Example—A Two Phase Commit Protocol

We illustrate our method by synthesizing a ring-based (non fault tolerant) two-phase commit protocol $P^I = P_0 \,\|\, P_1 \,\|\, \cdots \,\|\, P_{n-1}$, where I specifies a ring. P_0 is the *coordinator*, and $P_i, 1 \leq i < n$ 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 can submit only after it observes that P_i has submitted. After the first cycle, the coordinator observes the state of P_{n-1}. If P_{n-1} has submitted its transaction, that means that all participants have submitted their transactions, and so the coordinator decides commit. If P_{n-1} has aborted, that means that some participant P_i unilaterally aborted thereby causing all participants $P_j, i < j \leq n-1$ to abort. In that case, the coordinator decides abort. The second cycle then relays the coordinators decision around the ring. The participant processes are all similar to each other, but the coordinator is not similar to the participants. Hence, there are three pair-programs to consider: $P^0_{n-1} \,\|\, P^{n-1}_0$, $P^1_0 \,\|\, P^0_1$, and $P^i_{i-1} \,\|\, P^{i-1}_i$. These are given in Figures 1, 2, and 3, respectively, where $term_i \overset{\text{df}}{=} cm_i \vee ab_i$, and an incoming arrow with no source indicates an initial local state. Figures 5, 6, and 7 give the respective global state transition diagrams (i.e., pair-structures). The synthesized two phase commit protocol P^I is given in Figure 4. We establish the correctness of P^I as follows:

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

Here the formula $f \to g$ abbreviates $\mathsf{A}[(f \Rightarrow \mathsf{AF}g)\mathsf{U_w}g]$, which intuitively means that if f holds at some point, 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 $\mathsf{AG}[f \Rightarrow \mathsf{AF}g]$. The above formula hold in all initial states of M_I, the global state transition diagram of P^I. The notation LMT means that the formula was established first in the relevant pair structure, and then we used the large model theorem to deduce that the formula also hols in M^I. A notation of some formula numbers means that the formula was deduced using the preceding formulae, and using an appropriate CTL deductive system [Eme90]. Formula 11 gives us a correctness property of two phase commit: if the coordinator commits, then so does every participant. Using the large model theorem, we deduce $\bigwedge_{1\le i<n}(ab_{i-1} \to ab_i)$, from which $ab_0 \to \bigwedge_{1\le i<n} ab_i$ follows, namely if the coordinator aborts, then so does every participant. Likewise, we establish $\mathsf{AF}(cm_0 \vee ab_0)$ (the coordinator eventually decides), and $\bigwedge_{1\le i<n} \mathsf{AG}(st_i \Rightarrow \mathsf{EX}_i ab_i)$ (every participant can abort unilaterally). This last formula is not in ACTL^-_{ij}, but it was shown to be preserved in [AE98], and we have extended the proof there to the setting of this paper.

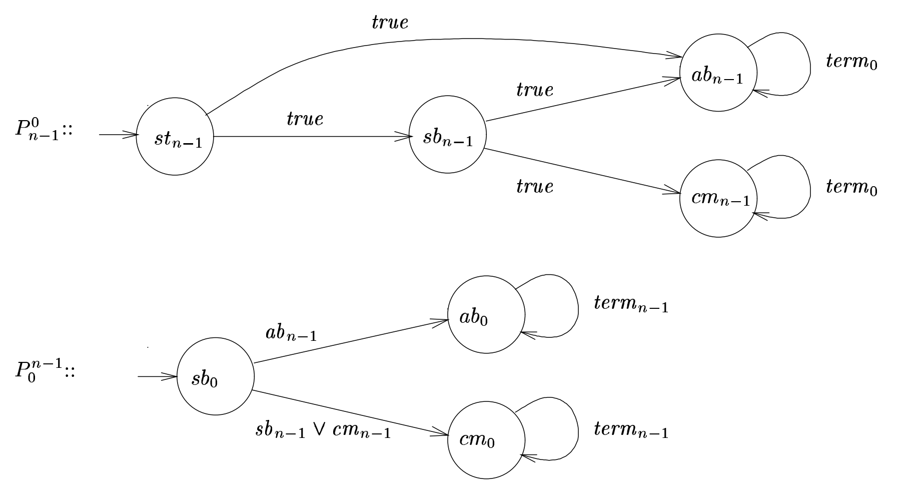

Figure 1: Pair program $P^0_{n-1} \,\|\, P^{n-1}_0$.

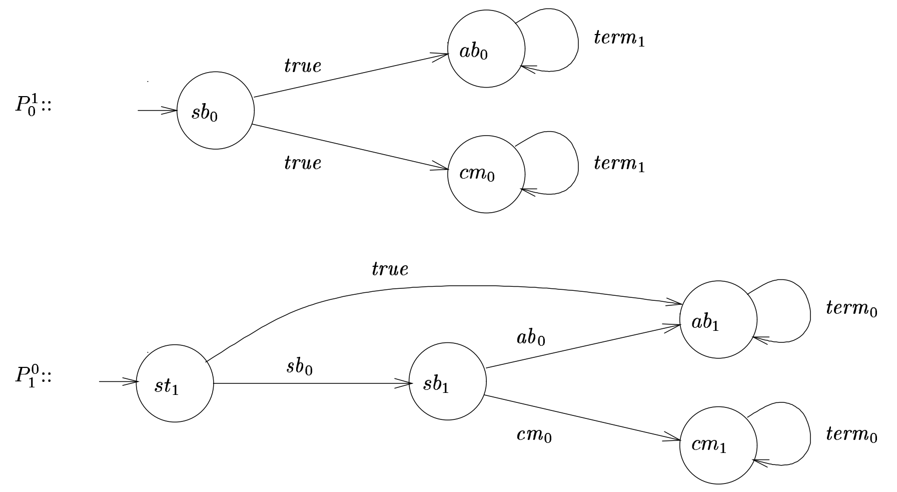

Figure 2: Pair program $P^1_0 \,\|\, P^0_1$.

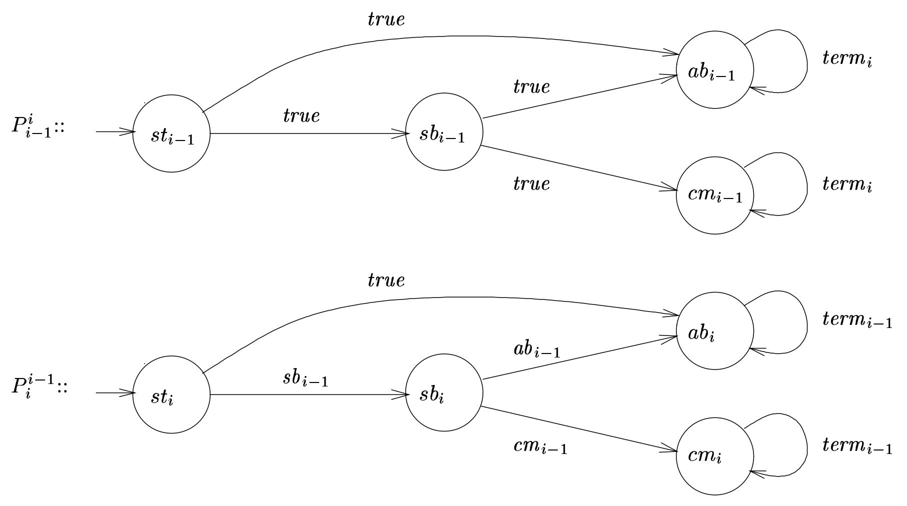

Figure 3: Pair program $P^i_{i-1} \| P^{i-1}_i$.

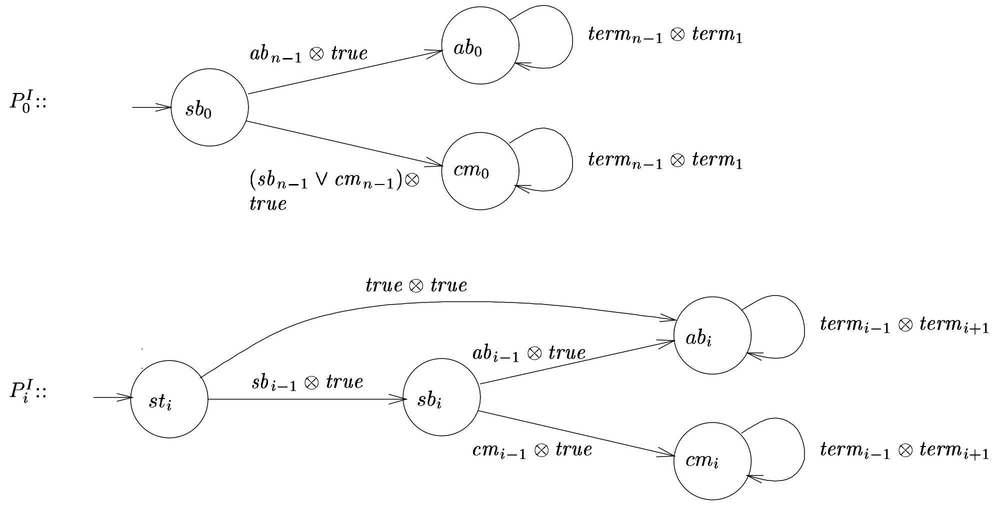

Figure 4: The synthesized two phase commit protocol $P^I = P^I_0 \| (\|_{1 \le i < n} P^I_i)$.

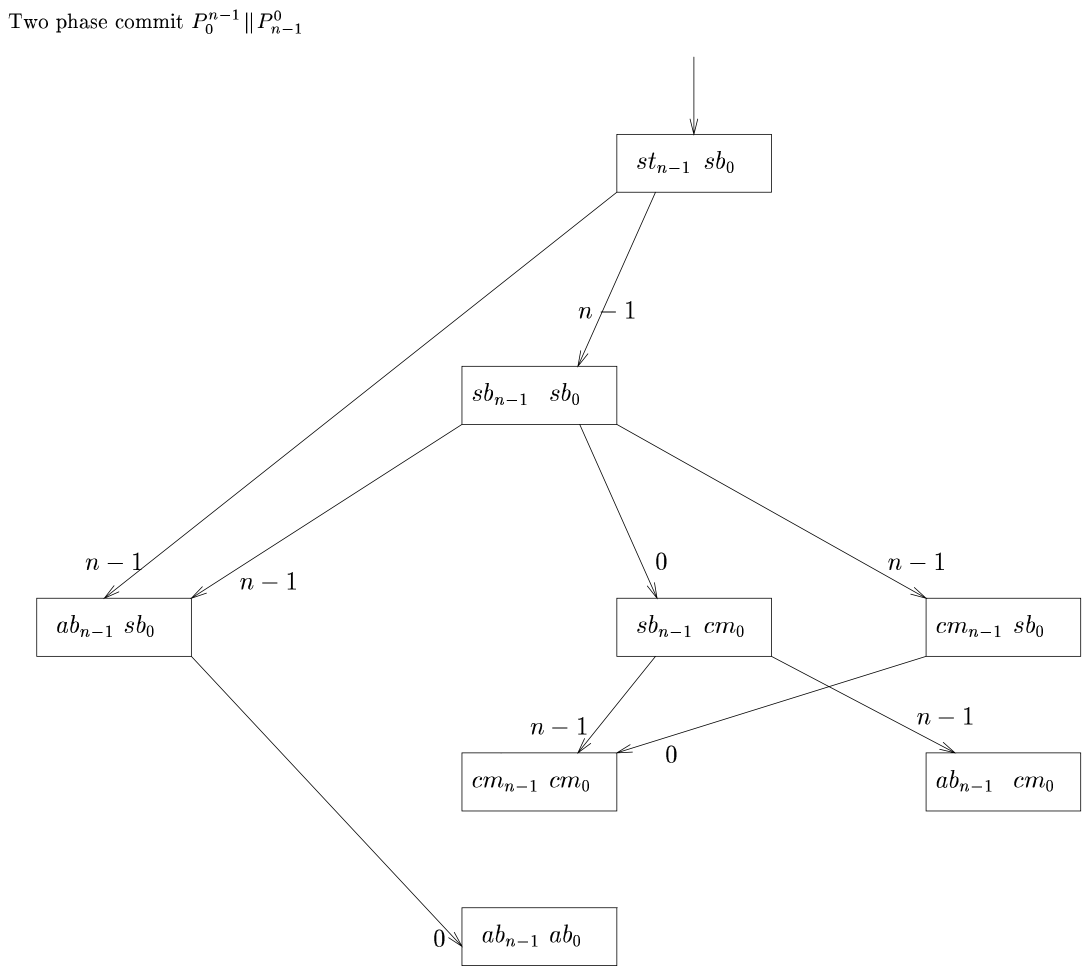

Figure 5: Global state transition diagram of the pair-program $P_{n-1}^0 \| P_0^{n-1}$.

