Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Synthesis of Large Dynamic Concurrent Programs from Dynamic Specifications

Paul C. Attie

Received: date / Accepted: date

Abstract We present two methods for synthesizing large concurrent programs from temporal logic specifications. The first method deals with finite-state concurrent programs that are static, i.e., the set of processes is fixed. It produces an infinite family of static finite-state concurrent programs. The second method deals with dynamic concurrent programs, i.e., new processes can be created and added at run-time. It produces a single dynamic concurrent program. A dynamic concurrent program may be viewed as a limiting case of an infinite family of static programs, and so the second method may be viewed as generalizing the first.

Our methods are algorithmically efficient, with complexity polynomial in the number of component processes (of the program) that are "alive" at any time. We do not explicitly construct the automata-theoretic product of all processes that are alive, thereby avoiding *state explosion*. Instead, for each interacting pair of processes, we construct (from a *pair-specification*) a *pair-structure* which embodies the interaction of the two processes. From each pair-structure, we synthesize a *pair-program* to coordinate the two processes. Our second method allows 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. We can thus 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. Thus, any state-explosion incurred is explosion in the product of only two processes. We establish "large model" theorems which show that the synthesized global program inherits correctness properties from the pair-programs.

Keywords concurrent program · dynamic process creation · formal specification · model checking · synthesis · temporal logic

Some results in this paper appeared in CONCUR 1999 as the paper "Synthesis of Large Concurrent Programs Via Pairwise Composition".

Paul C. Attie
Department of Computer Science, American University of Beirut, PO Box 11-0236, Beirut 1107 2020, Lebanon
Tel: +961 1 350000 extension 4221, Fax: +961 1 744 461 E-mail: pa07@aub.edu.lb

1 Introduction

We present two methods for mechanically synthesizing a concurrent program consisting of a large number of sequential finite-state processes executing in parallel. The first method produces an infinite family of concurrent programs, each of which is *static*, that is, the set of sequential processes comprising the program is fixed. The second method produces a *dynamic* concurrent program, that is a program in which new processes may be created and added at run-time We require that each process has a finite number of actions, and that the data referred to in action guards have finite domain. Underlying data that processes operate on, and which does not affect action guards, can be drawn from an infinite domain ("data independence", cf. Wolper [54]).

Our methods are computationally efficient; they do not explicitly construct the automata-theoretic product of a large number of processes (e.g., all processes that are "alive" at some point) and are therefore not susceptible to the *state-explosion problem*. Rather than build a global product, our methods construct 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-structure*, is a Kripke structure which embodies the interaction of the two processes. This pair-structure can be constructed manually, and then model-checked 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, by a *pair-specification*, and the pair-structure automatically synthesized from the pair-specification by the use of mechanical synthesis methods such as [6, 11, 30]. Both of these approaches are efficient, provided that there is no state-explosion in the product of two processes, i.e., that the pair-structures are small.

Corresponding to each pair-structure is a *pair-program*, a syntactic realization of the pair-structure, which generates the pair-structure as its global-state transition diagram. To synthesize a global program, 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 a pair-program to be added dynamically at run-time. It is then composed with the currently alive pair-programs to re-synthesize the program as it results after the addition. 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 "large model" theorems which show that all of the pair-properties also hold in the synthesized global program, even though the pair-programs are now no longer "executing in isolation".

When the pair-structures are small, our methods are 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 checking the 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 methods thus overcome the severe limitations previously imposed by state-explosion on the applicability of automatic synthesis methods, and extend these methods to the new domain of dynamic programs. In addition, we show in [8] that any static finite-state shared-memory concurrent program can be rewritten (up to strong bisimulation) in pairwise form, i.e., as a composition of pair programs. Thus, the restriction to pairwise form imposes no loss of expressiveness, in the static case. Investigation of expressiveness issues in the dynamic case is a topic of future work.

Our methods 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.

Our methods require the pair-programs to satisfy certain technical assumptions, and thus are not completely general. Nevertheless, they are applicable in many interesting cases. We illustrate our first 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. We use the second method to synthesize an eventually serializable data service [35, 42], which can deal with an unbounded number of operations. Each operation is handled by a set of processes that are dynamically created after the operation is submitted. Hence we synthesize a program that is usually considered to be *infinite-state*.

A crucial aspect of our methods is their soundness: which correctness properties can be established for the synthesized programs? Our large model theorems state 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 [38] minus the nexttime operator. In particular, invariants, event-ordering, and temporal leads-to properties can be expressed[1]. We also deal with correctness properties of the global program which are not directly expressible in pairwise fashion. We do so by using a deductive system for CTL [29] to show that such properties are a logical consequence of the pair-properties.

This paper extends previous work by Attie and Emerson [5] 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 methods, every process can be non-isomorphic with every other process.
2. It extends the set of pair-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 ACTL formulae, which can contain arbitrary nesting of temporal modalities.
3. It eliminates the requirement that the number of processes of the synthesized program be fixed: Attie and Emerson [5] synthesized an infinite family of programs, each of which contains a large, but fixed, number of processes. By contrast, our second method produces a single program, in which the number of processes can dynamically increase at run-time.
4. It introduces the use of temporal logic deduction and theorem proving to extend the range of properties that can be dealt with. In particular, any property that is logically implied by a conjunction of pair-properties can be established to hold for the synthesized global program. Hence we combine model-checking and theorem-proving.

[1] 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.

Related work. Many prior synthesis methods in the literature [2, 28, 30, 40, 41, 46–48] use some form of "global" construction (usually a tableau or an automaton) which embodies the entire concurrent program being synthesized. Such methods therefore suffer from full state explosion, and have complexity at least a single exponential (if not more) in the size of the specification. Some of these methods produce "open systems," or "reactive modules," [2,28, 40,41,47,48] 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 [29,30]. For example, an environment which can provide inputs $r_1,\dots,r_n$ can be modelled with the CTL formula $\mathsf{AG}(\mathsf{EX}r_1 \land \mathsf{EX}\neg r_1 \land \cdots \land \mathsf{EX}r_n \land \mathsf{EX}\neg r_n)$, which states that, in any reachable state, there is a successor state in which input r_1 is provided $(\mathsf{EX}r_1)$, and another successor state in which r_1 is not provided $(\mathsf{EX}\neg r_1)$, and likewise for $r_2,\dots,r_n$. We illustrate this point in our replicated data service example, where we specify that a client can submit operations at any time.

More recent work on synthesis includes various extensions such as methods for reusing libraries of components [13,44], or dealing with multivalued and noisy inputs [1]. All these methods have complexity polynomial in the global state space (and are thus subject to state-explosion) or significantly worse, in some cases. Deng et. al. [24] presents a method for synthesis based on using a global invariant as a specification. While effective for classical synchronization problems such as readers-writers and sleeping barber, this method sacrifices expressiveness, since it does not handle full temporal logic. Bonakdarpour et. al. [17] presents a symbolic (BDD-based) method for synthesizing fault-tolerant concurrent programs. However, this method requires a pre-exisiting fault-intolerant program, and adds fault-tolerance to it, and so it does not handle declarative temporal-logic specifications. Also, the efficiency of the method is, as usual, dependent on finding a good ordering for the variables in the BDD's.

Some recent synthesis methods deal with the high complexity of synthesis by using the notion of "bounded synthesis", and produce concurrent programs (as opposed to reactive modules) by including in the specification a description of the structure of the program to be synthesized, typically the number of processes and the communication mechanism that they use. Bounded synthesis was introduced by Schewe and Finkbeiner [36,50], which synthesizes concurrent programs to satisfy a specification that consists of a linear temporal logic (LTL) formula φ, an "architecture", and a family of bounds. The architecture specifies the set of system processes, and how these processes communicate with each other and with a distinguished environment process, namely via shared atomic propositions. The family of bounds provides a bound for the number of states of each system process, and a bound on the number of states of the resulting concurrent program, i.e., the product of all system processes and the environment process. Realizability of the specification is reduced (via a co-Buchi automaton for φ) to solving a set of constraints, using an SMT solver and a theory with order. If no solution is found for a given family of bounds, then the bounds can be increased and synthesis attempted again. Thus the solution found will be one with as few states as possible.

Faghih and Bonakdarpour [34] apply the framework of [36, 50] to synthesize self-stabilizing [27] concurrent programs. Their method takes as input a set LS of legitimate states, the temporal specification of self-stabilization, namely $\Diamond LS$ (convergence) and $LS \Rightarrow \bigcirc LS$ (closure), and a "topology", which plays the same role that "architecture" does in [36,50]. There is no bound on the set of states, since all possible states of the program must be considered. Again realizabiity of the specification is reduced to solving a set of constraints, which are produced using a co-Buchi automaton, and solved using an SMT solver.

Gascón and Tiwari [37] also apply the idea of bounded synthesis to the synthesis of self-stabilizing systems. They use a "template" written in the SAL [23] language to define the state-space of the synthesis problem, which includes both "regular variables" $\overrightarrow{x}$ (i.e., state variables of the concurrent program to be synthesized) and synthesis variables $\overrightarrow{y}$ (i.e., input variables which define the transition system under consideration). They deal with the properties "eventually φ" from any state, and "eventually always φ" from any state, where ϕ is a state predicate, and synthesize self-stabilizing concurrent programs that satisfy these properties. They bound the problem by replacing "eventually φ" by "φ within b steps" and generate a quantified boolean formula (QBF) which asserts the nonexistence of a solution that converges in b steps. If this formula is unsatisfiable, then a solution can be extracted. Any off-the-shelf QBF solver can be used.

Closely related to synthesis is the problem of *model repair*: given a structure M and a temporal logic formula φ, modify M so that it satisfies φ. The model repair problem was formulated by Buccafurri et. al. [19], for Kripke structures and the temporal logic CTL [29,30]. When M is intended to model the execution of a concurrent program with n processes, the size of M is exponential in n, and so model repair suffers from state-explosion. Chatzieleftheriou et. al. [20] presents an approach that attemps to avoid state-explosion, by repairing abstract structures, using Kripke modal transition systems and 3-valued semantics for CTL. They also aim to minimize the number of changes made to the concrete structure to effect a repair. They provide a set of basic repair operations: add/remove a may/must transition, change the propositional label of a state, and add/remove a state. Their repair algorithm is recursive CTL-syntax-directed. Attie et. al. [10] map an instance (M,φ) of "subtractive" model repair (repair is only by deleting transitions and states) to a boolean formula $repair(M,\varphi)$ such that (M,φ) has a solution iff $repair(M,\varphi)$ is satisfiable. Furthermore, a satisfying assignment determines which states and transitions must be removed from M to produce a model M' of φ. They also extend the method to repair of Kripke structures and concurrent programs in pairwise form (as given in this paper), in which case the size of $repair(M,\varphi)$ grows quadratically with n, the number of processes, rather than exponentially.

There are a number of methods proposed for *verifying* correctness properties of an infinite family of finite-state concurrent programs [4,22,31,33,49,52], where each program consists of a possibly large, but *fixed* set of processes (see [14] for a survey). 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 Attie and Emerson [5], 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 closer to the realm of practical distributed algorithms. We illustrate this point by using our method to synthesize an eventually-serializable replicated data service based on the algorithms of Fekete et. al. [35] and Ladin et. al. [42].