Two Phase Commit $P_0^1 \| P_1^0$

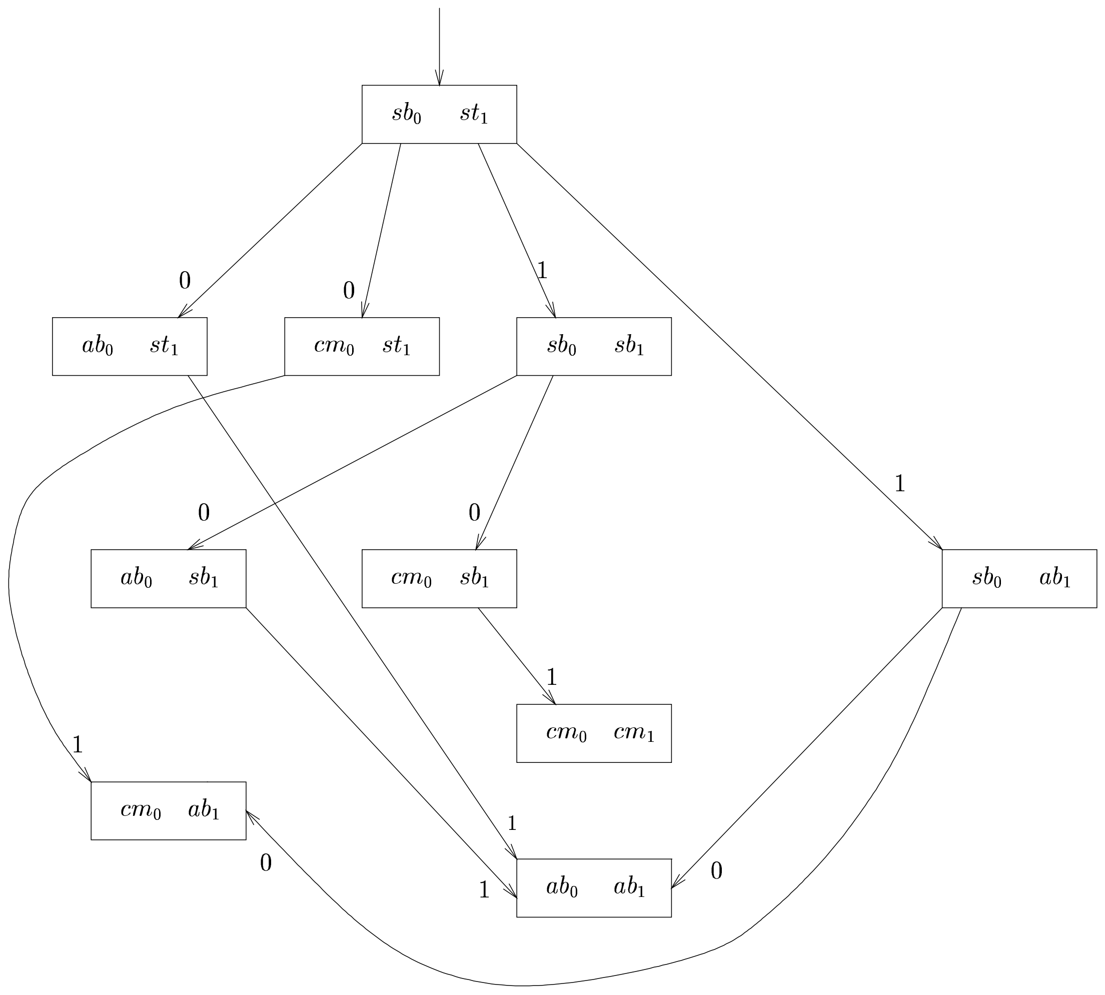

Figure 6: Global state transition diagram of the pair-program $P_0^1 \| P_1^0$.

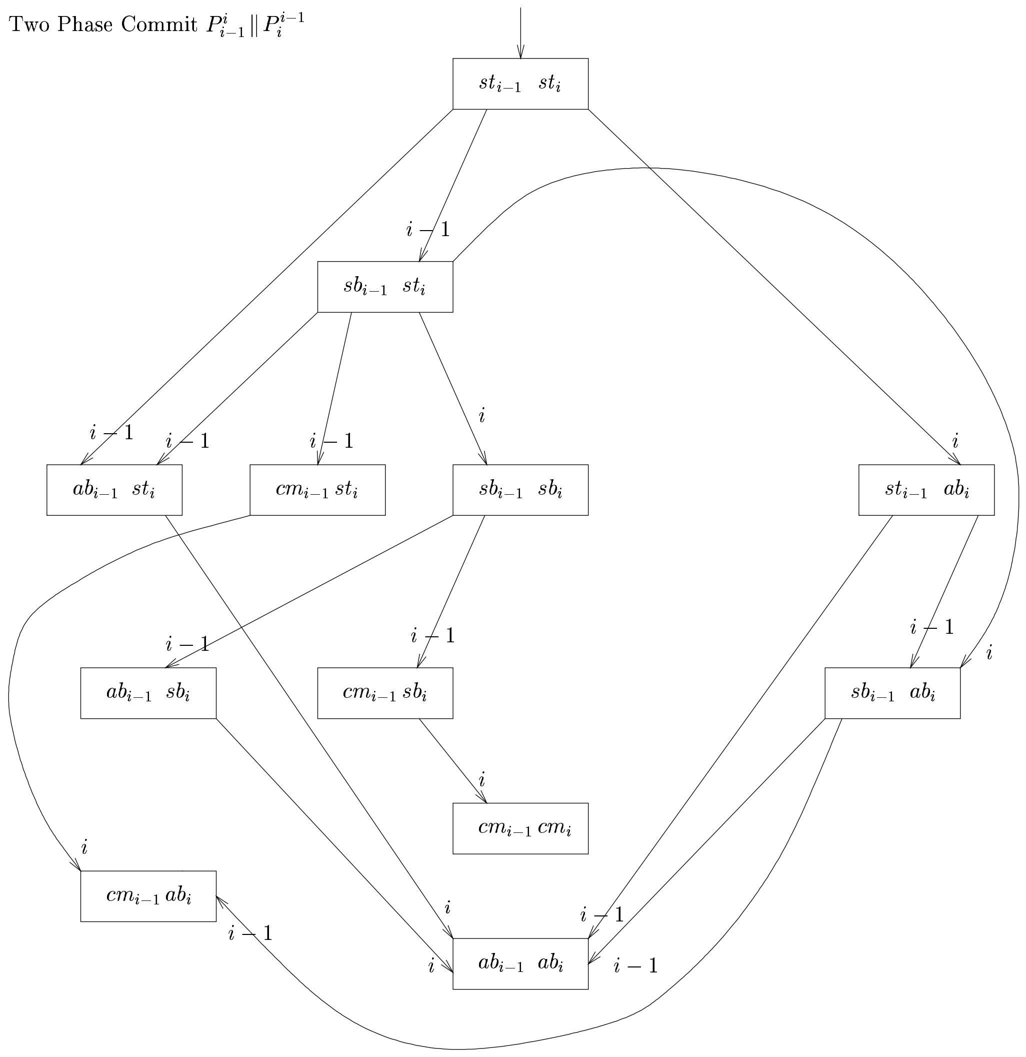

Figure 7: Global state transition diagram of the pair-program $P^{i}_{i-1} \| P^{i-1}_{i}$.

7 Synthesis of Dynamic Concurrent Programs

7.1 Dynamic Specifications

A *dynamic specification* consists of:

1. A "universal" set $\mathcal{UI}$ of *pair-specifications*. A pair-specification has the form $\langle\!\langle \{i, j\}, spec_{ij} \rangle\!\rangle$, where $i, j \in \mathsf{Pids}$, $i \neq j$, and $spec_{ij} \in \mathrm{ACTL}^-_{ij}$ specifies the interaction of processes i and j. $\mathcal{UI}$ can be infinite.
2. A finite set $\mathcal{I}_0 \subseteq \mathcal{UI}$, which gives the pair-specifications which are *in force*, that is, must be satisfied, initially.
3. A mapping $create : 2^{\mathcal{UI}} \mapsto 2^{\mathcal{UI}}$ which determines which new pair-specifications (in $\mathcal{UI}$) can be added to those that are in-force. If $\mathcal{I}$ is the set of pair-specifications that are in-force and $\langle\!\langle \{i, j\}, spec_{ij} \rangle\!\rangle \in create(\mathcal{I})$, then $\mathcal{I} \cup \{\langle\!\langle \{i, j\}, spec_{ij} \rangle\!\rangle\}$ is a possible next value for the set of pair-specifications in-force.

We show in the sequel that the synthesized dynamic program satisfies the dynamic specification in that every pair-specification is satisfied from the time it comes into force. We make these notions precise below.

7.2 Overview of the Synthesis Method: Dynamic Addition of Pair-programs

Our synthesis method produces a dynamic concurrent program $\mathcal{P}$. $\mathcal{P}$ consists of the *conjunctive overlay* of a dynamically increasing set of *pair-programs*. A pair-program is a static concurrent program consisting of exactly two processes. $(S^0_{ij}, P^j_i \parallel P^i_j)$ denotes a pair-program with processes i and j, and initial state set S^0_{ij}. We use P^j_i for the synchronization skeleton of process i within this pair-program, with the superscript j indicating the other process. $\mathcal{SH}_{i,j}$ denotes the shared variables in $(S^0_{ij}, P^j_i \parallel P^i_j)$. The shared variable sets of different pair-programs are disjoint: $\mathcal{SH}_{ij} \cap \mathcal{SH}_{i'j'} = \emptyset$ if $\{i, j\} \neq \{i', j'\}$. The component processes of a pair-program (e.g., P^j_i) are called *pair-processes*. Define $graph(P_i)$ to be the synchronization skeleton of P_i with all the arc labels removed.

Definition 13 (Conjunctive overlay, $P^j_i \otimes P^k_i$) *Let P^j_i and P^k_i be pair-processes for i such that $graph(P^j_i) = graph(P^k_i)$. Then,*

$P^j_i \otimes P^k_i$ contains an arc from s_i to t_i with label $(\oplus_{\ell \in [1:n_j]} B^j_{i,\ell} \to A^j_{i,\ell}) \otimes (\oplus_{\ell \in [1:n_k]} B^k_{i,\ell} \to A^k_{i,\ell})$
iff
P^j_i contains an arc from s_i to t_i with label $\oplus_{\ell \in [1:n_j]} B^j_{i,\ell} \to A^j_{i,\ell}$ and
P^k_i contains an arc from s_i to t_i with label $\oplus_{\ell \in [1:n_k]} B^k_{i,\ell} \to A^k_{i,\ell}$.

Note that the $\otimes$ operator is overloaded, and applies to both pair-processes and to guarded commands. When applied to guarded commands, $\otimes$ denotes the "conjunction" of guarded commands, so an arc with label $(\oplus_{\ell \in [1:n_j]} B^j_{i,\ell} \to A^j_{i,\ell}) \otimes (\oplus_{\ell \in [1:n_k]} B^k_{i,\ell} \to A^k_{i,\ell})$ can only be executed in a state in which $B^j_{i,\ell}$ holds for some $\ell \in [n_j]$ and $B^k_{i,\ell}$ holds for some $k \in [n_j]$. Execution then involves the parallel execution of the corresponding $A^j_{i,\ell}$ and $A^k_{i,\ell}$. See [AE98] for a full discussion of $\oplus$ and $\otimes$. Conjunctive overlay viewed as a binary operation on both guarded commands and pair-processes is commutative and associative, since the operands of $\otimes$ are treated identically. Thus, we define and use the n-ary version of $\otimes$ in the usual manner.

Given a dynamic concurrent program $\mathcal{P}$, a new pair-program $(S^0_{ij}, P^j_i \parallel P^i_j)$ can be dynamically added at run-time as follows. If $\mathcal{P}$ already contains P_i, then P_i is modified by taking the conjunctive overlay with P^j_i, i.e., $P_i := P_i \otimes P^j_i$. If $\mathcal{P}$ does not contain P_i, then P_i is dynamically created and added as a new process, and is given the synchronization skeleton of P^j_i, i.e., $P_i := P^j_i$. Likewise for P_j. We say that $(S^0_{ij}, P^j_i \parallel P^i_j)$ is *active* once it has been added. The "synchronization skeleton code" of the dynamic program thus changes at run time, as pair-programs are added. Since each P_i built up by successive conjunctive overlays of pair-processes, the n-ary version of the $\otimes$ operator can always be applied, provided that $graph(P_i) = graph(P^j_i)$. To assure this, we assume, in the sequel,

$$\text{For active pair-programs } (S^0_{ij}, P^j_i \parallel P^i_j) \text{ and } (S^0_{ik}, P^k_i \parallel P^i_k)\text{: } graph(P^j_i) = graph(P^k_i).$$

We emphasize that different pair-programs can have different functionality, since the guarded commands which label the arcs of P^j_i and P^k_i can be different.

Pair-programs are added only when a new pair-specification comes into force, and is the means of satisfying the new pair-specification. Thus, the transitions of $\mathcal{P}$ are of two kinds: (1) *normal* transitions, which are atomic transitions (as described in Section 2) arising from execution of the conjunctive overlay of all active pair-programs, and (2) *create* transitions, which correspond to making a new pair-specification $\langle\{i,j\}, spec_{ij}\rangle$ in-force, according to the *create* mapping. To satisfy $\langle\{i,j\}, spec_{ij}\rangle$, we dynamically create a new pair-program $(S^0_{ij}, P^j_i \parallel P^i_j)$ such that $(S^0_{ij}, P^j_i \parallel P^i_j) \models spec_{i,j}$, and incorporate it into the existing dynamic program by performing a conjunctive overlay with the currently active pair-programs.

7.3 Technical Definitions

If $\mathcal{I} \subseteq \mathcal{UI}$, then define $pairs(\mathcal{I}) = \{\{i,j\} \mid \exists spec_{ij} : \langle\{i,j\}, spec_{ij}\rangle \in \mathcal{I}\}$, and $procs(\mathcal{I}) = \{i \mid \exists j : \{i,j\} \in pairs(\mathcal{I})\}$, and $\mathcal{I}(i) = \{j \mid \{i,j\} \in pairs(I)\}$. Processes i and j are *neighbors* when $\{i,j\} \in pairs(\mathcal{I})$. If $\mathcal{I} \neq \emptyset$, then $\mathcal{I}(i) \neq \emptyset$ for all $i \in procs(I)$, by definition. Thus, every process always has at least one neighbor.

An *i-state* is a local state of P^j_i. An ij-state is a global state of $(S^0_{ij}, P^j_i \parallel P^i_j)$, i.e., (by Section 2) a tuple $(s_i, s_j, v^1_{ij}, \ldots, v^m_{ij})$ where s_i, s_j are i-states, j-states, respectively, and $v^1_{ij}, \ldots, v^m_{ij}$ give the values of all the variables in $\mathcal{SH}_{ij}$. When i and j are unspecified, we refer to an ij-state as a pair-state.

A *configuration* is a tuple $\langle\mathcal{I}, \mathcal{A}, \mathcal{S}\rangle$, where $\mathcal{I} \subseteq \mathcal{UI}$, $\mathcal{A}$ is a set of pair-programs $(S^0_{ij}, P^j_i \parallel P^i_j)$, one for each $\{i,j\} \in pairs(\mathcal{I})$, and $\mathcal{S}$ is a mapping from each $\{i,j\} \in pairs(\mathcal{I})$ to an ij-state. We refer to the components of s as $s.\mathcal{I}$, $s.\mathcal{A}$, $s.\mathcal{S}$. We write $procs(s)$ for $procs(s.\mathcal{I})$, and $pairs(s)$ for $pairs(s.\mathcal{I})$. A *consistent configuration* satisfies the constraint that all pair-states assign the same local state to all common processes, i.e., for all $\{i,j\},\{i,k\} \in pairs(s)$, if $\mathcal{S}(\{i,j\}) = (s_i, s_j, v^1_{ij}, \ldots, v^m_{ij})$ and $\mathcal{S}(\{i,k\}) = (s'_i, s_k, v^1_{ik}, \ldots, v^m_{ik})$, then $s_i = s'_i$. We assume henceforth that configurations are consistent, and our definitions will respect this constraint.

For configuration s, $i \in procs(s)$, and atomic proposition $p_i \in \mathcal{AP}_i$, we define $s(p_i) = \mathcal{S}(\{i,j\})(p_i)$, where $\{i,j\} \in pairs(s)$. By the above definitions and constraints, a j such that $\{i,j\} \in pairs(s)$ always exists when $i \in procs(s)$, and the value for $s(p_i)$ so defined is unique.

The state-to-formula operator $\{\!\!|s_i|\!\!\}$ converts an i-state s_i into a propositional formula: $\{\!\!|s_i|\!\!\} = (\bigwedge_{s_i(p_i)=true} p_i) \wedge (\bigwedge_{s_i(p_i)=false} \neg p_i)$, where p_i ranges over the members of $\mathcal{AP}_i$. $\{\!\!|s_i|\!\!\}$ characterizes

s_i in that $s_i \models \{\!\!\{ s_i \}\!\!\}$, and $s'_i \not\models \{\!\!\{ s_i \}\!\!\}$ for all $s'_i \neq s_i$. $\{\!\!\{ s_{ij} \}\!\!\}$ is defined similarly (but note that the variables in $\mathcal{SH}_{ij}$ must be accounted for).

We define the *state projection operator* $\upharpoonright$, which is an overloaded binary infix operator with several variants, depending on the type of the operands. For projection of ij-states onto a single process: if $s_{ij} = (s_i, s_j, v^1_{ij}, \ldots, v^m_{ij})$, then $s_{ij}\upharpoonright i = s_i$. For projection of ij-states onto the shared variables in $\mathcal{SH}_{ij}$: if $s_{ij} = (s_i, s_j, v^1_{ij}, \ldots, v^m_{ij})$, then $s_{ij}\upharpoonright\mathcal{SH}_{ij} = (v^1_{ij}, \ldots, v^m_{ij})$. For projection of a configuration $s = \langle\!\langle \mathcal{I}, \mathcal{A}, \mathcal{S} \rangle\!\rangle$ onto a single process: if $i \in \mathit{procs}(s)$, then $s\upharpoonright i = \mathcal{S}(\{i,j\})\upharpoonright i$, where $\{i,j\} \in \mathit{pairs}(s)$. This is unique because configurations are consistent. For projection of s onto a pair-program: if $\{i,j\} \in \mathit{pairs}(s)$, then $s\upharpoonright ij = \mathcal{S}(\{i,j\})$. If $\{i,j\} \notin \mathit{pairs}(s)$, then $s\upharpoonright ij$ is undefined. If J is a set of pairs such that $J \subseteq \mathit{pairs}(s)$, then we define the projection of s onto J: $s\upharpoonright J$ is the restriction of $s.\mathcal{S}$ to J.

7.4 The Synthesis Method

Given a dynamic specification, we synthesize a program $\mathcal{P}$ as follows:

1. Initially, $\mathcal{P}$ consists of the conjunctive overlay of the pair-programs corresponding to the pair-specifications in $\mathcal{I}_0$.
2. When a pair-specification $\langle\!\langle \{i,j\}, \mathit{spec}_{ij} \rangle\!\rangle$ is added, as permitted by the *create* mapping, synthesize a *pair-program* $(S^0_{ij}, P^j_i \,\|\, P^i_j)$ using spec_{ij} as the specification, and add it to $\mathcal{P}$ as discussed in Section 7.2 above.

To synthesize pair-programs, any synthesis method which produces static concurrent programs in the synchronization skeleton notation can be used, e.g., [AAE98, AE01, EC82].

Since the create transitions affect the actual code of $\mathcal{P}$, we define them first. The create transitions are determined by the intended meaning of the *create* rule, together with the constraint that creating a new pair-program does not change the current state of existing pair-programs.

Definition 14 (Create transitions) *Let s, t be configurations. Then (s, create, t) is a* create transition *iff there exists $\{i,j\} \notin \mathit{pairs}(s)$ such that*

1. *$\langle\!\langle \{i,j\}, \mathit{spec}_{ij} \rangle\!\rangle \in \mathit{create}(s)$, i.e., the rule for adding new pair-specifications allows the pair-specification $\langle\!\langle \{i,j\}, \mathit{spec}_{ij} \rangle\!\rangle$ to be added in global-state s.*
2. *$t.\mathcal{I} = s.\mathcal{I} \cup \langle\!\langle \{i,j\}, \mathit{spec}_{ij} \rangle\!\rangle$, and*
   $t.\mathcal{A} = s.\mathcal{A} \cup \{(S^0_{ij}, P^j_i \,\|\, P^i_j)\}$, where $(S^0_{ij}, P^j_i \,\|\, P^i_j) \models \mathit{spec}_{ij}$.
3. *$t\upharpoonright ij$ is a reachable state of $(S^0_{ij}, P^j_i \,\|\, P^i_j)$, and if $i \in \mathit{procs}(s)$ then $t\upharpoonright i = s\upharpoonright i$, and if $j \in \mathit{procs}(s)$ then $t\upharpoonright j = s\upharpoonright j$*
4. *for all $\{k,\ell\} \in \mathit{pairs}(s) : s.\mathcal{S}(\{k,\ell\}) = t.\mathcal{S}(\{k,\ell\})$*

Instead of a process index, we use a constant label create to indicate a create transition.

Our synthesis method is given by the following.

Definition 15 (Pairwise synthesis) *In configuration s, the synthesized program $\mathcal{P}$ is $\|_{i \in \mathit{procs}(s)}$ P_i, where $P_i = \otimes_{j \in s.\mathcal{I}(i)} P^j_i$.*

The set of initial configurations S_0 of $\mathcal{P}$ consists of all s such that (1) $s.\mathcal{I} = \mathcal{I}_0$, (2) $s.\mathcal{A}$ contains exactly one pair-program $(S^0_{ij}, P^j_i \| P^i_j)$ for each $\langle\{i,j\}, spec_{ij}\rangle \in \mathcal{I}_0$, (3) $(S^0_{ij}, P^j_i \| P^i_j) \models spec_{ij}$, and (4) $s.\mathcal{S}(\{i,j\}) \in S^0_{ij}$ for all $\{i,j\} \in pairs(s)$.