The rest of the paper is as follows. Section 2 presents technical preliminaries: model of concurrent computation, temporal logic, and fairness. Section 3 introduces the notions of pair-specification, pair-program, and pair-structure. Section 4 presents our first synthesis method, which produces static concurrent programs, and Section 5 establishes its soundness. Section 6 presents our second synthesis method, which produces dynamic concurrent programs, and Section 7 establishes its soundness. Section 8 discusses our results and proposes directions for further work, and Section 9 concludes. Appendix A gives a glossary of the main symbols used in the paper. We use as running examples throughout the paper a ring-based two-phase commit protocol (for the static case) and the eventually-serializable replicated data service mentioned above (for the dynamic case).

2 Technical preliminaries

2.1 Model of concurrent computation

We assume the existence of a possibly infinite, universal set Pids of unique process indices. With every process P_i, we associate a single, unique index, namely i. A static concurrent program $P = P_{i_1} \parallel \cdots \parallel P_{i_K}$ consists of a finite, fixed set of sequential processes $P_{i_1}, \ldots, P_{i_K}$ running in parallel, where $\{i_1, \ldots, i_K\} \subseteq \mathsf{Pids}$[2]. A dynamic 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., $P = P_{i_1} \parallel \cdots \parallel P_{i_K}$ where $P_{i_1}, \ldots, P_{i_K}$ 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, in which it remains forever, executing a "self-loop" transition.

To define the syntax and semantics of concurrent programs, we use the *synchronization skeleton* model of Clarke and Emerson [30]. The synchronization skeleton of a process P_i is a directed graph where each node 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. A node is an assignment of boolean values to an underlying set AP_i of *atomic propositions*, which are unique to P_i. We call a node of P_i an *i-state*. By changing its current i-state, P_i updates the values of the atomic propositions in AP_i. Other processes can read, but not write, the atomic propositions in AP_i. 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.

In addition to the atomic propositions of each process, there are shared variables $x_1, \ldots, x_m$ which are written and read by all processes.

Formally, the synchronization skeleton of each process P_i is a directed graph where each node s_i is a unique i-state of P_i, and each arc has a label of the form $\bigoplus_{\ell \in [1:n]} B_\ell \rightarrow A_\ell$,[3] where each $B_\ell \rightarrow A_\ell$ is a guarded command [25], and $\oplus$ is guarded command "disjunction," i.e., the arc is equivalent to n arcs, between the same pair of i-states, each labeled with one of the $B_\ell \rightarrow A_\ell$. This allows us to have at most one arc between any pair of i-states.

Roughly, the operational semantics of $\bigoplus_{\ell \in [1:n]} B_\ell \rightarrow A_\ell$ 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.[4] 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 $(s_{i_1}, \ldots, s_{i_K}, v_1, \ldots, v_m)$ where each s_i is the current local state of P_i, and $v_1, \ldots, v_m$ is a list giving the current values of all the shared variables, $x_1, \ldots, x_m$ (we assume these are ordered

[2] We use $i_1, \ldots, i_K$ instead of the more usual $1, \ldots, K$ since it is important for us to take subsets of process indices and use them to define a *sub-program*. The more general notation $i_1, \ldots, i_K$ emphasizes this.

[3] $[1:n]$ denotes the integers from 1 to n inclusive.

[4] This interpretation was proposed by Dijkstra [26].

in a fixed way, so that $v_1, \ldots, v_m$ specifies a unique value for each shared variable). A guard B_ℓ is a predicate on global states, and a body A_ℓ is a parallel assignment statement that updates the values of the shared variables. If B_ℓ is omitted from a guarded command, it is interpreted as *true*, and we write the command as A_ℓ. If A_ℓ is omitted, the shared variables are unaltered, and we write the command as B_ℓ.

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:

Definition 1 (Next-state relation) Let $s = (s_{i_1}, \ldots, s_i, \ldots, s_{i_K}, v_1, \ldots, v_m)$ be the current global state, and let P_i contain an arc from s_i to s'_i labeled by the command $\bigoplus_{\ell \in [1:n]} B_\ell \to A_\ell$. If B_ℓ (for some $\ell \in [1:n]$) is true in s, then a permissible next state is $s' = (s_{i_1}, \ldots, s'_i, \ldots, s_{i_K}, v'_1, \ldots, v'_m)$ where $v'_1, \ldots, v'_m$ is the list of updated values for the shared variables produced by executing A_ℓ in state s. The set of all (and only) such triples (s, i, s') constitutes the next-state relation of program P.

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 $P_{i_1} \parallel \ldots \parallel P_{i_K}$, where K is fixed. In this case, we also specify 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 $(S_0, P_{i_1} \parallel \ldots \parallel P_{i_K})$. 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. A *reachable path* is a path whose first state is reachable.

Finally, we define the *graph* of a process P_i to be the result of removing all arc labels:

Definition 2 (Graph of a process) Let $graph(P_i)$ denote the synchronization skeleton of P_i with all the arc labels removed.

2.2 Temporal logic

CTL* is a propositional branching time temporal logic [29] 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* [38] 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 [30] results from restricting CTL* so that every linear-time modality is paired with a path quantifier, and vice-versa. The sublogic ACTL [38] results from restricting ACTL* in the same way. The linear-time temporal logic PTL [46] 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:

(S1) The constants *true* and *false* are state formulae.
p is a state formula 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 [46] consists of the set of path formulae generated by rules (S1) and (P1)–(P3). The logic CTL forbids nesting and boolean combinations of linear time modalities, and is obtained by replacing rules (P1)–(P3) by

(P0) If f, g are state formulae, then $\mathsf{X}_j f$, $f \mathsf{U} g$ are path formulae.

We also introduce some additional modalities as abbreviations: $\mathsf{F} f$ (eventually) for $[true \mathsf{U} f]$, $\mathsf{G} f$ (always) for $\neg \mathsf{F} \neg f$, $[f \mathsf{W} g]$ (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 $f \mathsf{W} g$.

The logic ACTL [38] 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 $\mathsf{A}[f \mathsf{W} g]$.

We define the following sublogics of ACTL. ACTL$^-$ is ACTL without the AX_j modality, and ACTL$^-_{ij}$ is ACTL$^-$ where the atomic propositions are drawn only from $AP_i \cup AP_j$.

Formally, we define the semantics of CTL* formulae with respect to a Kripke structure $M = (S_0, S, R)$ consisting of

- S, a countable set of states. Each state is a mapping from a set AP of atomic propositions into $\{true, false\}$, and
- $S_0 \subseteq S$, a countable set of initial states, and
- $R = \bigcup_{i \in \varphi} R_i$, where $\varphi \subseteq \mathsf{Pids}$ and $R_i \subseteq S \times \{i\} \times S$ is a binary relation on S giving the transitions of process i.

Here $AP = \bigcup_{i \in \varphi} AP_i$, where AP_i is the set of atomic propositions that "belong" to process i. Other processes can read propositions in AP_i, but only process i can modify these propositions (which collectively define the local state of process i). Also, AP must contain all the atomic propositions that appear in the CTL* formula.

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, where a maximal path is one that is either infinite or ends in a state with no outgoing transitions.

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: $M, s \models f$ (respectively $M, \pi \models f$) means that f is true in structure M at state s (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) $M, s \models true$ and $M, s \not\models false$.
for $p \in AP$: $M, s \models p$ iff $s(p) = true$.
(S2) $M, s \models f \vee g$ iff $M, s \models f$ or $M, s \models g$
$M, s \models \neg f$ iff it is not the case that $M, s \models f$

(S3) $M,s \models \mathsf{A}f$ iff for every fullpath $\pi = (s_1, s_2, \ldots)$ in M with $s = s_1$: $M,\pi \models f$
(P1) $M,\pi \models f$ iff $M,s \models f$ where s is the first state along π
(P2) $M,\pi \models f \vee g$ iff $M,\pi \models f$ or $M,\pi \models g$
$M,\pi \models \neg f$ iff it is not the case that $M,\pi \models f$
(P3) $M,\pi \models \mathsf{X}_j f$ iff $\pi = (s_1, s_2, \ldots)$, π^2 is defined, $(s_1,s_2) \in R_j$ 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 $j \in [1:i-1]$: $M,\pi^j \models f$

When the structure M is understood from context, it may be omitted (e.g., $M,s \models p$ is written as $s \models p$). Since the other logics are all sublogics of CTL*, the above definition provides semantics for them as well. We refer the reader to Emerson [29] for details in general, and to Grumberg and Long [38] for details of ACTL.

2.3 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 Φ-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 Emerson and Lei [32] as follows:
(S3-fair) $M,s \models_\Phi \mathsf{A}f$ iff for every fullpath $\pi = (s_1, s_2, \ldots)$ in M such that $M,\pi \models \Phi$ and $s = s_1$: $M,\pi \models f$
Effectively, path quantification is only over the fullpaths that satisfy Φ, i.e., the Φ-fair fullpaths.

3 Pair-specifications, pair-programs, and pair-structures

Since our methods are based on composing pair-programs, we first discuss specifications for pair-programs, i.e., pair-specifications, then pair-programs themselves, and then pair-structures, which define the semantics of pair-programs. We use thoughout this section as a running example the problem of synthesizing a two-phase commit protocol in a ring.

3.1 Running example—two phase commit in a ring

We illustrate our method by synthesizing a ring-based (non fault tolerant) two-phase commit protocol $P = P_0 \,\|\, P_1 \,\|\, \cdots \,\|\, P_{n-1}$. P_0 is the *coordinator*, and $P_i, 1 \leq i < n$ are the participants of a transaction, where n is the size of the ring. The interacting pairs of processes are the neighboring pairs in the ring, i.e., P_{i-1} and $P_{i \bmod n}$, for $1 \leq i \leq n$. Each process P_i $(1 \leq i < n)$ has the following atomic propositions, which are mutually exclusive and exhaustive:

- st_i: the start state of P_i
- sb_i: P_i has submitted its part of the transaction
- cm_i: P_i has committed its part of the transaction
- ab_i: P_i has aborted its part of the transaction

The coordinator P_0 has atomic propositions sb_0, cm_0, and ab_0. It starts in sb_0. The protocol proceeds in two cycles around the ring. P_0 initiates the first cycle, in which each participant decides to either submit its part of the transaction or unilaterally abort. P_i can submit only after it observes that P_{i-1} has submitted. After the first cycle, the coordinator observes the state of P_{n-1}. If P_{n-1} has submitted, that means that all participants have submitted, 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 $P_j, i < j \le n-1$ 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_{n-1} and P_0, P_0 and P_1, and, for all $i = 2, \ldots, n-1$, P_{i-1} and P_i.

3.2 Pair-specifications

Definition 3 (Pair-specification) A *pair-specification for processes i and j* is a tuple $(\{i,j\}, spec_{ij})$, where $i, j \in \mathsf{Pids}$, $i \neq j$, and $spec_{ij}$ is a formula of ACTL^-_{ij}.

If $\mathsf{PS} = (\{i,j\}, spec_{ij})$ is a pair-specification for processes i and j, then we define $\mathsf{PS}.procs = \{i,j\}$, $\mathsf{PS}.for = spec_{ij}$. We intend the meta-syntactic use of the indices i and j to be symmetric, and so we make the convention that $spec_{ji}$ and $spec_{ij}$ denote the same formula. Note however, that the formula $spec_{ij}$ itself can treat i and j assymetrically, e.g., $\mathsf{AG}(T_i \wedge T_j \Rightarrow \mathsf{A}[T_i \mathsf{U} C_j])$.