Another way to characterize process P_i of $\mathcal{P}$ is that $(s_i, \otimes_{j \in s.\mathcal{I}(i)} \oplus_{\ell \in [1:n_j]} B^j_{i,\ell} \to A^j_{i,\ell}, t_i)$ is an arc in P_i iff $\forall j \in s.\mathcal{I}(i) : (s_i, \oplus_{\ell \in [1:n_j]} B^j_{i,\ell} \to A^j_{i,\ell}, t_i)$ is an arc in P^j_i. Definition 15 gives the initial configurations S_0 of $\mathcal{P}$, and the code of $\mathcal{P}$ as a function of the $s.\mathcal{I}$ and $s.\mathcal{A}$ components of the current configuration s. The code of $\mathcal{P}$ does not depend on the the $s.\mathcal{S}$ component of s, which gives the values of the atomic propositions and shared variables, i.e., the state. Definition 14 shows how $s.\mathcal{I}$ and $s.\mathcal{A}$ are changed by create transitions. We assume that the S^0_{ij} are such that $S_0 \neq \emptyset$, i.e., there exist consistent configurations that project onto a state in each S^0_{ij}.

Since a configuration of $\mathcal{P}$ determines both the state and the code of all processes, the normal transitions that can be executed in a configuration are determined intrinsically by that configuration, Definition 15, and the semantics of synchronization skeletons, as follows.

Definition 16 (Normal transitions) *Let s, t be configurations and $i \in procs(s)$. Then (s, i, t) is a* normal transition *iff*

1. *there exist local states $s{\restriction}i$, $t{\restriction}i$ of P_i such that, for all $\{i,j\} \in pairs(s)$, there exists an arc $(s{\restriction}i, \oplus_{\ell \in [n_j]} B^j_{i,\ell} \to A^j_{i,\ell}, t{\restriction}i)$ in P^j_i such that*
   $\exists m \in [n_j] : s{\restriction}ij(B^j_{i,m}) = true$ and
   $< (s{\restriction}ij){\restriction}\mathcal{SH}_{ij} > A^j_{i,m} < (t{\restriction}ij){\restriction}\mathcal{SH}_{ij} >$
2. *for all j in $procs(s) - \{i\}$: $s{\restriction}j = t{\restriction}j$, and*
3. *for all $\{j,k\}$ in $pairs(s)$, $i \notin \{j,k\}$: $s{\restriction}jk = t{\restriction}jk$.*
4. *$s.\mathcal{I} = t.\mathcal{I}$ and $s.\mathcal{A} = t.\mathcal{A}$*

Thus, P_i can execute a transition from global state s to global state t only if, for every $\{i,j\} \in pairs(s)$, P^j_i can execute a transition from $s{\restriction}ij$ to $t{\restriction}ij$. Also, P_i reads the local state of its neighbors, and reads/writes variables that are shared pairwise, i.e., between P_i and exactly one neighbor. Thus $\mathcal{P}$ enjoys a *spatial locality* property, which is useful when implementing $\mathcal{P}$ in atomic read/write memory.

$< (s{\restriction}ij){\restriction}\mathcal{SH}_{ij} > A < (t{\restriction}ij){\restriction}\mathcal{SH}_{ij} >$ is Hoare triple notation [Hoa69] for total correctness, which in this case means that execution of A always terminates,[9] and, when the shared variables in $\mathcal{SH}_{ij}$ have the values assigned by $s{\restriction}ij$, leaves these variables with the values assigned by $t{\restriction}ij$. $s{\restriction}ij(B^j_{i,m}) = true$ states that the value of guard $B^j_{i,m}$ in state s_{ij} is *true*.

The semantics of the synthesized program $\mathcal{P}$ is given by its global state transition diagram (GSTD), which is obtained by starting with the initial configurations, and taking the closure under all the normal and create transitions.

Definition 17 (Global-state transition diagram of $\mathcal{P}$) *The semantics of $\mathcal{P}$ is given by the structure $M_{\mathcal{P}} = (S_0, S, R_n, R_c)$ where*

[9]Termination is obvious, since the right-hand side of A is a list of constants.

1. *S_0 is the set of initial configurations of $\mathcal{P}$, and consists of all the configurations s_0 such that $s_0 = \langle \mathcal{I}_0, \mathcal{A}, \mathcal{S} \rangle$, $\mathcal{A} = \{(S^0_{ij}, P^j_i \,\|\, P^i_j) \mid \{i,j\} \in pairs(\mathcal{I}_0)\}$, and $\mathcal{S}(\{i,j\}) \in S^0_{ij}$, i.e., the pair-specifications in $\mathcal{I}_0$ are initially active, and all pair-programs are in one of their start states.*

2. *S is the set of all configurations such that (1) $S_0 \subseteq S$ and (2) if $s \in S$ and there is a normal or create transition from s to t, then $t \in S$.*

3. *$R_n \subseteq S \times \mathsf{Pids} \times S$ is a transition relation consisting of the normal transitions of $\mathcal{P}$, as given by Definition 16.*

4. *$R_c \subseteq S \times \mathsf{create} \times S$ is a transition relation consisting of the create transitions of $\mathcal{P}$, as given by Definition 14.*

It is clear that R_c and R_n are disjoint.

The creation of a pair-program is modeled in the above definition as a single transition. At a lower level of abstraction, this creation is realized by a protocol which synchronizes the "activation" of $(S^0_{ij}, P^j_i \,\|\, P^i_j)$ with the current computation of P_i and P_j, if they are already present. We give details in the full paper.

Let $M_{ij} = (S^0_{ij}, S_{ij}, R_{ij}, V_{ij})$ be the GSTD of $(S^0_{ij}, P^j_i \,\|\, P^i_j)$ as defined in Section 3. M_{ij} gives the semantics of $(S^0_{ij}, P^j_i \,\|\, P^i_j)$ *executing in isolation.*

7.5 The Creation Protocol

When a new pair-program $(S^0_{ij}, P^j_i \,\|\, P^i_j)$ is to be added, it must be synchronized with P_i and P_j, if these are already present, so that the (pair-consistency) requirement is not violated.

```
   CREATE($(S^0_{ij}, P^j_i \,\|\, P^i_j)$)
1. if $P_i$ is alive, then send $P_i$ a request to halt execution;
2. if $P_j$ is alive, then send $P_j$ a request to halt execution;
3. Wait for the necessary acknowledgments from $P_i$, $P_j$;
4. Select a reachable state $s_{ij}$ of $M_{ij}$ such that $s_{ij}{\upharpoonright}i = s_i$ if $P_i$ is alive, and $s_{ij}{\upharpoonright}j = s_j$ if $P_j$ is alive.
   (We require that the creation rule imposes sufficient constraints on pair-program creation so
   that this is guaranteed to hold).
5. Set the current state of $(S^0_{ij}, P^j_i \,\|\, P^i_j)$ to $s_{ij}$
6. Send $P_i$, $P_j$ permission to resume execution
```

8 Soundness of the Method for Dynamic Programs

Let π be a computation path of $\mathcal{P}$. Let $J \subseteq \mathsf{Pids} \times \mathsf{Pids}$ be such that $J \subseteq pairs(s.\mathcal{I})$ for all s along $\mathcal{P}$. Then, the *path-projection* of π onto J, denoted $\pi{\upharpoonright}J$, is obtained as follows. Start with the first configuration s along π such that $pairs(s) \cap J \neq \emptyset$. (If no such configuration exists, then $\pi{\upharpoonright}J$ is the empty sequence.) Replace every configuration t that occurs after s along π by $t{\upharpoonright}J$, and then remove all transitions $t \xrightarrow{i} t'$ along π such that P_i is not a process in some pair in J, coalescing the source and target states of all such transitions, which must be the same, since they do not refer to

P_i. Define M_J to be the $M_{\mathcal{P}}$ for the case when $\mathcal{I}_0 = J$, and no create transitions occur, i.e., the set of active pairs is always J.

Let $M_{ij} = (S^0_{ij}, S_{ij}, R_{ij})$ be the global state transition diagram of $(S^0_{ij}, P^j_i \parallel P^i_j)$, as given by Definition 3. S^0_{ij}, S_{ij} are the set of initial states, set of all states, respectively, of M_{ij}. $R_{ij} \subseteq S_{ij} \times \{i, j\} \times S_{ij}$ is the sets of transitions of M_{ij}. M_{ij} and $M_{\mathcal{P}}$ can be interpreted as ACTL structures. M_{ij} gives the semantics of $(S^0_{ij}, P^j_i \parallel P^i_j)$ *executing in isolation*, and $M_{\mathcal{P}}$ gives the semantics of $\mathcal{P}$. Our main soundness result below (the large model theorem) relates the ACTL formulae that hold in $M_{\mathcal{P}}$ to those that hold in M_{ij}. We characterize transitions in $M_{\mathcal{P}}$ as compositions of transitions in all the relevant M_{ij}:

Lemma 13 (Transition mapping) *For all configurations $s, t \in S$ and $i \in procs(s)$:*
$s \xrightarrow{i} t \in R_n$ iff

$$\forall j \in s.\mathcal{I}(i) \,:\, s{\restriction}ij \xrightarrow{i} t{\restriction}ij \in R_{ij} \text{ and}$$
$$\forall \{j,k\} \in pairs(s), i \notin \{j,k\} \,:\, s{\restriction}jk = t{\restriction}jk.$$

Proof. In configuration s, the constraints on a transition by P_i are given by exactly the pair-programs of which P_i is a member, i.e., those $(i, j) \in pairs(s)$. If all such pairs permit a transition ($\forall j \in s.\mathcal{I}(i) : s{\restriction}ij \xrightarrow{i} t{\restriction}ij \in R_{ij}$), and if all pair-programs in which P_i is not a member do not execute a transition ($\forall\{j,k\} \in pairs(s), i \notin \{j,k\} : s{\restriction}jk = t{\restriction}jk$), then P_i can indeed execute the transition $s \xrightarrow{i} t$, according to the semantics of $M_{\mathcal{P}}$. The other direction follows by similar reasoning. The technical formulation of this argument follows exactly the same lines as the proof of Lemma 6.4.1 in [AE98]. □

Corollary 14 (Transition mapping) *For all configurations $s, t \in S$, $J \subseteq pairs(s)$, and $i \in procs(J)$, if $s \xrightarrow{i} t \in R_n$, then $s{\restriction}J \xrightarrow{i} t{\restriction}J \in R_J$.*

Lemma 15 (Path mapping) *If π is a path in M, and let $J \subseteq$ Pids$\times$Pids be such that $J \subseteq pairs(s)$ for every configuration s along π. Then $\pi{\restriction}J$ is a path in M_J.*

Proof. The proof carries over from [AE98] with the straightforward modifications to deal with create transitions. □

In particular, when $J = \{(i, j)\}$, Lemma 15 forms the basis for our soundness proof, since it relates computations of the synthesized program $\mathcal{P}$ to computations of the pair-programs.

8.1 Deadlock-Freedom

In our dynamic model, the definition of wait-for-graph is essentially the same as the static case (Definition 5), except that the set of process nodes are also a function of the current configuration.

Definition 18 (Wait-for-graph $W(s)$) *Let s be an arbitrary configuration. The* wait-for-graph *$W(s)$ of s is a directed bipartite graph, where*

1. *the nodes of $W(s)$ are*
   (a) *the processes $\{P_i \mid i \in procs(s)\}$, and*
   (b) *the arcs $\{a_i \mid i \in procs(s)$ and $a_i \in P_i$ and $s{\restriction}i = a^I_i.start\}$*

2. *there is an edge from P_i to every node of the form a_i in $W(s)$, and*
3. *there is an edge from a_i to P_j in $W(s)$ if and only if $\{i,j\} \in pairs(s)$ and $a_i \in W(s)$ and $s{\restriction}ij(a_i.guard_j) = false$.*

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

Definition 19 (Supercycle) *SC is a* supercycle *in $W(s)$ if and only if:*

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

Note that this definition implies that SC is a subgraph of $W(s)$.

To extend the wait-for-graph condition (Section 5.1.1) to the dynamic model, we need to take the create transitions (R_c) into account. Thus, we modify the wait-for-graph condition as follows. In addition to the static Wait-For-Graph Condition of Definition 7, we require that a newly added pair-machine have at least one of its processes initially enabled.

Definition 20 (Dynamic wait-for-graph condition) *Let $k \in \mathsf{Pids}$, and let t_k be an arbitrary local state of $\hat{P}_k$, and let n be the number of outgoing arcs of t_k in $\hat{P}_k$. Let s, t be arbitrary configurations such that either*

1. *$(s,k,t) \in R_n$, $pairs(s) = pairs(t) = \{\{j,k\},\{k,\ell_1\},\ldots,\{k,\ell_n\}\}$, $k \notin \{j,\ell_1,\ldots,\ell_n\}$, and $t{\restriction}k = t_k$, or*
2. *$(s,\mathsf{create},t) \in R_c$, $pairs(s) = \{\{k,\ell_1\},\ldots,\{k,\ell_n\}\}$, $pairs(t) = \{\{j,k\},\{k,\ell_1\},\ldots,\{k,\ell_n\}\}$, $k \notin \{j,\ell_1,\ldots,\ell_n\}$, and $t{\restriction}k = t_k$.*

Then,

$$\forall a_j : (a_j \longrightarrow P_k \notin W(t)) \ \ or \ \ \exists a_k \in W(t) : (\forall \ell \in \{\ell_1,\ldots,\ell_n\} : a_k \longrightarrow P_\ell \notin W(t)).$$

Theorem 16 (Dynamic supercycle-free wait-for-graph) *If the wait-for-graph condition holds, and $W(s_0)$ is supercycle-free for every initial configuration $s_0 \in S_0$, then for every reachable configuration t of $M_{\mathcal{P}}$, $W(t)$ is supercycle-free.*

Proof. Similar to the proof of Theorem 6 with straightforward adaptations to deal with the create transitions (assumption 2 of Definition 20). □

8.1.1 Establishing Deadlock-freedom

We show that the absence of supercycles in the wait-for-graph of a configuration implies that there is at least one enabled move in that configuration. The proofs are very similar to the static case, and are omitted.

Proposition 17 (Supercycle [AE98]) *If $W(s)$ is supercycle-free, then some move a_i has no outgoing edges in $W(s)$.*

Theorem 18 (Deadlock freedom) *If, for every reachable configuration s of $M_{\mathcal{P}}$, $W(s)$ is supercycle-free, then $M_{\mathcal{P}}, S_0 \models \mathsf{AGEX}true$.*

8.2 Liveness

To assure liveness properties of the synthesized program $\mathcal{P}$, 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 a configuration s iff P_i has some arc that is enabled in s. Let *normal* be an assertion that is true along all transitions of $M_{\mathcal{P}}$ that are drawn from R_n. Let π be a fullpath of $M_{\mathcal{P}}$. Define $states(\pi) = \{s \mid s \text{ occurs along } \pi\}$. Define $procs(\pi) = \bigcup_{s \in states(\pi)} procs(s)$, and $pairs(\pi) = \bigcup_{s \in states(\pi)} pairs(s)$.

Definition 21 (Weak blocking fairness Φ_b) $\Phi_b(\pi) \overset{\text{df}}{=\!=} \bigwedge_{i \in procs(\pi)} \overset{\infty}{\mathsf{G}}(blk_i \wedge en_i) \Rightarrow \overset{\infty}{\mathsf{F}} ex_i$

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

Definition 22 (Weak eventuality fairness, Φ_ℓ)

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

Weak eventuality fairness requires that if an eventuality is continuously pending, and one of P_i or P_j is continuously enabled, then eventually one of them will be executed.

Definition 23 (Creation fairness Φ_c) $\Phi_c \overset{\text{df}}{=\!=} \overset{\infty}{\mathsf{F}} normal.$

A fullpath π satisfies creation fairness iff it contains an infinite number of normal transitions.

A fullpath π is *fair* iff $\pi \models_L \Phi_b(\pi) \wedge \Phi_\ell(\pi) \wedge \Phi_c$, where $\models_L$ is the satisfaction relation of propositional linear-time temporal logic [Eme90, MW84]. Our overall fairness notion Φ is thus the conjunction of weak blocking fairness, weak eventuality fairness, and creation fairness: $\Phi \overset{\text{df}}{=\!=} \Phi_b \wedge \Phi_\ell \wedge \Phi_c$.

Let $aen_j \overset{\text{df}}{=\!=} \forall a^i_j \in P^i_j : (\{\!\!\{ a^i_j.start \}\!\!\} \Rightarrow a^i_j.guard)$, i.e., aen_j holds iff every arc of P^i_j whose start state is a component of the current ij-state s_{ij} is also enabled in s. We say that P_k *blocks* P_i in configuration s iff, in $W(s)$, there is a path from P_i to P_k. Define $Wt_{ij}(s)$ to be the set of all k such that there is a path in $W(s)$ from at least one of P_i or P_j to P_k. Thus, $Wt_{ij}(s)$ is the set of processes that block the pair-program $(S^0_{ij}, P^j_i \,\|\, P^i_j)$ from executing some arc of P^j_i or P^i_j.

Definition 24 (Liveness condition for dynamic programs) *The liveness condition is the conjunction of the following:*

1. *Let s be an arbitrary reachable configuration. Then, for every $\{i,j\} \in pairs(s)$: $M_{ij}, S^0_{ij} \models \mathsf{AGA}(\mathsf{G}ex_i \Rightarrow \overset{\infty}{\mathsf{G}} aen_j)$*

2. *Let s be an arbitrary reachable configuration. For every $\{i,j\} \in pairs(s)$ such that $s \models pnd_{ij}$, the following must hold. There exists a finite $W \subseteq \mathsf{Pids}$ such that for all t reachable from s along paths in which pnd_{ij} holds in all configurations, $Wt_{ij}(t) \subseteq W$.*

The first condition above is a "local one," i.e., it is evaluated on pair-programs in isolation. It requires that, for every pair-program $(S^0_{ij}, P^j_i \parallel P^i_j)$, 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. The second condition is "global," it requires that a process is not forever delayed because new processes which block it are constantly being added.

Given the liveness condition and the absence of deadlocks and the use of Φ-fair scheduling, we can show that one of P_i or P_j is guaranteed to be executed from any configuration whose ij-projection has a pending eventuality. Let $\models_\Phi$ be the satisfaction relation of CTL* when the path quantifiers A and E are restricted to fair fullpaths (A: for all fair fullpaths, E: for some fair fullpath) [EL87].

Lemma 19 (Progress for dynamic programs) *Let s be an arbitrary reachable configuration and $\{i,j\} \in pairs(s)$. If*

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

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

Proof. By assumption 2 and Theorem 18, $M_{\mathcal{P}}, S_0 \models \mathsf{AGEX}\mathit{true}$. Hence every fullpath in $M_{\mathcal{P}}$ is infinite. Let π be an arbitrary Φ-fair fullpath starting in s. If $M_{\mathcal{P}}, \pi \models \mathsf{F}(ex_i \vee ex_j)$, then we are done. Hence we assume

$$\pi \models \mathsf{G}(\neg ex_i \wedge \neg ex_j) \tag{*}$$

in the remainder of the proof. Let t be an arbitrary configuration along π. By clause 2 of the liveness condition for dynamic programs (Definition 24), $Wt_{ij}(t) \subseteq W$ for some finite $W \subseteq \mathsf{Pids}$. Hence, these exists a configuration v along π such that, for all subsequent configurations w along π, $Wt_{ij}(w) \subseteq Wt_{ij}(v)$, i.e., after v, the set of processes that block $(S^0_{ij}, P^j_i \parallel P^i_j)$ does not increase. Now consider the static concurrent program P_J with interconnection relation $J = \{\{k,l\} \mid \{k,l\} \in pairs(v)$ and $\{k,l\} \subseteq Wt_{ij}(v)\}$ and initial state set $\{v{\upharpoonright}J\}$. By applying Lemma 10 to P_J, we conclude that $M_J, v{\upharpoonright}J \models_\Phi \mathsf{AF}(ex_i \vee ex_j)$. Now let $\rho_J = \pi^v{\upharpoonright}J$, where π^v is the infinite suffix of π starting in v. We now establish

$$\rho_J \text{ is an infinite path in } M_J \tag{**}$$

given the assumption that (*) holds. From (*) and weak eventuality fairness (Definition 22), we see that $Wt_{ij}(t)$ is nonempty for every configuration t along π, since otherwise one of P_i, P_j would be executed. By definition, there is no path in $W(t)$ from a process in $Wt_{ij}(t)$ to a process outside $Wt_{ij}(t)$. Hence, by assumption 2 and Proposition 17, there exists some $P_k \in Wt_{ij}(t)$ such that P_k has an enabled move in configuration t. Since this holds for all configurations t along π, we conclude by Weak blocking fairness (Definition 21), that infinitely often along π, some process in $Wt_{ij}(v)$ is executed. Hence, by Definition 1 and the definition of J, ρ_J is infinite.

From Lemma 15 ρ_J is a path in M_J. Hence, ρ_J is a fullpath in M_J. By Definition 1, the first state of ρ_J is $v{\restriction}J$. Hence, by $M_J, v{\restriction}J \models_\Phi \mathsf{AF}(ex_i \vee ex_j)$, we have $\rho_J \models \mathsf{F}(ex_i \vee ex_j)$. From $\rho_J = \pi^v{\restriction}J$ and Definition 1, we conclude $\pi^v \models \mathsf{F}(ex_i \vee ex_j)$. Hence, $\pi \models \mathsf{F}(ex_i \vee ex_j)$, contrary to assumption. □

8.3 The Large Model Theorem for Dynamic Programs

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

Theorem 20 (Large model) *Let $i, j \in$ Pids and let s be an arbitrary reachable configuration in $M_\mathcal{P}$ such that $\langle\{i,j\}, spec_{ij}\rangle \in s.\mathcal{I}$, where $spec_{ij}$ is an ACTL^-_{ij} formula. If*

1. *the liveness condition for dynamic programs holds,*
2. *$W(u)$ is supercycle-free for every reachable configuration u in $M_\mathcal{P}$, and*
3. *$M_{ij}, s{\restriction}ij \models f_{ij}$ for some $f_{ij} \in CL(spec_{ij})$,*

then

$$M_\mathcal{P}, s \models_\Phi f_{ij}.$$

Proof. The theorem follows from Theorem 18 and Lemma 19 in essentially the same way that Theorem 11 follows from Theorem 9 and Lemma 10, i.e., the static case. The proof is very similar, since the statements (but not the proofs) of Theorem 18 and Lemma 19 are identical to those of Theorem 9 and Lemma 10. The only difference in the proof is in dealing with create transitions. This is straightforward, since $(S^0_{ij}, P^j_i \,\|\, P^i_j)$ is created with its current state set to one of its reachable states, and so the same projection relationships hold between $M_\mathcal{P}$ and M_{ij} in the dynamic case as between M_I and M_{ij} in the static case, in particular, Lemma 15 provides the exact dynamic analogue for Lemma 3, and is the only projection result used in establishing the large model theorem. The only difference is that in the dynamic case the projection starts from the point that $(S^0_{ij}, P^j_i \,\|\, P^i_j)$ is created. Since we do not require computation paths to start from an initial state, this does not pose a problem. We note that the only result for static programs that involves reachability is Corollary 4, and this is only used to establish deadlock-freedom for the static case. For the dynamic case, deadlock freedom is guaranteed by the dynamic wait-for-graph condition (Definition 20), which contains an explicit clause (clause 2) to deal with creation. □

We note the important case of $f_{ij} = \mathsf{AG}g_{ij}$, i.e., f_{ij} expresses a *global* property, since g_{ij} holds in all configurations reachable from s.