A pair-specification $(\{i,j\}, spec_{ij})$ specifies the interaction of processes i and j. The temporal behavior property given by $spec_{ij}$ is called a *pair-property*. We assume in the sequel, without further statement, that $\mathsf{AG}\,\mathsf{EX}\,true$ (every state has some outgoing transition) is a conjunct of every pair-specification. This is because a necessary (but not sufficient) condition for deadlock-freedom of the synthesized program is that every pair-program be deadlock-free.

3.3 Pair-specifications for two-phase commit example

For the two-phase commit running example, the pair-specifications are given below, where we use $f \longrightarrow g$ to abbreviate $\mathsf{A}[(f \Rightarrow \mathsf{AF}g)\mathsf{W}g]$[5]. Intuitively, this means that if f holds at some state along a path π, then g holds at some (possibly different) state along π. There is no ordering on the times at which f and g hold, so that g could hold before f does. Hence we call this *temporal implication*.

Local structure of the coordinator P_0

sb_0: P_0 is initially in the submit state

$\mathsf{AG}(sb_0 \Rightarrow \mathsf{AX}_0(ab_0 \vee cm_0))$: P_0 moves from sb_0 to either ab_0 or cm_0

Local structure of other processes, P_i, $1 \le i < n$

st_i: P_i is initially in the start state

$\mathsf{AG}(st_i \Rightarrow (\mathsf{AX}_i(sb_i \vee ab_i) \wedge \mathsf{EX}_i ab_i))$: P_i moves from st_i to either sb_i or ab_i, and can always move from st_i to ab_i (unilateral abort)

$\mathsf{AG}(sb_i \Rightarrow \mathsf{AX}_i(ab_i \vee cm_i))$: P_i moves from sb_i to either ab_i or cm_i

[5] $f \Rightarrow \mathsf{AF}g$ is expressible in ACTL when f is purely propositional. This will always be the case when we use $\longrightarrow$.

The local structure specification for a pair-process is included in the pair-specification for any pair which the pair-processes is part of.

Pair-specification for P_{n-1} and P_0

Local structure specifications for P_{n-1} and P_0

$cm_0 \longrightarrow sb_{n-1}$: P_0 commits only if P_{n-1} submits

$\mathsf{AG}(\neg cm_0 \vee \neg ab_0) \wedge \mathsf{AG}(cm_0 \Rightarrow \mathsf{AG}cm_0) \wedge \mathsf{AG}(ab_0 \Rightarrow \mathsf{AG}ab_0)$: P_0 does not both commit and abort, commitment is stable, and abortion is stable

$\mathsf{AG}(\neg cm_{n-1} \vee \neg ab_{n-1}) \wedge \mathsf{AG}(cm_{n-1} \Rightarrow \mathsf{AG}cm_{n-1}) \wedge \mathsf{AG}(ab_{n-1} \Rightarrow \mathsf{AG}ab_{n-1})$: P_{n-1} does not both commit and abort, commitment is stable, and abortion is stable

Pair-specification for P_{i-1} and P_i, where $1 \le i < n$.

Local structure specifications for P_{i-1} and P_i

$sb_i \longrightarrow sb_{i-1}$: P_i submits only if P_{i-1} submits

$cm_i \longrightarrow cm_{i-1}$: P_i commits only if P_{i-1} commits

$(cm_{i-1} \wedge sb_i) \longrightarrow cm_i$: if P_{i-1} commits and P_i submits, then P_i commits

$\mathsf{AG}(\neg cm_{i-1} \vee \neg ab_{i-1}) \wedge \mathsf{AG}(cm_{i-1} \Rightarrow \mathsf{AG}cm_{i-1}) \wedge \mathsf{AG}(ab_{i-1} \Rightarrow \mathsf{AG}ab_{i-1})$: P_{i-1} does not both commit and abort, commitment is stable, and abortion is stable

$\mathsf{AG}(\neg cm_i \vee \neg ab_i) \wedge \mathsf{AG}(cm_i \Rightarrow \mathsf{AG}cm_i) \wedge \mathsf{AG}(ab_i \Rightarrow \mathsf{AG}ab_i)$: P_i does not both commit and abort, commitment is stable, and abortion is stable

$\mathsf{AG}(sb_i \Rightarrow \mathsf{A}[sb_i \mathsf{U}(sb_i \wedge (cm_{i-1} \vee ab_{i-1}))])$: upon submitting, P_i remains in the submit state until P_{i-1} decides

Note that the pair-specification for P_0 and P_1 is not quite isomorphic to those for P_{i-1} and P_i ($2 \le i < n$), since the local structure specification for P_0 is not isomorphic to the local structure specifications for P_i ($1 \le i < n$).

3.4 Pair-programs

Definition 4 (Pair-program $(S^0_{ij}, P^j_i \parallel P^i_j)$, pair-process, pair-state) A pair-program $(S^0_{ij}, P^j_i \parallel P^i_j)$ consists of a set S^0_{ij} of initial states and two processes P^j_i, P^i_j. P^j_i and P^i_j are *pair-processes*. The superscript j in P^j_i indicates that P^j_i implements the interaction of process i vis-a-vis process j. A node of P^j_i is an i-state, i.e., a mapping of AP_i to $\{true, false\}$. A node of P^i_j is a j-state, i.e., a mapping of AP_j to $\{true, false\}$.

Denote by SH_{ij} the set of shared variables of $(S^0_{ij}, P^j_i \parallel P^i_j)$. Since the indices i and j are used symmettrically, we make the convention that $SH_{ij} = SH_{ji}$, i.e., that SH_{ij} and SH_{ji} denote the same set of shared variables.

A state of $(S^0_{ij}, P^j_i \parallel P^i_j)$ is 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 SH_{ij}. We refer to states of $(S^0_{ij}, P^j_i \parallel P^i_j)$ as *ij-states*. When i and j are unspecified, we refer to an ij-state as a *pair-state*.

An ij-state inherits the assignments to atomic propositions that are 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 $s_{ij} = (s_i, s_j, v^1_{ij}, \ldots, v^m_{ij})$, and p_i, p_j are arbitrary atomic propositions in AP_i, AP_j, respectively. We require that different pair-programs have disjoint sets of shared variables, i.e., $\forall i, j, k, \ell : i \neq j \wedge k \neq \ell \wedge \{i, j\} \neq \{k, \ell\} \Rightarrow SH_{ij} \cap SH_{k\ell} = \emptyset$. In our synthesized programs, there will be many pair-processes $P^{j_1}_i, \ldots, P^{j_n}_i$, which implement the interaction of process i vis-a-vis processes $j_1, \ldots, j_n$, which are the neighbors of process i. We discuss this further below.

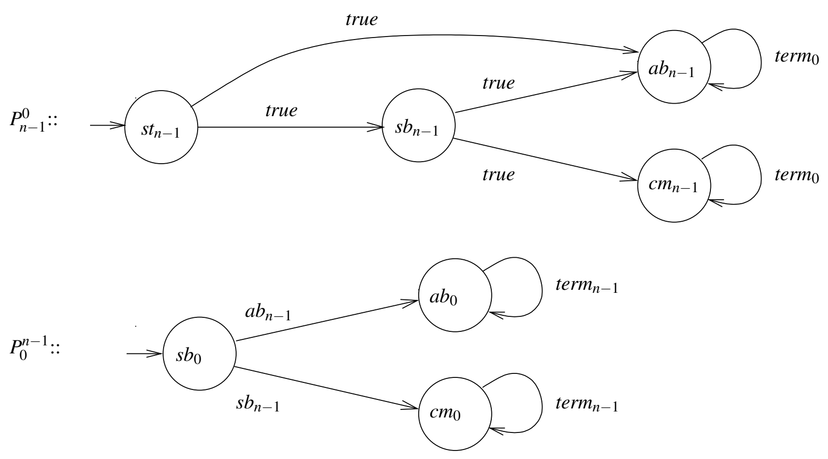

Fig. 1 Pair program $P^0_{n-1} \| P^{n-1}_0$.

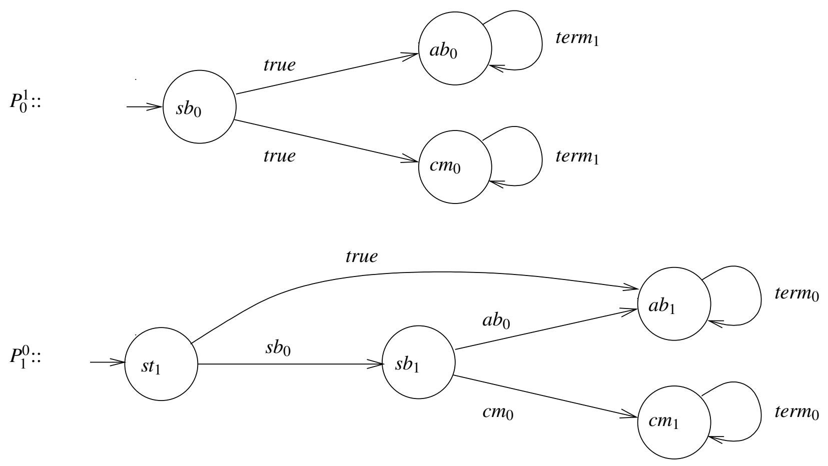

Fig. 2 Pair program $P^1_0 \| P^0_1$.

3.5 Pair-programs for two-phase commit example

For the two-phase commit running example, the pair-programs $P^0_{n-1} \| P^{n-1}_0$, $P^1_0 \| P^0_1$, and $P^i_{i-1} \| P^{i-1}_i$ $(2 \le i < n)$ are given in Figures 1, 2, and 3, respectively, where $term_i \equiv cm_i \vee ab_i$, and an incoming arrow with no source indicates an initial local state. The coordinator P_0 starts in the submit state sb_i since it does not choose whether to submit or not.

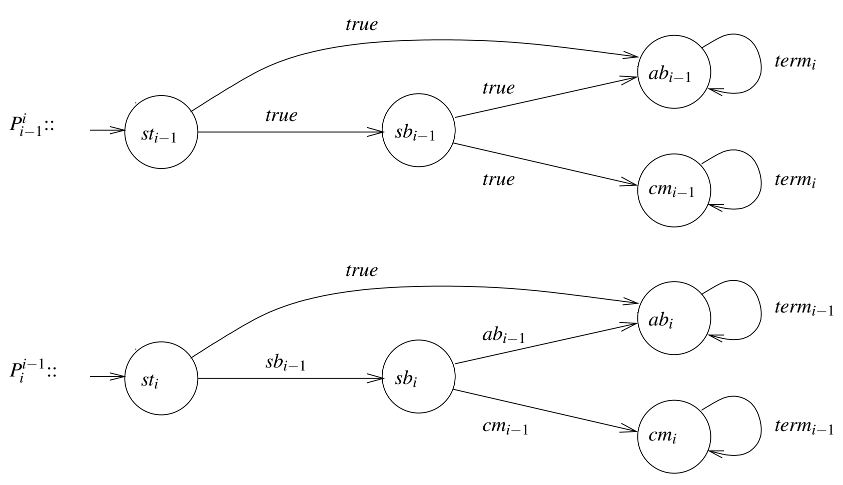

Fig. 3 Pair program $P^i_{i-1} \parallel P^{i-1}_i$.