9 Implementation in Atomic read/write Shared Memory

We now show how the synthesized program can be implemented in atomic read/write memory. To break down the atomicity of an arc in the synthesized program, we require that, in all pair-programs, all guards are *temporarily stable*, [Kat86], that is, once the guard holds, it continues to hold until some arc is executed, not necessarily the arc corresponding to the guard.

We generalize this discussion as follows. Let $(s_i, \oplus_{\ell \in [n]} B_\ell \to A_\ell, t_i)$ be an arc of P_i^j in pair-program $(S_{ij}^0, P_i^j \parallel P_j^i)$. We require

$$M_{ij}, S_{ij}^0 \models \bigwedge_{\ell \in [n]} \mathsf{AG}((\{\!|s_i|\!\} \land B_{i,\ell}^j) \Rightarrow \mathsf{A}[(\{\!|s_i|\!\} \land B_{i,\ell}^j) \, \mathsf{U_w} \, \neg s_i]). \qquad \text{(TSTAB)}$$

Now consider, in process P_i of $\mathcal{P}$, the arc in P_i s_i to t_i. By Definition 15, this arc has the label $\otimes_{j \in s.\mathcal{I}(i)} \oplus_{\ell \in [1:n_j]} B_{i,\ell}^j \to A_{i,\ell}^j$ in configuration s. Since P_i will be explicitly involved in any creation step which adds a pair of which P_i is a member, we assume that this label does not change, for the time being. Now, P_i can evaluate each of the $B_{i,\ell}^j$ sequentially, rather than simultaneously, since once true, each $B_{i,\ell}^j$ will remain true until P_i executes either the above arc or some other arc. Once P_i has observed that $j \in s.\mathcal{I}(i)$, there exists $\ell \in [1:n_j]$ such that $B_{i,\ell}^j$ holds, then P_i can execute the arc. The condition (TSTAB) can be checked in polynomial time by the model-checking algorithm of [CES86].

Execution of the arc will also involve the simultaneous execution of the assignments $A_{i,\ell}^j$. To break this multiple assignment down into atomic read and write operations, we use efficient solutions to the dining/drinking philosophers problem [SP88, CM88] to guarantee mutual exclusion of neighboring processes. Once a process has excluded all its neighbors (i.e., it "has all the forks"), it can then perform the multiple assignment sequentially. The following subsection gives details of this implementation.

As an alternative to using dining/drinking philosophers, if we have available hardware operations such as compare-and-swap. or load-linked/store conditional, then we can use the constructions of [Moi97, Moi00]. These algorithms permit the efficient, wait-free implementation of the multiple assignments.

9.1 Implementation using underlying dining/drinking philosophers algorithm

The problem is to implement every move $a_i = \otimes_{j \in s.\mathcal{I}(i)} \oplus_{\ell \in [1:n_j]} B_{i,\ell}^j \to A_{i,\ell}^j$ of every process P_i, in configuration s. The implementation consists of the following three procedures. The first, Poll(P_i, a_i) repeatedly polls all the guards of the move a_i, until a guard $B_{i,\ell}^j$ for each neighbor P_j of P_i is found which is true. When this occurs, the move a_i can be executed.

```
    Poll($P_i, a_i$)
1.  $X[a_i] := s.\mathcal{I}(i)$;
2.  **repeat**
        poll all the $B_{i,\ell}^j$ for $j \in X$, $\ell \in [1:n_j]$;
        **for** every $j$ such that $B_{i,\ell}^j$ polled *true* for some $\ell$
            $X := X - \{j\}$;
            $choice_i^j[a_i] := \ell$
    **until** $X[a_i] = \emptyset$
```

Now in a local state s_i, P_i will usually have a choice of several moves. The second procedure, Choose(P_i, s_i), repeatedly poll the guards of all such moves, until one is found all of whose guards are true. This move can then be executed by P_i. The actual execution is carried out by the

EXECUTE(P_i, s_i) procedure. EXECUTE(P_i, s_i) first invokes CHOOSE(P_i, s_i) to determine which move to execute. It then obtains the exclusive access to all the shared variables that execution of a_i updates, and exclusive access to the atomic propositions of P_i. Once all necessary locks are obtained, the move chosen move a_i can be executed in an "atomic" manner.

```
    CHOOSE$(P_i, s_i)$
1.  Let $a_i^1 \ldots a_i^k$ be all the moves of $P_i$ with start state $s_i$;
2.  Invoke POLL$(P_i, a_i^1) \ldots$ POLL$(P_i, a_i^k)$ simultaneously, and in an "interleaved" manner, i.e., interleave
    the executions of POLL$(P_i, a_i^1) \ldots$ POLL$(P_i, a_i^k)$;
3.  Let $a_i^c = (s_i, \otimes_{j \in s.\mathcal{I}(i)} \oplus_{\ell \in [1:n_j]} B_{i,\ell}^j \rightarrow A_{i,\ell}^j, t_i)$ be the first move for which $X[a_i^c] = \emptyset$ becomes true
4.  **return**$(a_i^c, choice_i^j)$
```

```
    EXECUTE$(P_i, s_i)$
1.  Invoke CHOOSE$(P_i, s_i)$ and let $a_i^c, choice_i^j$ be the returned values;
2.  **forall** $j \in s.\mathcal{I}(i)$ **do**
        obtain a lock on all variables in $A_{i,\ell}^j$, $\ell = choice_i^j[a_i^c]$, e.g., by using a drinking philosophers algorithm;
        obtain a lock on the atomic propositions of $P_i$ (i.e., those in $\mathcal{AP}_i$)
3.  **forall** $j \in s.\mathcal{I}(i)$ **do**
        execute $A_{i,\ell}^j$, $\ell = choice_i^j[a_i^c]$;
        change the local state of $P_i$ to $t_i$
4.  **forall** $j \in s.\mathcal{I}(i)$ **do**
5.      release all locks
```

The overall implementation is given by the procedure MAIN(P_i), which implements the process P_i. MAIN(P_i) repeatedly invokes EXECUTE(P_i, s_i), where s_i is the current local state of P_i. The low-level concurrent program P_r is then given by the concurrent composition of MAIN(P_i) for every process P_i that has been created so far. Let M_r be the global state transition diagram of P_r. M_r can be formally defined in a similar manner to $M_{\mathcal{P}}$ (Definition 17).

```
    MAIN$(P_i)$
1.  Let $s_i$ be an initial local state of $P_i$;
2.  **repeat** forever
        invoke EXECUTE$(P_i, s_i)$;
        update $s_i$ to be the resulting local state of $P_i$;
3.      participate in any outstanding CREATE protocols, if a request to suspend execution has
        been received
```

Note that P_i participates in executions of CREATE only when it is not executing normal transitions. This prevents the interleaving of the low atomicity implementations of normal and create transitions. Thus, in particular, during the low atomicity execution of a single normal transition, the value of $s.\mathcal{I}(i)$, i.e., the set of neighbors of P_i, does not change. This is essential to the correctness of the implementation.

9.2 Soundness of the Implementation in Atomic read/write shared memory

We show that M_r satisfies the same $\mathrm{ACTL}^{-}{}_{ij}$ formulae as $M_{\mathcal{P}}$. Roughly, we can consider M_r to consist of a "stretched out" version of $M_{\mathcal{P}}$, in which each transition of $M_{\mathcal{P}}$ is replaced by a sequence of transitions, together with all of the possible interleavings that result from this refinement of the transitions in $M_{\mathcal{P}}$. Due to our use of locking, this refinement does not generate any configurations that are unreachable in $M_{\mathcal{P}}$. Likewise, paths in M_r have "corresponding" paths in $M_{\mathcal{P}}$. Hence, so correctness is preserved.

Let s, u be configurations of $M_{\mathcal{P}}$, M_r respectively. Then define $s \sim u$ iff $\forall p \in \mathcal{AP} : s(p) = u(p)$ and $(\forall x \in \mathcal{SH} : s(x) = u(x))$. Let π, ρ be fullpaths of $M_{\mathcal{P}}$, M_r respectively. Then define $\pi \sim \rho$ iff π can be written as a sequence of finite bocks of configurations $\pi_1, \pi_2, \ldots$, ρ can be written as a sequence of finite bocks of configurations $\rho_1, \rho_2, \ldots$, and for all $i \geq 0$, for every s in π_i and every u in ρ_i, $s \sim u$.

Lemma 21 *Let s, u be configurations of $M_{\mathcal{P}}$, M_r respectively such that $s \sim u$. Then, for every fullpath ρ of M_r starting in u, there exists a fullpath π of $M_{\mathcal{P}}$ starting in s such that $\pi \sim \rho$.*

Proof. We assume that line 3 of $\textsc{Execute}(P_i, s_i)$ is executed atomically. This is reasonable, since exclusive access locks to all the shared variables and atomic propositions modified by line 3 of $\textsc{Execute}(P_i, s_i)$ are obtained first. We do not assume the atomic execution of any other part of the implementation algorithm.

Given ρ, consider the subsequence of the transitions of ρ given by the transitions that correspond to the execution of line 3 of $\textsc{Execute}(P_i, s_i)$. These are the only transitions of ρ which change the shared variables and atomic propositions, and so affect the truth of $\sim$. From the construction of the implementation algorithm, we can show that there exists a fullpath π of $M_{\mathcal{P}}$ starting in s which executes the same sequence of changes to the shared variables and atomic propositions. It follows that $\pi \sim \rho$. □

Theorem 22 *Let s, u be configurations of $M_{\mathcal{P}}$, M_r respectively such that $s \sim u$. Let π, ρ be fullpaths of $M_{\mathcal{P}}$, M_r respectively such that $\pi \sim \rho$. Let f be any formula of $\mathrm{ACTL}^* - X$. Then,*
If $M_{\mathcal{P}}, s \models_{\Phi} f$, then $M_r, u \models_{\Phi} f$.
If $M_{\mathcal{P}}, \pi \models_{\Phi} f$, then $M_r, \rho \models_{\Phi} f$.

Proof. The proof is by induction on the structure of f, i.e., by induction on the number of times rules S2, S3, and P1–3 of the definition of ACTL^* syntax are applied to generate f. Rule S1 of that definition gives the base case.

Base case: f is one of *true*, *false*, p, $\neg p$ for some atomic proposition p. Since s and u agree on all atomic propositions, $M_r, u \models f$ follows immediately from $M_{\mathcal{P}}, s \models f$.

Induction step: There are several cases.

Case 1: S2 is applied, and f is $g \vee h$, a state formula. Hence $M_{\mathcal{P}}, s \models g \vee h$. By ACTL^* semantics, $M_{\mathcal{P}}, s \models g$ or $M_{\mathcal{P}}, s \models h$. By the induction hypothesis, $M_r, u \models g$ or $M_r, u \models h$. Hence, by ACTL^* semantics, $M_r, u \models g \vee h$.

Case 2: S2 is applied, and f is $g \wedge h$, a state formula. Hence $M_{\mathcal{P}}, s \models g \wedge h$. By ACTL^* semantics, $M_{\mathcal{P}}, s \models g$ and $M_{\mathcal{P}}, s \models h$. By the induction hypothesis, $M_r, u \models g$ and $M_r, u \models h$. Hence, by ACTL^* semantics, $M_r, u \models g \wedge h$.