3.6 Semantics of pair-programs: pair-structures

We define the *state projection operator* for pair-states. This operator has two variants, which are both denoted by the same symbol $\upharpoonright$. Context will always be sufficient to resolve the intended operator.

Definition 5 (State-projection for pair-programs) We define projection onto a single process from ij-states: 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 ij-state s_{ij}.

We also define projection onto the shared variables in SH_{ij} from ij-states: if $s_{ij} = (s_i, s_j, v^1_{ij}, \ldots, v^m_{ij})$, then $s_{ij}{\upharpoonright}SH_{ij} = (v^1_{ij}, \ldots, v^m_{ij})$. This gives the values that s_{ij} assigns to the shared variables in SH_{ij}.

The semantics of $(S^0_{ij}, P^j_i \parallel P^i_j)$ is given by the global state transition diagram M_{ij} generated by its execution. We call the global state transition diagram of a pair-program a *pair-structure*.

Definition 6 (Pair-structure) The semantics of a pair-program $(S^0_{ij}, P^j_i \parallel P^i_j)$ is given by the *pair-structure* $M_{ij} = (S^0_{ij}, S_{ij}, R_{ij})$ where

1. S_{ij} is the set of all ij-states of $(S^0_{ij}, P^j_i \parallel P^i_j)$,
2. $R_{ij} \subseteq S_{ij} \times \{i, j\} \times S_{ij}$ is a transition relation giving the transitions of $(S^0_{ij}, P^j_i \parallel P^i_j)$. Let $\bar{h} = i$ if $h = j$ and $\bar{h} = j$ if $h = i$. Then a transition (s_{ij}, h, t_{ij}) by $P^{\bar{h}}_h$ is in R_{ij} if and only if all of the following hold:
   (a) s_{ij} and t_{ij} are ij-states, and
   (b) there exists an arc in $P^{\bar{h}}_h$ from $s_{ij}{\upharpoonright}h$ to $t_{ij}{\upharpoonright}h$ with label $\bigoplus_{\ell \in [1:n]} B^{\bar{h}}_{h,\ell} \to A^{\bar{h}}_{h,\ell}$ such that there exists $m \in [1:n]$:
      (i) $s_{ij}(B^{\bar{h}}_{h,m}) = true$,
      (ii) $\langle s_{ij}{\upharpoonright}SH_{ij}\rangle\, A^{\bar{h}}_{h,m}\, \langle t_{ij}{\upharpoonright}SH_{ij}\rangle$, and

(iii) $s_{ij}\lceil\bar{h} = t_{ij}\lceil\bar{h}$.

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}). $\langle s_{ij}\lceil SH_{ij}\rangle\, A^{\bar{h}}_{h,m}\, \langle t_{ij}\lceil SH_{ij}\rangle$ is Hoare triple notation [39] for total correctness, which in this case means that execution of $A^{\bar{h}}_{h,m}$ always terminates,[6] and, when the shared variables in SH_{ij} have the values assigned by s_{ij}, leaves these variables with the values assigned by t_{ij}. $s_{ij}(B^{\bar{h}}_{h,m}) = \mathit{true}$ states that the value of guard $B^{\bar{h}}_{h,m}$ in state s_{ij} is *true*.[7]

3.7 Pair-structures for two-phase commit example

Figures 4, 5, and 6 give the respective global state transition diagrams (i.e., pair-structures) of the pair-programs $P^0_{n-1} \parallel P^{n-1}_0$, $P^1_0 \parallel P^0_1$, and $P^i_{i-1} \parallel P^{i-1}_i$ $(2 \le i < n)$, which were given in Figures 1, 2, and 3, respectively.

4 Synthesis of static concurrent programs

Our first synthesis method produces *static* concurrent programs, i.e., those with a fixed set of processes. 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 K 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 (in a "syntactic" manner) all the pair-programs to produce P.

To formalize the static synthesis problem, we introduce the notion of a specification for static programs.

Definition 7 (Global static specification) A *global static specification* $\mathscr{I}$ over process indices $\{i_1, \ldots, i_K\}$ is a finite set of *pair-specifications* such that

1. $\forall \mathsf{PS}, \mathsf{PS}' \in \mathscr{I} : \mathsf{PS}.\mathit{procs} \neq \mathsf{PS}'.\mathit{procs}$; i.e., every pair of processes has at most one pair-specification, and
2. $\forall i \in \{i_1, \ldots, i_K\}, \exists \mathsf{PS} \in \mathscr{I} : i \in \mathsf{PS}.\mathit{procs}$; i.e., every process is referenced by at least one pair-specification.

Also define $\mathscr{I}.\mathit{pairs} \overset{\mathrm{df}}{=\!=} \{\{i,j\} \mid \exists \mathsf{PS} \in \mathscr{I} : \{i,j\} = \mathsf{PS}.\mathit{procs}\}$.

$\mathscr{I}.\mathit{pairs}$ gives the pairs of processes P_i, P_j that must satisfy some temporal property spec_{ij} expressed solely in terms of $AP_i \cup AP_j$, i.e., in terms of the atomic propositions of the two processes. Our strategy for satisfying spec_{ij} is to have P_i and P_j *interact directly*, i.e., each process reads the other processes' atomic propositions (which, recall, encode the processes' local state), and they have a set SH_{ij} of shared variables that they both read and write.

[6] Termination is obvious, since $A^{\bar{h}}_{h,m}$ is a parallel assignment and the right-hand side of $A^{\bar{h}}_{h,m}$ is a list of constants.

[7] $s_{ij}(B)$ is defined by the usual inductive scheme: $s_{ij}(\text{“}x_{ij} = v_{ij}\text{”}) = \mathit{true}$ iff $s_{ij}(x_{ij}) = v_{ij}$, $s_{ij}(B1 \wedge B2) = \mathit{true}$ iff $s_{ij}(B1) = \mathit{true}$ and $s_{ij}(B2) = \mathit{true}$, $s_{ij}(\neg B) = \mathit{true}$ iff $s_{ij}(B) = \mathit{false}$.

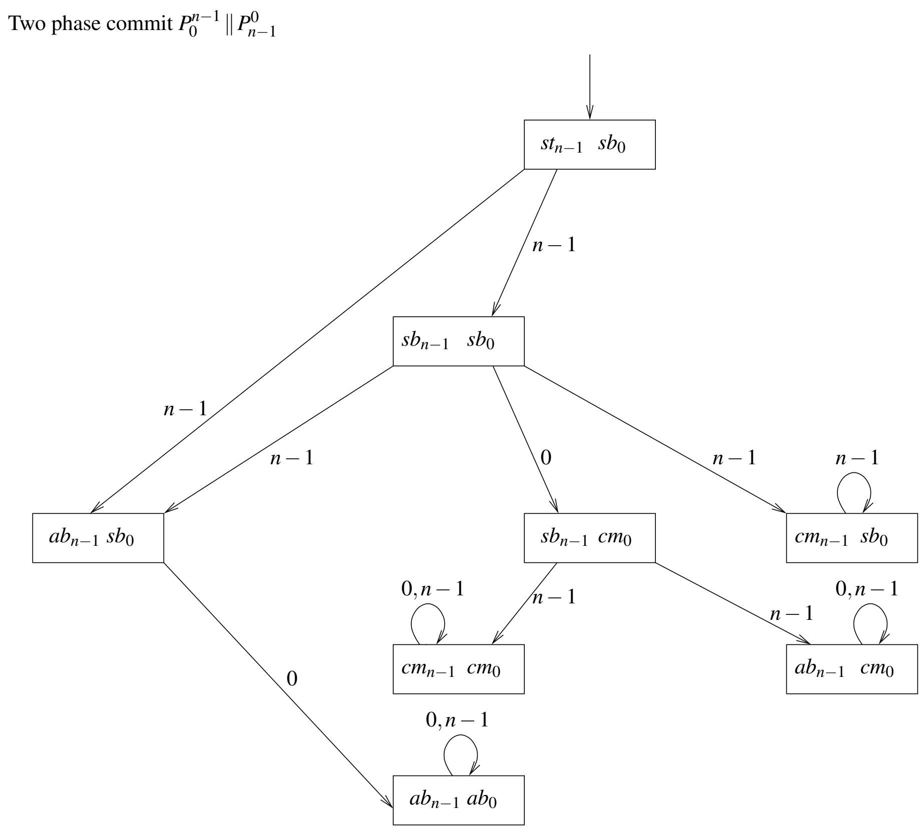

Fig. 4 Global state transition diagram of the pair-program $P_{n-1}^0 \,\|\, P_0^{n-1}$.

Definition 8 (Interconnection relation) The *interconnection relation* I corresponding to specification $\mathscr{I}$ is given by $I \overset{\text{df}}{=\!=} \{(i,j) \mid \{i,j\} \in \mathscr{I}.pairs\}$, that is $I = \{(i,j) \mid \exists \mathsf{PS} \in \mathscr{I} : \{i,j\} = \mathsf{PS}.procs\}$.

We use infix notation, so that $i I j$ is the same as $(i,j) \in I$. We define $dom(I) = \{i \mid \exists j : i I j\}$ to be the *domain* of I. Note that I is irreflexive and symmettric by construction, and that every process interacts directly with at least one other process: $\forall i \in dom(I), \exists j : i I j$. We say that i and j are *neighbors* when $(i,j) \in I$. We also introduce the following abbreviations: $I(i)$ denotes the set $\{j \mid i I j\}$; and $\hat{I}(i)$ denotes the set $\{i\} \cup \{j \mid i I j\}$.

Definition 9 (Static spatial modality) We introduce the *static 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 $\forall (i,j) \in I : spec_{ij}$.

Since our focus is on avoiding state-explosion, we do not explicitly address step 1 of the synthesis method outlined above. Any method for deriving concurrent programs (in synchronization-skeleton form) from temporal logic specifications can be used to generate the required pair-programs, for example the methods given in [6, 11, 30]. Since a pair-

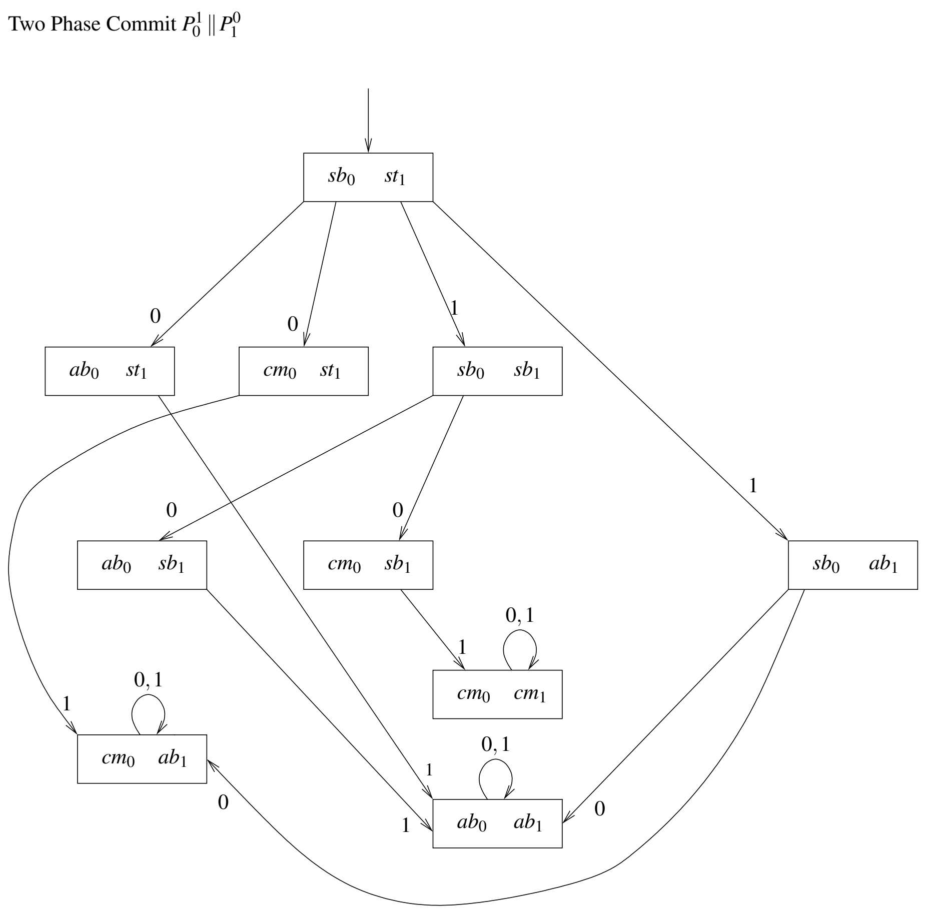

Fig. 5 Global state transition diagram of the pair-program $P_0^1 \| P_1^0$.

program has only $O(N^2)$ states (where N is the size of each sequential process), the problem of deriving a pair-program $(S_{ij}^0, P_i^j \| P_j^i)$ from a pair-specification is considerably easier than that of deriving a global program $(S_I^0, P_{i_1} \| \dots \| P_{i_K})$ from a global static specification, which is $O(N^K)$, if not worse. Hence, the contribution of this article, namely the second step above, is to reduce the more difficult problem (deriving the global program) to the easier problem (deriving the pair-programs). The following, taken from Attie and Emerson [5], gives some intuition for our approach:

> For sake of argument, first assume that all the pair-programs are isomorphic to each other. Let iIj. We take $(S_{ij}^0, P_i^j \| P_j^i)$ and generalize it in a natural way to a global program. We also show below that this 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., $I = \{(i, j), (j, k), (k, i)\}$. 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 synchroniza-

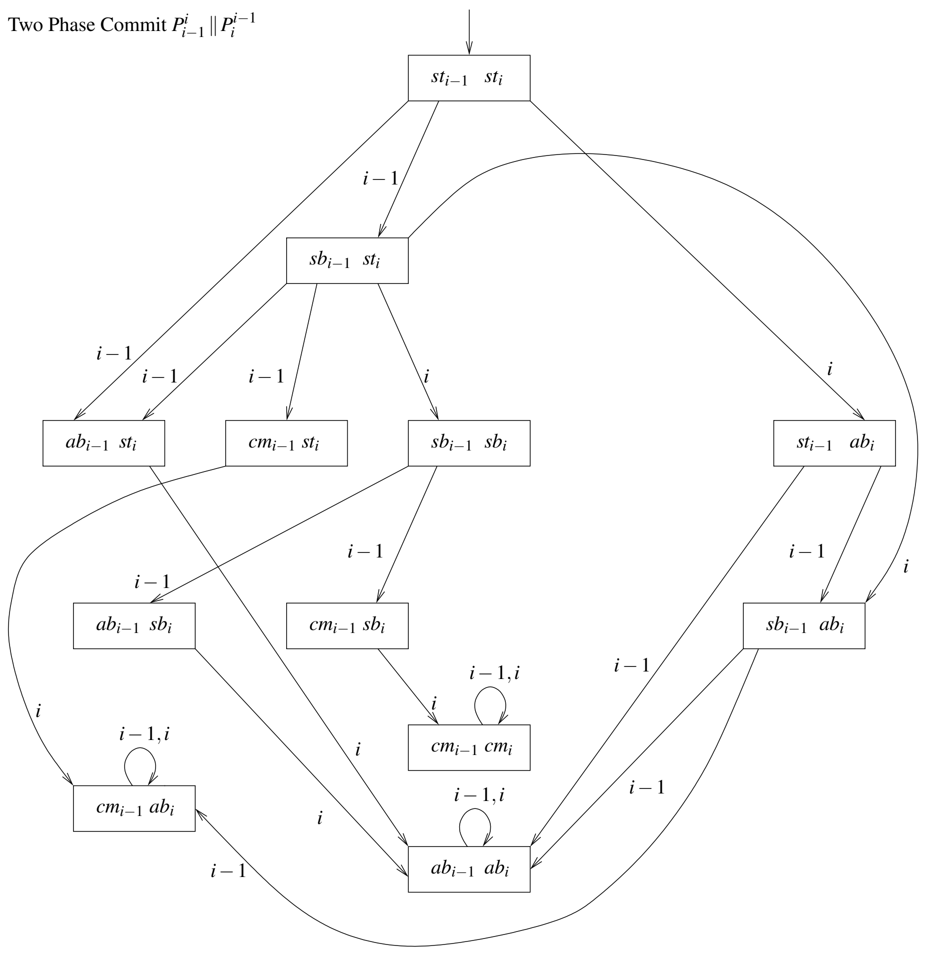

Fig. 6 Global state transition diagram of the pair-program $P^i_{i-1} \| P^{i-1}_i$.

tion 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 in the aforementioned three-process program (call it P^{jk}_i) has the same

graph 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 $(S_{ij}^0, P_i^j \| P_j^i)$, $(S_{ik}^0, P_i^k \| P_k^i)$, and $(S_{jk}^0, P_j^k \| P_k^j)$.
Generalizing the above to an arbitrary interconnection relation I, we see that the skeleton for process i in the global program (call it P_i) has the same graph 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 global program is exactly those states all of whose projections onto all the pairs in I give initial states of the corresponding pair-program.

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*, i.e., the same graph, on all common processes. That is, for pair-programs $(S_{ij}^0, P_i^j \| P_j^i)$ and $(S_{ik}^0, P_i^k \| P_k^i)$, we require that P_i^j and P_i^k have the same graph.

Definition 10 (Static process graph consistency) For pair-programs $(S_{ij}^0, P_i^j \| P_j^i)$ and $(S_{ik}^0, P_i^k \| P_k^i)$: $graph(P_i^j) = graph(P_i^k)$.

Since the setting of this paper is that of Attie and Emerson [5], except for the assumption of process similarity, we carry forward results from Attie and Emerson [5] which were established without using the process similarity assumption of that paper.

Before formally defining our synthesis method, we need some technical definitions. We define an I-state, which gives the form of the global state of the global programs that our method produces.

Definition 11 (I-state) Let $\mathscr{I}$ be a global static specification over $\{i_1, \ldots, i_K\}$, and let I be its interconnection relation. An I-state 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 values to the shared variables in $\bigcup_{(i,j)\in I} SH_{ij}$ (we assume some fixed ordering of these variables, so that the values assigned to them are uniquely determined by the list $v_1, \ldots, v_n$).

An I-state inherits the assignments to atomic propositions that are defined by its component i-states $(i \in \{i_1, \ldots, i_K\})$: $s(p_i) = s_i(p_i)$, where $s = (s_{i_1}, \ldots, s_i, \ldots, s_{i_K}, v_1, \ldots, v_n)$, and p_i is an arbitrary atomic proposition in AP_i.

If $J \subseteq I$, then we define a J-state exactly like an I-state, but using interconnection relation J instead of I.

We shall usually use s, t, u to denote I-states. We define the *state projection operator* $\upharpoonright$ for I-states. This operator has several variants, which are all denoted by the symbol $\upharpoonright$, also used for projection of pair-states. Context will always be sufficient to resolve the intended operator. We also use J as an arbitrary sub-relation of I $(J \subseteq I)$.

Definition 12 (State-projection for I-states) We define projection onto a single process from I-states: if $s = (s_{i_1}, \ldots, s_i, \ldots, s_{i_K}, v_1, \ldots, v_n)$, then $s\upharpoonright i = s_i$. This gives the i-state corresponding to the I-state s.

Define projection of an I-state onto a pair-program: if $s = (s_{i_1}, \ldots, s_{i_K}, v_1, \ldots, v_n)$, then $s\upharpoonright ij = (s_i, s_j, v_{ij}^1, \ldots, v_{ij}^m)$, where $s_i = s\upharpoonright i$, $s_j = s\upharpoonright j$, and $v_{ij}^1, \ldots, v_{ij}^m$ are those values from

$v_1, \ldots, v_n$ that denote values of variables in SH_{ij}. This gives the ij-state corresponding to the I-state s, and is well defined only when $i I j$.

Define projection onto the shared variables in SH_{ij} from I-states: if $s = (s_{i_1}, \ldots, s_{i_K}, v_1, \ldots, v_n)$, then $s \upharpoonright SH_{ij} = (v_{ij}^1, \ldots, v_{ij}^m)$, where $v_{ij}^1, \ldots, v_{ij}^m$ are those values from $v_1, \ldots, v_n$ that denote values of variables in SH_{ij}. This is well defined only when $i I j$.

Finally, define projection of an I-state onto a J-state. If $s = (s_{i_1}, \ldots, s_{i_K}, v_1, \ldots, v_n)$, then $s \upharpoonright J = (s_{j_1}, \ldots, s_{j_L}, v_J^1, \ldots, v_J^m)$, where $\{j_1, \ldots, j_L\}$ is the domain of J, and $v_J^1, \ldots, v_J^m$ are those values from $v_1, \ldots, v_n$ that denote values of variables in $\bigcup_{(i,j) \in J} SH_{ij}$. This gives the J-state corresponding to the I-state s and is well defined only when $J \subseteq I$.

For the synthesized program to actually exist, it requires at least one initial global state. Hence we require that the initial state sets of all the pair-programs must be such 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).

Definition 13 (Static initial state assumption) To ensure the existence of at least one initial state for the synthesized global static program, we assume that there exists an I-state s such that $\forall (i, j) \in I : s \upharpoonright ij \in S_{ij}^0$.

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

Definition 14 (Pairwise synthesis of global static programs) Let $\mathscr{I}$ be a global static specification over $\{i_1, \ldots, i_K\}$, and let I be its interconnection relation. For each pair-specification $(\{i, j\}, spec_{ij})$, let $(S_{ij}^0, P_i^j \| P_j^i)$ be a pair-program that satisfies $spec_{ij}$. Furthermore assume that this collection of pair-programs satisfies Definitions 10 and 13. Then, the global static program $P = (S_I^0, P_{i_1} \| \ldots \| P_{i_K})$ synthesized from $\mathscr{I}$ via the $(S_{ij}^0, P_i^j \| P_j^i)$ is as follows.

For all $i \in \{i_1, \ldots, i_K\}$, process P_i is derived from the pair-processes P_i^j, for all $j \in I(i)$ as follows:

P_i contains an arc from s_i to t_i with label $\bigotimes_{j \in I(i)} \bigoplus_{\ell \in [1:n_j]} B_{i,\ell}^j \to A_{i,\ell}^j$
iff
for every j in $I(i)$: P_i^j contains an arc from s_i to t_i with label $\bigoplus_{\ell \in [1:n_j]} B_{i,\ell}^j \to A_{i,\ell}^j$.