Case 3: S3 is applied, and f is $\mathsf{A}g$, a state formula. Hence g is a path formula. Assume $M_{\mathcal{P}}, s \models \mathsf{A}g$. Let ρ be an arbitrary fullpath of M_r starting in u. By Lemma 21, there exists a fullpath π of $M_{\mathcal{P}}$ starting in s such that $\pi \sim \rho$. Since $M_{\mathcal{P}}, s \models \mathsf{A}g$, we have $M_{\mathcal{P}}, \pi \models g$, by ACTL* semantics. From $\pi \sim \rho$ and the induction hypothesis, we obtain $M_r, \rho \models g$. Since ρ was chosen arbitrarily from the fullpaths starting in u, we conclude $M_r, u \models \mathsf{A}g$, by ACTL* semantics.

Case 4: P1 is applied, and f is g, where f is a path formula and g is a state formula.. Assume $M_{\mathcal{P}}, \pi \models f$. Hence, $M_{\mathcal{P}}, s \models g$, where s is the first state of π. Let u be the first state of ρ. Then, $s \sim u$, by the definition of $\pi \sim \rho$. By the induction hypothesis, $M_{\mathcal{P}}, s \models g$, and $s \sim u$, we obtain $M_r, u \models g$. Hence, by ACTL* semantics, $M, \rho \models f$.

Case 5: P2 is applied, and f is $g \vee h$, a path formula. Hence $M_{\mathcal{P}}, \pi \models g \vee h$. By ACTL* semantics, $M_{\mathcal{P}}, \pi \models g$ or $M_{\mathcal{P}}, \pi \models h$. By the induction hypothesis, $M_r, \rho \models g$ or $M_r, \rho \models h$. Hence, by ACTL* semantics, $M_r, \rho \models g \vee h$.

Case 6: P2 is applied, and f is $g \wedge h$, a path formula. Hence $M_{\mathcal{P}}, \pi \models g \wedge h$. By ACTL* semantics, $M_{\mathcal{P}}, \pi \models g$ and $M_{\mathcal{P}}, \pi \models h$. By the induction hypothesis, $M_r, \rho \models g$ and $M_r, \rho \models h$. Hence, by ACTL* semantics, $M_r, \rho \models g \wedge h$.

Case 7: P3 is applied, and f is $g\mathsf{U}h$, a path formula. Assume $M_{\mathcal{P}}, \pi \models f$. Hence, there exists $i \geq 1$ such that $M_{\mathcal{P}}, \pi^{i'} \models h$ and $(\forall i : 1 \leq i < i' : M_{\mathcal{P}}, \pi^i \models g)$. Let j' be the smallest natural number such that $\pi^{i'} \sim \rho^{j'}$. By the induction hypothesis, $M_r, \rho^{j'} \models h$. Let j be any natural number such that $1 \leq j < j'$. By the definition of $\pi \sim \rho$, there exists some i such that $1 \leq i < i'$ and $\pi^i \sim \rho^j$. Since $1 \leq i < i'$, we have $M_{\mathcal{P}}, \pi^i \models g$. Hence, by the induction hypothesis, $M_r, \rho^j \models g$. We have thus shown $M_r, \rho^{j'} \models h$ and $(\forall j : 1 \leq j < j' : M_r, \rho^j \models g)$. By ACTL* semantics, $M_r, \rho \models g\mathsf{U}h$.

Case 8: P3 is applied, and f is $g\mathsf{U}_{\mathsf{w}}h$, a path formula. Assume $M_{\mathcal{P}}, \pi \models f$. Hence, by ACTL* semantics, $M_{\mathcal{P}}, \pi \models g\mathsf{U}h$ or $M_{\mathcal{P}}, \pi \models \mathsf{G}g$. $M_{\mathcal{P}}, \pi \models g\mathsf{U}h$ is just Case 6 above. $M_{\mathcal{P}}, \pi \models \mathsf{G}gh$ can also be treated with an argument analogous to that of Case 6. Hence, we can establish $M_r, \rho \models g\mathsf{U}h$ or $M_r, \rho \models \mathsf{G}g$. Thus, $M_r, \rho \models g\mathsf{U}_{\mathsf{w}}h$. □

Theorem 23 (Large model theorem for low-atomicity implementation) *Let $i, j \in$ Pids and let u be an arbitrary reachable configuration in M_r such that $\langle\{i, j\}, spec_{ij}\rangle \in u.\mathcal{I}$, where $spec_{ij}$ is an* ACTL^-_{ij} *formula. If*

1. *the liveness condition for dynamic programs holds,*
2. *$W(v)$ is supercycle-free for every reachable configuration v in M_r, and*
3. *$M_{ij}, s{\upharpoonright}ij \models f_{ij}$ for some $f_{ij} \in CL(spec_{ij})$,*

then

$$M_r, u \models_{\Phi} f_{ij}.$$

Proof. Immediate from Theorem 20 and Theorem 22. □

10 Example—The Eventually Serializable Data Service

The eventually-serializable data service (ESDS) of [FGL+99, LLSG92] is a replicated, distributed data service that trades off immediate consistency for improved efficiency. A shared data object is replicated, and the response to an operation at a particular replica may be out of date, i.e., not reflecting the effects of other operations that have not yet been received by that replica. Thus, operations may be reordered *after* the response is issued. Replicas communicate amongst each other the operations they receive, so that eventually every operation "stabilizes," i.e., its ordering is fixed with respect to all other operations. Clients may require an operation to be *strict*, i.e., stable at the time of response (and so it cannot be reordered after the response is issued). Clients may also specify, in an operation x, a set $x.prev$ of other operations that should precede x (client-specified constraints, CSC). We let $\mathcal{O}$ be the (countable) set of all operations, $\mathcal{R}$ the set of all replicas (which may increase dynamically), $client(x)$ be the client issuing operation x, $replica(x)$ be the replica that handles operation x. We use x to index over operations, c to index over clients, and r, r' to index over replicas. For each operation x, we define a client process C_c^x and a replica process R_r^x, where $c = client(x)$, $r = replica(x)$. Thus, a client consists of many processes, one for each operation it issues. As the client issues operations, these processes are created dynamically. Likewise a replica consists of many processes, one for each operation it processes. Thus, we can use dynamic process creation and finite-state processes to model an infinite-state system, such as the one here, which in general handles an unbounded number of operations with time. The pair-specifications are as follows. The local structure specification of a process are implicitly conjoined with any pair-specification referring to that process. The atomic predicates have the following meaning for operation x. in is the initial state. wt means that x is submitted but not yet done. dn means that x is done. st means that x is table. snt means that the result of x has been sent to the client. We give pair-programs for a strict operation x. The pair-programs for a non-strict operation are similar, except that the transitions from dn_r^x to st_r^x to $[st_r^x\ snt_r^x]$ can also be performed in the reverse order (i.e., there is a branch from the dn_r^x state), since the result of x can be sent before x stabilizes. For example, Figure 11 gives the pair-program $R_r^x \,\|\, R_{r'}^x$ when x is not strict.

Local structure of clients C_c^x

in_c^x: x is initially pending

$\mathsf{AG}(in_c^x \Rightarrow (\mathsf{AX}_c wt_c^x \land \mathsf{EX}_c wt_c^x)) \land \mathsf{AG}(wt_c^x \Rightarrow \mathsf{AX}_c dn_c^x) \land \mathsf{AG}(dn_c^x \Rightarrow (\mathsf{AX}_c dn_c^x \land \mathsf{EX}_c dn_c^x))$: C_c^x moves from in_c^x to wt_c^x to dn_c^x, and thereafter remains in dn_c^x, and C_c^x can always move from in_c^x to wt_c^x.

$\mathsf{AG}((in_c^x \equiv \neg(wt_c^x \lor dn_c^x)) \land (wt_c^x \equiv \neg(in_c^x \lor dn_c^x)) \land (dn_c^x \equiv \neg(in_c^x \lor wt_c^x)))$: C_c^x is always in exactly one of the states in_c^x (initial state), wt_c^x (x has been submitted, and the client is waiting for a response), or dn_c^x (x is done).

Local structure of replicas R_r^x This is as shown in Figures 8, 9, and 10. We omit the temporal logic formulae to save space. They are constructed in an analogous manner to those for the clients

Client-replica interaction, $C_c^x \,\|\, R_r^x$, $x \in \mathcal{O}$, $c = client(x)$, $r = replica(x)$

$\mathsf{AG}(wt_r^x \Rightarrow wt_c^x)$: x is not received by its replica before it is submitted

$\mathsf{AG}(wt_c^x \Rightarrow \mathsf{AF}wt_r^x)$: every submitted x is eventually received by its replica

$\mathsf{AG}(wt_c^x \Rightarrow \mathsf{AF}dn_c^x)$: every submitted x is eventually performed

$\mathsf{AG}(dn_c^x \Rightarrow \mathsf{AG}dn_c^x)$: once an operation x is done, it remains done

CSC constraints, pair-machine $R_r^x \,\|\, R_{r'}^{x'}$, $x \in \mathcal{O}$, $x' \in x.prev$, $r = replica(x)$, $r' = replica(x')$

$\mathsf{AG}(dn_r^x \Rightarrow dn_{r'}^{x'})$: every operation in $x.prev$ is performed before x is

$\mathsf{AG}(dn_r^x \Rightarrow \mathsf{AG}dn_r^x) \land \mathsf{AG}(dn_{r'}^{x'} \Rightarrow \mathsf{AG}dn_{r'}^{x'})$: once an operation is done, it remains done

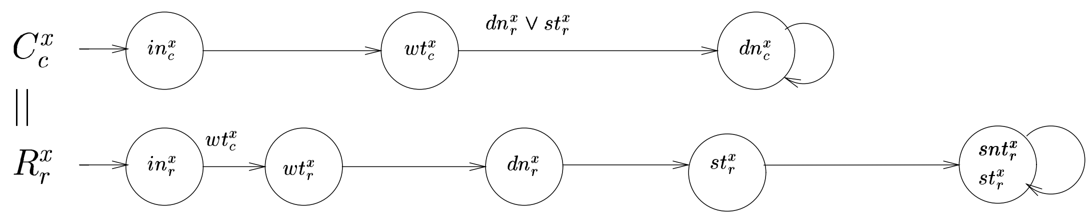

Figure 8: Client-replica interaction: pair-program $C_c^x \,\|\, R_r^x$, $r = replica(x)$.

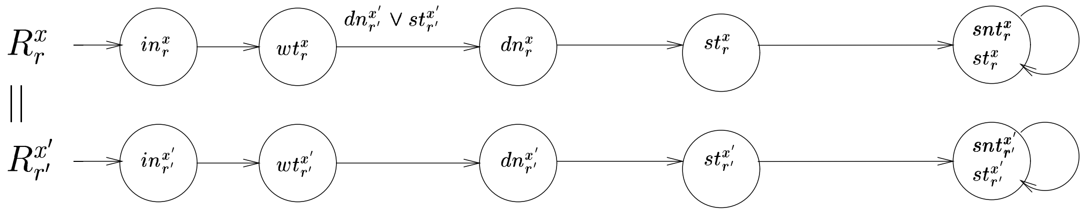

Figure 9: *CSC* constraints: pair-program $R_r^x \,\|\, R_{r'}^{x'}$, $r = replica(x)$, $x' \in x.prev$, $r' = replica(x')$.