The *initial state set* S_I^0 of the synthesized global static program is derived from the initial state sets S_{ij}^0 of the pair-programs 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 $B_1 \to A_1 \oplus B_2 \to A_2$ is that if one of the guards B_1, B_2 evaluates to true, then the corresponding body A_1, A_2 respectively, can be executed. If neither B_1 nor B_2 evaluates to true, then the command "blocks," i.e., waits until one of B_1, B_2 evaluates to true. Since $\oplus$ is commutative and associative, we define and use the n-ary version $\bigoplus_{\ell \in [1:n]}$ of $\oplus$ in the usual manner. The operational semantics of $B_1 \to A_1 \otimes B_2 \to A_2$ is that if both of the guards B_1, B_2 evaluate to true, then the bodies A_1, A_2 can be executed in parallel. If at least one of B_1, B_2 evaluates to false, then the command "blocks," i.e., waits until both of B_1, B_2 evaluate to true. Since $\otimes$ is commutative and associative, we define and use the n-ary version $\bigotimes_{j \in I(i)}$ of $\otimes$ in the usual manner. Note also that, when using $\otimes$, we never have two assignments A_1, A_2 that update a common variable, and so the semantics of $\otimes$ is always

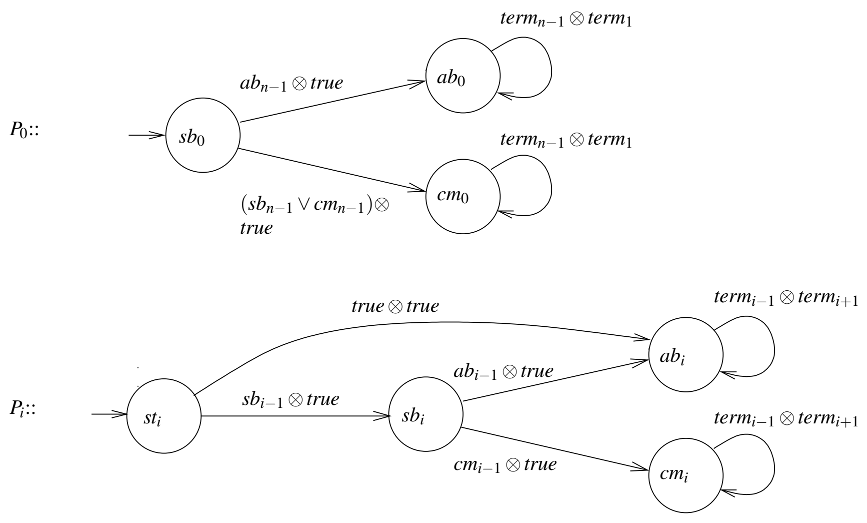

Fig. 7 The synthesized two phase commit protocol $P = P_0 \,\|\, (\|_{1 \le i < n} P_i)$.

well-defined. The operators $\oplus$ and $\otimes$ were defined in Attie and Emerson [5, Appendix D] as part of the "compact notation" for synchronization skeletons that was defined in that paper, and we refer the reader to that paper for a full discussion of $\oplus$ and $\otimes$. The compact notation enables the definition of our synthesis methods without incurring an exponential blowup in the size of the resulting synchronization skeletons, due to the "cartesian product" of the labels $\bigoplus_{\ell \in [1:n_j]}$ in the pair-programs that are being composed conjunctively.

If, in Definition 14, we replace I by some $J \subseteq I$, then we obtain the *J-subprogram* of $(S^0_I, P_{i_1} \| \ldots \| P_{i_K})$. In particular, for $J = \{(i,j),(j,i)\}$, the J-subprogram of $(S^0_I, P_{i_1} \| \ldots \| P_{i_K})$ is $(S^0_{ij}, P^j_i \| P^i_j)$, as expected.

Definition 14 is, in effect, a *syntactic transformation* that can be carried out in linear time and space (in both $(S^0_{ij}, P^j_i \| P^i_j)$ and I). In particular, we avoid explicitly constructing the global state transition diagram of $(S^0_I, P_{i_1} \| \ldots \| P_{i_K})$, which is of size exponential in K.

4.1 Synthesized two-phase commit protocol

The two phase commit protocol P, that is synthesized from the pair-programs $P^0_{n-1} \| P^{n-1}_0$, $P^1_0 \| P^0_1$, and $P^i_{i-1} \| P^{i-1}_i$ $(2 \le i < n)$ is given in Figure 7. We give P_0, and P_i for $i = 1, \ldots, n-1$.

4.2 Semantics of static global programs: I-structures

The semantics of $(S^0_I, P_{i_1} \| \ldots \| P_{i_K})$ is given by the global state transition diagram M_I generated by its execution, which we call an *I-structure*, since the form of the global state of $(S^0_I, P_{i_1} \| \ldots \| P_{i_K})$ is determined by the interconnection relation I.

Definition 15 (*I*-structure) The semantics of $(S^0_I, P_{i_1} \| \ldots \| P_{i_K})$ is given by the *I-structure* $M_I = (S^0_I, S_I, R_I)$ where

1. S_I is the set of all I-states,
2. $S_I^0 \subseteq S_I$ is given by Definition 14, and is the set of initial states of $(S_I^0, P_{i_1} \parallel \dots \parallel P_{i_K})$, and
3. $R_I \subseteq S_I \times \{i_1, \dots, i_K\} \times S_I$ is a transition relation giving the transitions of $(S_I^0, P_{i_1} \parallel \dots \parallel P_{i_K})$. A transition (s,i,t) by P_i is in R_I if and only if
   (a) $i \in \{i_1, \dots, i_K\}$,
   (b) s and t are I-states, and
   (c) there exists an arc in P_i from $s{\upharpoonright}i$ to $t{\upharpoonright}i$ with label $\bigotimes_{j \in I(i)} \bigoplus_{\ell \in [1:n_j]} B^j_{i,\ell} \to A^j_{i,\ell}$ such that all of the following hold:
      (i) $\forall j \in I(i), \exists m \in [1:n_j] : s{\upharpoonright}ij(B^j_{i,m}) = \mathit{true}$ and $\langle s{\upharpoonright}SH_{ij}\rangle A^j_{i,m} \langle t{\upharpoonright}SH_{ij}\rangle$,
      (ii) $\forall j \in \{i_1, \dots, i_K\} - \{i\}$: $s{\upharpoonright}j = t{\upharpoonright}j$,
      (iii) $\forall j,k \in \{i_1, \dots, i_K\} - \{i\}$, jIk: $s{\upharpoonright}SH_{jk} = t{\upharpoonright}SH_{jk}$.

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 $\{(i,j),(j,i)\}$ in Definition 15, then the result is, as expected, the same as that given by Definition 6, the definition of a pair-structure. That is, the two definitions are consistent. Furthermore, the semantics of a J-subprogram of $(S_I^0, P_{i_1} \parallel \dots \parallel P_{i_K})$, where $J \subseteq I$, 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 15.

5 Soundness of the method for synthesis of global static programs

We express correctness properties as formulae of a suitable sublogic of CTL*. We consider that a program satisfies a property if the formula expressing that property holds in all initial states of the global state transition diagram of the program.

Definition 16 (Satisfication of correctness property) Let P be a concurrent program, $M = (S^0, S, R)$ be the global state transition diagram of P, and f be a CTL* formula. Then P *satisfies* f iff $M, S^0 \models f$. Write $P \models f$ in this case.

Since M_{ij} and M_I are the global state transition diagrams of the pair-program, global program, respectively, our soundness results will relate the ACTL formulae that hold in M_I to those that hold in M_{ij}.

5.1 Projection onto subprograms

We define an I-path to be an alternating sequence of I-states and process indices.

Definition 17 (I-path) Let $\mathscr{I}$ be a static specification over $\{i_1, \dots, i_K\}$, and let I be its interconnection relation. An *I-path* is a finite or infinite sequence $s^1 \xrightarrow{d_1} \cdots s^n \xrightarrow{d_n} s^{n+1} \cdots$, where each s^n is an I-state, and each $d_n \in \{i_1, \dots, i_K\}$.

Let π be an arbitrary I-path. For any J such that $J \subseteq I$, define a *J-block* (cf. Browne et. al. [18] and Clarke et. al. [22]) 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 $i \in \mathit{dom}(J)$ (the domain of 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 $i \in \mathit{dom}(J)$, noting that a J-block can consist of a single state. It 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 $i \in dom(J)$ cannot affect any atomic proposition in $\bigcup_{i \in dom(J)} AP_i$, nor can it change the value of a variable in $\bigcup_{(i,j) \in J} SH_{ij}$; 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 n'th J-block of π.

Definition 18 (Path projection in static programs) Let π be an arbitrary I-path. Write π as $B^1 \xrightarrow{d_1} \cdots B^n \xrightarrow{d_n} B^{n+1} \cdots$ where B^n is a J-block for all $n \geq 1$. 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 n'th J-block of π corresponding to the n'th state of $\pi{\upharpoonright}J$. Note that path projection is well defined when π is finite.

The 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 [5]) *Let $dom(I) = \{i_1, \ldots, i_K\}$. For all I-states $s,t \in S_I$ and $i \in \{i_1, \ldots, i_K\}$:*

$$s \xrightarrow{i} t \in R_I \textit{ iff } ((\forall j \in I(i) : s{\upharpoonright}ij \xrightarrow{i} t{\upharpoonright}ij \in R_{ij}) \textit{ and } (\forall (j,k) \in I, i \notin \{j,k\} : s{\upharpoonright}jk = t{\upharpoonright}jk)).$$

For $J \subseteq I$, we apply Lemma 1 to every pair in J, to obtain:

Corollary 1 (Transition mapping [5]) *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 Corollary 1 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 [5].

Lemma 2 (Path mapping [5]) *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 2 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 2 (State mapping [5]) *Let $J \subseteq I$. If t is a reachable state in M_I, then $t{\upharpoonright}J$ is a reachable state in M_J.*

The proofs of the above four results can be found in Attie and Emerson [5]. The proofs all carry over since they did not assume that the pair-programs $(S^0_{ij}, P^j_i \parallel P^i_j)$ are isomorphic to each other. The statement of Lemma 1 is simpler, but logically equivalent to, that in [5].

5.2 Deadlock-freedom of global static programs

As Attie and Emerson [5] show, it is possible for the synthesized global static program P to be deadlock-prone even though all the pair-programs are deadlock-free. To ensure deadlock-freedom of P, they imposed a condition on the "blocking behavior" of processes: after a process executes an arc, it must either have another arc 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 blocking behavior using the notion of a *wait-for graph*. The wait-for graph in a particular I-state s contains as nodes all the processes, and all the arcs whose start state is a component of s. These arcs have an outgoing edge to every process which blocks them. If the arc a_i of P_i has label $\bigotimes_{j \in I(i)} \bigoplus_{\ell \in [1:n_j]} B^j_{i,\ell} \to A^j_{i,\ell}$, then define $a_i.guard = \bigwedge_{j \in I(i)} \bigvee_{\ell \in [1:n_j]} B^j_{i,\ell}$ and $a_i.guard_j = \bigvee_{\ell \in [1:n_j]} B^j_{i,\ell}$. $a_i.guard$ is the guard of arc a_i. $a_i.guard_j$ is the conjunct of the guard of arc a_i which is evaluated over the (pairwise) shared state with P_j. Also, let $a_i.start$ denote the start state of arc a_i.

Definition 19 (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 processes $\{P_i \mid i \in dom(I)\}$, and
   (b) the arcs $\{a_i \mid i \in dom(I)$ and $a_i \in P_i$ and $s{\upharpoonright}i = a_i.start\}$, i.e., the arcs that P_i is "ready" to execute
2. there is an edge from P_i to every node a_i such that $a_i \in P_i$ and $s{\upharpoonright}i = a_i.start$, and
3. there is an edge from a_i to P_j if and only if iIj and $s{\upharpoonright}ij(a_i.guard_j) = false$.

We abuse notation here and write $a_i \in P_i$ to mean that a_i is an arc of P_i, and $a_i \in W_I(s)$ to mean that a_i is a node of $W_I(s)$. Also, $a_i \longrightarrow P_j \in W_I(s)$ means that $a_i \longrightarrow P_j$ is an edge of $W_I(s)$, etc. An edge from P_i to a_i means that P_i is ready to exectute a_i, and is waiting for its guard to hold. An edge from a_i to P_j means that the conjunct of the guard of a_i that depends on P_j is false, so that a_i is "blocked" by P_j. A path in $W_I(s)$ is called a *wait-for path*.

We characterize a deadlock as the occurrence in the wait-for graph of a graph-theoretic construct that we call a *supercycle*:

Definition 20 (Supercycle) SC is a supercycle in $W_I(s)$ if and only if all of the following hold:

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

If $W_I(s)$ does not contain a supercycle, we say that s is supercycle-free. We say that a global static program is supercycle-free iff all of its reachable states are supercycle-free.

In Attie and Emerson [5], Attie and Chockler [12], and Attie et. al. [9], several sufficient but not necessary criteria for supercycle-freedom are given. These can usually be evaluated over the product of a small number of processes, thereby avoiding state-explosion. The model of computation in Attie et. al. [9] is somewhat different, being based on multiparty interactions, but the results can be seen to carry over, since the same notion of wait-for graph and supercycle captures deadlock behavior in both cases. Note also that Attie and Emerson [5] give a transformation from the global static programs that their method produces, to concurrent programs which use multiparty interactions as the only communication and synchronization primitive.

A *local deadlock* exists in a global state s of a program when there is a subset of processes that are all disabled in every state reachable from s, even though other processes can execute. Attie et. al. [9] show that state s has a local deadlock iff $W_I(s)$ contains a supercycle. An immediate consequence of this is that, in a supercycle-free state, there is at least one process with an enabled arc.

Proposition 1 (Deadlock freedom [9]) *If $W_I(s)$ is supercycle-free, then in I-state s, some process P_i of $(S^0_I, P_{i_1} \parallel \ldots \parallel P_{i_K})$ has an enabled arc.*

Proof Suppose not. Then the set of all processes $\{P_i \mid i \in dom(I)\}$ together with the arcs $\{a_i \mid i \in dom(I)$ and $a_i \in P_i$ and $s{\upharpoonright}i = a_i.start\}$ constitute a supercycle in $W_I(s)$. $\square$

5.3 Liveness of global static programs

To assure liveness properties of the synthesized global static programs, we apply a form of weak fairness to their scheduling. We define this fairness below as a formula of the linear time temporal logic PTL [46]. 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 P_i. Let en_i hold in a configuration s iff P_i has some arc that is enabled in s. Let a^j_i be an arc in the pair-process P^j_i, from i-state s_i to i-state t_i, and with the label $\bigoplus_{\ell \in [1:n]} B^j_{i,\ell} \to A^j_{i,\ell}$. Define $a^j_i.guard = \bigvee_{\ell \in [1:n]} B^j_{i,\ell}$, the guard of a^j_i. Also define $a^j_i.start$ to be s_i, the start state of a^j_i. Recall that $M_{ij} = (S^0_{ij}, S_{ij}, R_{ij})$ is the global state transition diagram of $(S^0_{ij}, P^j_i \parallel P^i_j)$.

Definition 21 (State-to-formula operator, $\{\!|s_i|\!\}$, $\{\!|s_{ij}|\!\}$) Let s_i be an i-state. The state-to-formula operator $\{\!|s_i|\!\}$ 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 $s'_i \neq s_i$:

$$\{\!|s_i|\!\} \overset{\text{df}}{=\!=} (\bigwedge_{s_i(p_i)=true} p_i) \wedge (\bigwedge_{s_i(p_i)=false} \neg p_i)$$

where p_i ranges over the members of AP_i. Likewise, for ij-state $s_{ij} = (s_i, s_j, v^1_{ij}, \ldots, v^m_{ij})$, we define

$$\{\!|s_{ij}|\!\} \overset{\text{df}}{=\!=} \{\!|s_i|\!\} \wedge \{\!|s_j|\!\} \wedge (\bigwedge_{\ell \in [1:m]} x^\ell_{ij} = v^\ell_{ij}).$$

Thus, $\{\!|s_{ij}|\!\}$ characterizes s_{ij} in that $s_{ij} \models \{\!|s_{ij}|\!\}$, and $s'_{ij} \not\models \{\!|s_{ij}|\!\}$ for all ij-states s'_{ij} such that $s'_{ij} \neq s_{ij}$.

Definition 22 (Sometimes-blocking, blk^j_i, blk_i) An i-state s_i is *sometimes-blocking in M_{ij}* if and only if:

$$\exists s^0_{ij} \in S^0_{ij} : M_{ij}, s^0_{ij} \models \mathsf{EF}(\ \{\!|s_i|\!\} \wedge (\exists a^i_j \in P^i_j : (\{\!|a^i_j.start|\!\} \wedge \neg a^i_j.guard))\ ).$$

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

Thus, s_i is sometimes-blocking in M_{ij} if and only if some (not necessarily all) reachable state s_{ij} in M_{ij} blocks some arc a^i_j of P^i_j and has s_i as its P_i-component. blk^j_i holds in s_{ij} (respectively, I-state s) exactly when $s_{ij}{\upharpoonright}i$ (respectively, $s{\upharpoonright}i$) is sometimes-blocking in M_{ij}. blk_i holds in I-state s iff some pair-projection $s{\upharpoonright}ij$ satisfies blk^j_i, for some $j \in I(i)$, i.e., if s_i is sometimes-blocking in M_{ij} for some $j \in I(i)$.

Definition 23 (Weak blocking fairness, Φ_b) The fairness notion of *weak blocking fairness* is given by the following PTL formula:

$$\Phi_b \overset{\text{df}}{=\!=} \bigwedge_{i \in dom(I)} \overset{\infty}{\mathsf{G}}(blk_i \wedge en_i) \Rightarrow \overset{\infty}{\mathsf{F}} ex_i.$$

Weak blocking fairness states that a process that is, continuously, enabled and in a sometimes-blocking state, must eventually be executed. The intuition is that a process P_i which might be blocking another process P_j must be executed, so that P_j can make progress.

Definition 24 (Pending eventuality, pnd_{ij}) 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}. pnd_{ij} holds in all pair-states with a pending eventuality.

Definition 25 (Weak eventuality fairness, Φ_ℓ) The fairness notion of *weak eventuality fairness* is given by the following PTL formula:

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

Weak eventuality fairness states that if there is, continuously, a pending eventuality in M_{ij}, and at least one of P_i, P_j are continuously enabled (within the global program), then eventually one of P_i, P_j is executed.

Our overall fairness notion Φ for global static programs is then the conjunction of weak blocking and weak eventuality fairness.

Definition 26 (Global static fairness, Φ) The fairness notion of *global static fairness* is given by the PTL formula $\Phi \overset{\text{df}}{=\!=} \Phi_b \wedge \Phi_\ell$.

We also need a condition on the behavior of pair-programs. This condition, together with global static fairness, prevents a pair-process P_i^j from monopolizing execution and preventing the other pair-process P_j^i from making progress.

Definition 27 (Static liveness condition for $(S_{ij}^0, P_i^j \| P_j^i)$) Let $M_{ij} = (S_{ij}^0, S_{ij}, R_{ij})$ be the pair-structure for $(S_{ij}^0, P_i^j \| P_j^i)$. Define $aen_j \overset{\text{df}}{=\!=} (\forall a_j^i \in P_j^i : \{\!|a_j^i.start|\!\} \Rightarrow a_j^i.guard)$. aen_j means that every arc of P_j^i whose start state is a component of the current pair-state is also enabled in the current pair-state. Then, the static liveness condition for $(S_{ij}^0, P_i^j \| P_j^i)$ is:

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

The liveness condition requires, in every pair-program $(S_{ij}^0, P_i^j \| P_j^i)$, that if P_i^j can execute continuously along some path, then there exists a suffix of that path along which P_i^j does not block any arc of P_j^i.

Lemma 3 (Progress for global static programs) *Let $\mathscr{I}$ be a global static specification, and let $I = \mathscr{I}.pairs$. For all $(k,\ell) \in I$, let $(S_{k\ell}^0, P_k^\ell \| P_\ell^k)$ be a pair-program for (k,ℓ), and let $(S_I^0, P_{i_1} \| \ldots \| P_{i_K})$ be the global static program synthesized (Def. 14) by our method from $\mathscr{I}$ via the $(S_{k\ell}^0, P_k^\ell \| P_\ell^k)$. Let M_I be the I-structure (Def. 15) of $(S_I^0, P_{i_1} \| \ldots \| P_{i_K})$, and let s be a reachable state of M_I. Let $(\{i,j\}, spec_{ij})$ be an arbitrary pair-specification in $\mathscr{I}$. If the following assumptions all hold*

1. *for all $(k,\ell) \in I$, the static liveness condition (Def. 27) holds for $(S^0_{k\ell}, P^\ell_k \,\|\, P^k_\ell)$,*
2. *for every reachable I-state u of M_I, $W_I(u)$ is supercycle-free,*
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 above and Proposition 1, in every reachable state there is some process with an enabled arc. 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

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

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 (a), $i \in \psi_{fin}$ and $j \in \psi_{fin}$.

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 arcs. By Assumption 2, W is supercycle-free. We apply Proposition 1 to the subprogram consisting of the processes in W, and so conclude that there exists a process P_k in W such that P_k has some arc a_k with no wait-for edges to any process in W. According to Definition 14, a_k results from the pairwise-composition (using $\otimes$) of some set of arcs a^ℓ_k, for all $\ell \in I(k)$. Let $en(a^\ell_k) = a^\ell_k.guard$, i.e., a^ℓ_k is enabled, and $en(a_k) = \bigwedge_{\ell \in I(k)} a^\ell_k.guard$, i.e., a_k is enabled, since, by Definition 14, $a_k.guard = \bigwedge_{\ell \in I(k)} a^\ell_k.guard$.

Hence, in state $t{\upharpoonright}k\ell$, a^ℓ_k is enabled in all pair-structures $M_{k\ell}$ such that $\ell \in \psi_{fin}$, i.e., $t{\upharpoonright}k\ell \models \{\!|a^\ell_k.start|\!\} \wedge en(a^\ell_k)$. 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 $\bigwedge \ell \in \psi_{fin} \cap I(k) : \rho{\upharpoonright}k\ell \models \mathsf{G} en(a^\ell_k)$.