<u>*Strictness constraints, pair-machine* $R_r^x \,\|\, R_{r'}^x$</u>, $x \in \mathcal{O}$, $x.strict$, $r = replica(x)$, $r' \in \mathcal{R} - \{replica(x)\}$

$\mathsf{AG}(snt_r^x \Rightarrow \bigwedge_i st_i^x)$: a strict operation is not performed until it is stable at all replicas

$\mathsf{AG}(snt_r^x \Rightarrow \mathsf{AG}snt_r^x) \wedge \mathsf{AG}(st_r^x \Rightarrow \mathsf{AG}st_r^x)$: once operation results are sent, they remain sent, and once an operation is stable, it remains stable

<u>*Eventual stabilization,* $R_r^x \,\|\, R_{r'}^x$</u>, $x \in \mathcal{O}$, $r = replica(x)$, $r' \in \mathcal{R} - \{replica(x)\}$

$\mathsf{AG}(wt_r^x \Rightarrow \bigwedge_i \mathsf{AF}st_i^x)$: every submitted operation eventually stabilizes

<u>*Rule for Dynamic process creation*</u> At any point, a client C_c can create the pair-programs required for the processing of a new operation x, for which $client(x) = C_c$. These pair-programs are $C_c^x \,\|\, R_r^x$ where $r = replica(x)$, $R_r^x \,\|\, R_{r'}^{x'}$ where $x' \in x.prev$, $r' = replica(x')$, and $R_r^x \,\|\, R_i^x$ $r = replica(x)$, $i \in \mathcal{R}$. It is permissible for $replica(x)$ to be a "new" replica, i.e., one that currently does not occur in any pair-program. Thus, the set of "current replicas" can be expanded at run-time. This is done implicitly when the first operation which is processed by that replica is instantiated. Likewise, a "new" client can submit an operation for the first time. Thus, clients can

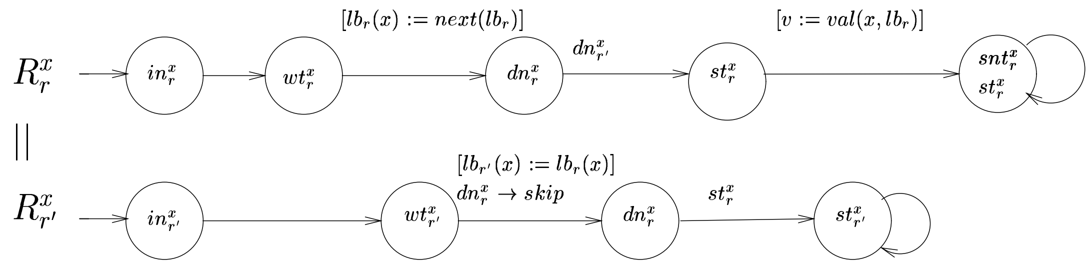

Figure 10: The Pair-program $R_r^x \,\|\, R_{r'}^x$, when x is strict, $r = replica(x)$, $r' \in \mathcal{R} - \{replica(x)\}$.

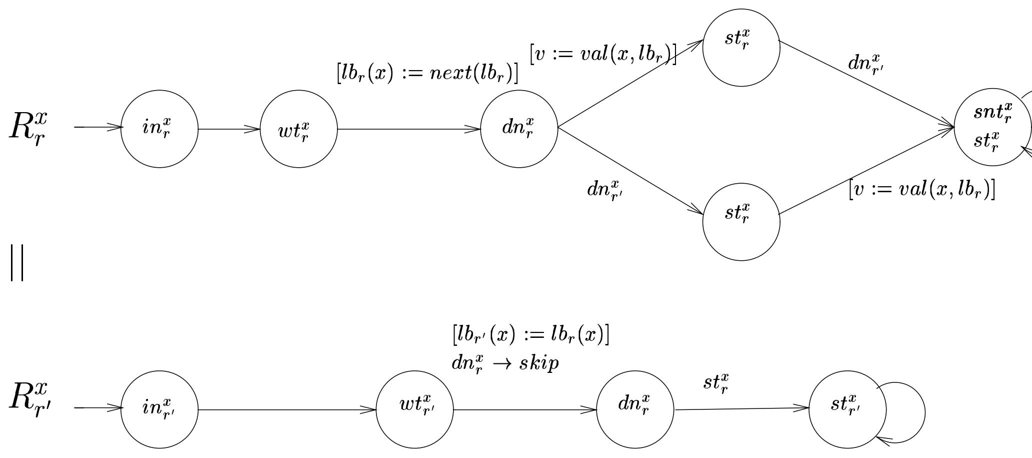

Figure 11: The Pair-program $R_r^x \,\|\, R_{r'}^x$, when x is not strict

also be created dynamically.

For each pair-specification, we synthesize a pair-program satisfying it, e.g., using the method of [EC82]. Figures 8, 9, and 10 show the resulting pair-programs. We then apply Definition 15 to synthesize the ESDS program with a dynamic number of clients and replicas, shown in Figure 12. The ESDS program, and the pair-program $R_r^x \,\|\, R_{r'}^x$ of Figure 10 both manipulate some "underlying" data, i.e., data which is updated, but not referenced in any guard, and so does not affect control-flow. This data consists of a labeling function lb_r which assigns to each operation x at replica r a label, drawn from a well-ordered set. The assignment $lb_r(x) := next(lb_r)$ takes the smallest label not yet allocated by lb_r and assigns it to $lb_r(x)$. The labels encode ordering information for the operations. The assignment $v := val(x, lb_r)$ computes a value v for operation x, using the ordering given by lb_r: operations with a smaller label are ordered before operations with a larger label. In the figures, these assignments to underlying data are shown within [..] brackets, alongside the arc-labels obtained by pairwise synthesis. They are not used when verifying correctness properties; the ordering constraints given by the $x.prev$ sets are sufficient to verify that the client-specified constraints are obeyed. Finally, we add self-loops to the final local state of every process for technical reasons related to establishing deadlock-freedom.

Correctness of the ESDS program follows immediately from Theorem 20, 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 [AE98], and the proof given there still applies).

11 Conclusions and Further Work

We presented a synthesis method which deals with an arbitrary and dynamically changing 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, preserves all pairwise correctness properties expressed as nexttime-free formulae of ACTL, and produces efficient low-grain atomicity programs which require only operations commonly available in hardware.

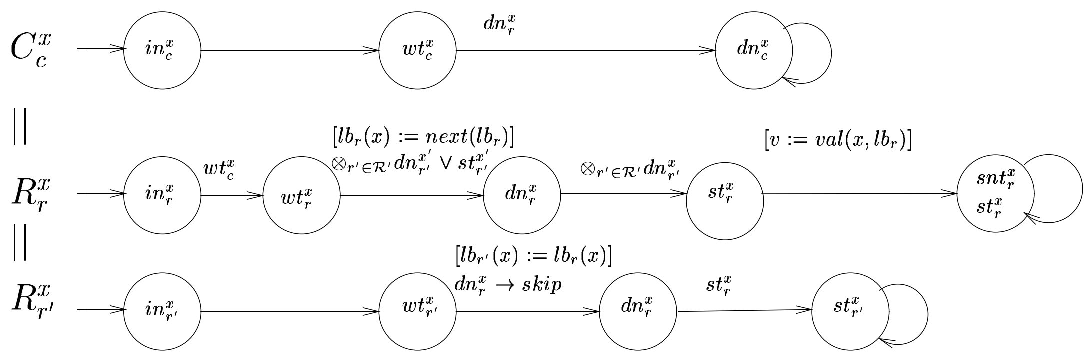

Figure 12: The Synthesized ESDS System. $c = client(x)$, $r = replica(x)$. x' ranges over $x.prev$, and r' ranges over $\mathcal{R}' = \mathcal{R} - \{replica(x)\}$ in $\otimes_{r'}$. $R^{x'}_{r'}$ is not shown since it is isomorphic to R^x_r.

Further work includes extending the method to a model of concurrent computation which facilitates abstraction and refinement, via a notion of external behavior, such as the model of [AL01], which also handles dynamic process creation. We also plan to deal with fault-tolerance by incorporating the work of [AAE98], and to investigate extending the method to other models of computation such as real-time and probabilistic.

References

[AAE98] A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. In *7th Annual ACM Symposium on the Principles of Distributed Computing*, pages 173 – 182, June 1998.

[AE98] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst.*, 20(1):51–115, Jan. 1998.

[AE01] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems for an atomic read/write model of computation. *ACM Trans. Program. Lang. Syst.*, 23(2):187–242, Mar. 2001. Extended abstract appears in ACM Symposium on Principles of Distributed Computing (PODC) 1996.

[AL01] P. C. Attie and N.A. Lynch. Dynamic input/output automata: a formal model for dynamic systems (extended abstract). In *CONCUR'01: 12th International Conference on Concurrency Theory*, LNCS. Springer-Verlag, Aug. 2001.

[AM94] A. Anuchitanukul and Z. Manna. Realizability and synthesis of reactive modules. In *Proceedings of the 6th International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 156–169, Berlin, 1994. Springer-Verlag.

[APR$^+$01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, 2001.

[Att99] P. C. Attie. Synthesis of large concurrent programs via pairwise composition. In *CONCUR'99: 10th International Conference on Concurrency Theory*, number 1664 in LNCS, Aalborg, Denmark, Aug. 1999. Springer-Verlag.

[BCG88] M.C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

[CES86] 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, Apr. 1986. Extended abstract in Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages.

[CGB86] E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 240 – 248, New York, 1986. ACM.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, Reading, Mass., 1988.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

[Dij82] E. W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*, pages 188–199. Springer-Verlag, New York, 1982.

[DWT90] D.L. Dill and H. Wong-Toi. Synthesizing processes and schedulers from temporal specifications. In *International Conference on Computer-Aided Verification*, number 531 in LNCS, pages 272–281. Springer-Verlag, 1990.

[EK00] E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *CADE*, pages 236–254, 2000.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, *Formal Models and Semantics.* The MIT Press/Elsevier, Cambridge, Mass., 1990.

[EC82] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2:241 – 266, 1982.

[EL87] E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8:275–306, 1987.

[EN96] E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *CAV*, pages 87–98, 1996.

[FGL$^+$99] A. Fekete, D. Gupta, V. Luchango, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220:113–156, 1999. Conference version appears in ACM Symposium on Principles of Distributed Computing, 1996.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, May 1994.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 583, 1969.

[Kat86] S. Katz. Temporary stability in parallel programs. Tech. Rep., Computer Science Dept., Technion, Haifa, Israel, 1986.

[KMTV00] O. Kupferman, P. Madhusudan, P.S. Thiagarajan, and M.Y. Vardi. Open systems in reactive environments: Control and synthesis. In *Proc. 11th Int. Conf. on Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 92–107, State College, Pennsylvania, 2000. Springer-Verlag.

[KV97] O. Kupferman and M.Y. Vardi. Synthesis with incomplete information. In *2nd International Conference on Temporal Logic*, pages 91–106, Manchester, July 1997. Kluwer Academic Publishers.

[LLSG92] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, Nov. 1992.

[Moi97] M. Moir. Transparent support for wait-free transactions. In *Workshop on Distributed Algorithms*, 1997.

[Moi00] M. Moir. Laziness pays! using lazy synchronization mechanisms to improve non-blocking constructions. In *Symposium on Principles of Distributed Computing*, 2000.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, Jan. 1984. Also appears in Proceedings of the Workshop on Logics of Programs, Yorktown-Heights, N.Y., Springer-Verlag Lecture Notes in Computer Science (1981).

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 179–190, New York, 1989. ACM.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of asynchronous reactive modules. In *Proceedings of the 16th ICALP*, volume 372 of *Lecture Notes in Computer Science*, pages 652–671, Berlin, 1989. Springer-Verlag.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, 2001.

[SG92] A. P. Sistla and S. M. German. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992. Conference version appears in IEEE Logic in Computer Science 1987.

[SP88] E. Styer and G. Peterson. Improved algorithms for distributed resource allocation. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, New York, Jan. 1988. ACM.