Now consider a pair-structure $M_{k\ell}$ 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 2 and Corollary 2, $\rho{\upharpoonright}k\ell$ is a reachable 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 reachable fullpath in $M_{k\ell}$. By the liveness condition for static programs (Definition 27), $\rho{\upharpoonright}k\ell \models \overset{\infty}{\mathsf{G}} aen_k$. Now $t{\upharpoonright}k\ell \models \{\!|a^\ell_k.start|\!\}$. 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 $\rho{\upharpoonright}k\ell \models \mathsf{G}\{\!|a^\ell_k.start|\!\}$. Hence, by definition of aen_k, $\rho{\upharpoonright}k\ell \models \overset{\infty}{\mathsf{G}} en(a^\ell_k)$. Since ℓ is an arbitrary element of $\psi_{inf} \cap I(k)$, we have $\bigwedge \ell \in \psi_{inf} \cap I(k) : \rho{\upharpoonright}k\ell \models \overset{\infty}{\mathsf{G}} en(a^\ell_k)$. Since $(\psi_{inf} \cap I(k)) \cup (\psi_{fin} \cap I(k)) = I(k)$, we conclude $\bigwedge \ell \in I(k) : \rho{\upharpoonright}k\ell \models \overset{\infty}{\mathsf{G}} en(a^\ell_k)$. By Definitions 14 (synthesis) and 18 (path projection), we have $\rho \models \overset{\infty}{\mathsf{G}} en(a_k)$, We therefore conclude

$$\rho \models \overset{\infty}{\mathsf{G}} en_k. \tag{b}$$

Assume $k \notin \{i,j\}$. Then, by definition of W, in state t P_k blocks some arc a^k_ℓ of some process P_ℓ, i.e., $t \models \{\!|a^k_\ell.start|\!\} \wedge \neg a^k_\ell.guard$. By Definition 22, $t{\upharpoonright}k$ is sometimes-blocking in $M_{k\ell}$ (since t is reachable, so is $t{\upharpoonright}k$, by Corollary 2. Hence $t{\upharpoonright}k \models blk^\ell_k$, and so $t \models blk^\ell_k$. 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, $\rho \models \mathsf{G} blk^\ell_k$, since $t \models blk^\ell_k$. Thus $\rho \models \mathsf{G} blk_k$, by definition of blk_k. From this and (b), we have $\rho \models \overset{\infty}{\mathsf{G}}(blk_k \wedge en_k)$. Hence, by weak blocking fairness, (Definition 23), $\rho \models \overset{\infty}{\mathsf{F}} ex_k$, 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 (a), and also $s = \mathit{first}(\pi)$, 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 24. 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 (b) 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 25), $\pi \models \overset{\infty}{\mathsf{F}}(ex_i \vee ex_j)$. This contradicts the assumption (a), 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. □

5.4 The large model theorem for global static programs

The main technical result for our static synthesis method is that it is sound. This is given by the large model theorem, which states that any subformula of $spec_{ij}$ which holds in the ij-projection of a global state s also holds in s itself. That is, correctness properties satisfied by a pair-program executing in isolation also hold in the global static program $\mathscr{P}$ that our method synthesizes.

Theorem 1 (Large model) *Let $\mathscr{I}$ be a global static specification, and let I be its interconnection relation. For all $(k, \ell) \in I$, let $(S^0_{k\ell}, P^\ell_k \,\|\, P^k_\ell)$ be a pair-program for (k, ℓ), and let $(S^0_I, P_{i_1} \,\|\, \ldots \,\|\, P_{i_K})$ be the global static program synthesized (Def. 14) by our method from $\mathscr{I}$ via the $(S^0_{k\ell}, P^\ell_k \,\|\, P^k_\ell)$. Let M_I be the I-structure (Def. 15) of $(S^0_I, P_{i_1} \,\|\, \ldots \,\|\, P_{i_K})$, and let s be a reachable state of M_I. Let $(\{i, j\}, spec_{ij})$ be an arbitrary pair-specification in $\mathscr{I}$. If the following assumptions all hold:*

1. *for all $(k, \ell) \in I$, the static liveness condition (Def. 27) holds for $(S^0_{k\ell}, P^\ell_k \,\|\, P^k_\ell)$,*
2. *for every reachable I-state u of M_I, $W_I(u)$ is supercycle-free,*
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 Recall that $spec_{ij}$, and therefore f_{ij}, are formulae of ACTL^-_{ij}. 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 $p_i \in AP_i$, i.e., p_i is an atomic proposition.
By definition of $\upharpoonright ij$, s and $s\upharpoonright ij$ agree on all atomic propositions in $AP_i \cup AP_j$. 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 $M_I, s \models_\Phi (g_{ij} \vee h_{ij})$.

$f_{ij} = \mathsf{A}[g_{ij} \mathsf{W} h_{ij}]$. Let π be an arbitrary Φ-fair fullpath starting in s. We establish $\pi \models [g_{ij} \mathsf{W} h_{ij}]$. By Definition 18 (path projection), $\pi{\restriction}ij$ starts in $s{\restriction}ij = s_{ij}$. Hence, by CTL semantics, $\pi{\restriction}ij \models [g_{ij} \mathsf{W} h_{ij}]$ (note that this holds even if $\pi{\restriction}ij$ is not a fullpath, i.e., is a finite path). We have two cases.

Case 1: $\pi{\restriction}ij \models \mathsf{G} g_{ij}$. Let t be an arbitrary state along π. By Definition 18, $t{\restriction}ij$ lies along $\pi{\restriction}ij$. Hence $t{\restriction}ij \models g_{ij}$. By the induction hypothesis, $t \models g_{ij}$. Hence $\pi \models \mathsf{G} g_{ij}$, since t was arbitrarily chosen. Hence $\pi \models [g_{ij} \mathsf{W} h_{ij}]$ by CTL*semantics.

Case 2: $\pi{\restriction}ij \models [g_{ij} \mathsf{U} h_{ij}]$. Let $s_{ij}^{m'}$ be the first state along $\pi{\restriction}ij$ that satisfies h_{ij}[8]. By Definition 18, there exists at least one state t along π such that $t{\restriction}ij = s_{ij}^{m'}$. 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 18, $s^n{\restriction}ij$ lies along the portion of $\pi{\restriction}ij$ up to, and possibly including, $s_{ij}^{m'}$. That is, $s^n{\restriction}ij = s_{ij}^m$, where $0 \le m \le m'$. Now suppose $s^n{\restriction}ij = s_{ij}^{m'}$ (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 $\pi{\restriction}ij$ that satisfies h_{ij}, and $\pi{\restriction}ij \models [g_{ij} \mathsf{U} h_{ij}]$, we have $s_{ij}^m \models g_{ij}$ by CTL*semantics. From $s^n{\restriction}ij = s_{ij}^m$ 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 $\pi \models [g_{ij} \mathsf{W} h_{ij}]$ by CTL*semantics.

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

$f_{ij} = \mathsf{A}[g_{ij} \mathsf{U} h_{ij}]$. 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. 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 Proposition 1, 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 18, $\rho{\restriction}ij$ is a path from s_{ij} to $t{\restriction}ij$. By Lemma 2, $\rho{\restriction}ij$ is a path in M_{ij}. Suppose $\rho{\restriction}ij$ contains a state u_{ij} such that $u_{ij} \models h_{ij}$. By Definition 18, there exists a state u along ρ such that $u{\restriction}ij = u_{ij}$. By the induction hypothesis, we have $u \models_\Phi h_{ij}$, contradicting the assumption $\pi \models_\Phi \mathsf{G} \neg h_{ij}$. Hence $\rho{\restriction}ij$ contains no state that satisfies h_{ij}. Since $s_{ij} \models \mathsf{AF} h_{ij}$ and $\rho{\restriction}ij$ is a path from s_{ij} to $t{\restriction}ij$ (inclusive) which contains no state satisfying h_{ij}, we must have $t{\restriction}ij \models \neg h_{ij} \wedge \mathsf{AF} h_{ij}$ by CTL semantics. Let π' be the suffix of π starting in t. Since $t{\restriction}ij \models \neg h_{ij} \wedge \mathsf{AF} h_{ij}$ and $h_{ij} \in CL(spec_{ij})$, we can apply Lemma 3 (progress) 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 18, $\pi{\restriction}ij$ is a fullpath. By Lemma 2, $\pi{\restriction}ij$ is a fullpath in M_{ij}. Since $\pi{\restriction}ij$ starts in $s_{ij} = s{\restriction}ij$, and $s_{ij} \models \mathsf{AF} h_{ij}$, $\pi{\restriction}ij$ must contain a state v_{ij} such that $v_{ij} \models h_{ij}$. By Definition 18, π contains a state v such that $v{\restriction}ij = v_{ij}$. 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 $s_{ij} \models \mathsf{A}[g_{ij} \mathsf{W} h_{ij}]$. From the above proof case for $\mathsf{A}[g_{ij} \mathsf{W} h_{ij}]$, we have $s \models_\Phi \mathsf{A}[g_{ij} \mathsf{W} h_{ij}]$. Hence $\pi \models_\Phi [g_{ij} \mathsf{W} h_{ij}]$, since π is a Φ-fair fullpath

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

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 $s \models_\Phi \mathsf{A}[g_{ij} \mathsf{U} h_{ij}]$. □

Note that, if the liveness condition does not hold, then we can still verify safety properties of $(S_I^0, P_{i_1} \| \dots \| P_{i_K})$, since the cases for atomic propositions, $g_{ij} \wedge h_{ij}$, $g_{ij} \vee h_{ij}$, and $\mathsf{A}[g_{ij} \mathsf{W} h_{ij}]$ above still apply.

To apply the large model theorem to program correctness, we establish two corollaries. The first relates satisfaction of pair-specifications in initial states of the corresponding pair-programs, to satisfaction of pair-specifications in initial states of the global static program.

Corollary 3 (Large model) *Let $\mathscr{I}$ be a global static specification, and let I be its interconnection relation. For all $(k,\ell) \in I$, let $(S_{k\ell}^0, P_k^\ell \| P_\ell^k)$ be a pair-program for (k,ℓ), so that $M_{k\ell}, S_{k\ell}^0 \models spec_{k\ell}$. Let $(S_I^0, P_{i_1} \| \dots \| P_{i_K})$ be the global static program synthesized (Def. 14) by our method from $\mathscr{I}$ via the $(S_{k\ell}^0, P_k^\ell \| P_\ell^k)$. Let M_I be the I-structure (Def. 15) of $(S_I^0, P_{i_1} \| \dots \| P_{i_K})$. If both the following assumptions hold:*

1. *for all $(k,\ell) \in I$, the static liveness condition (Def. 27) holds for $(S_{k\ell}^0, P_k^\ell \| P_\ell^k)$, and*
2. *for every reachable I-state u of M_I, $W_I(u)$ is supercycle-free,*

then

$$M_I, S_I^0 \models_\Phi \textstyle\bigwedge_{ij} spec_{ij}.$$

Proof Follows immediately by applying Theorem 1 to every state $s \in S_I^0$ and every pair (i,j) in I, and setting f_{ij} to $spec_{ij}$. □

Unlike Attie and Emerson [5], $spec_{ij}$ and $spec_{k\ell}$, where $\{k,\ell\} \neq \{i,j\}$, can be completely different formulae, whereas in [5] these formulae had to be "similar," i.e., one was obtained from the other by substituting process indices.

Our second corollary enables us to deal with global specifications that are not in the form of a conjunction of pair-specifications. Any global specification that is logically implied by the conjunction of pair-specifications is also satisfied by the global static program. Let $\vdash_{\mathrm{CTL}}$ denote the deducibility relation of any sound and complete deductive system for CTL, see for example Emerson [29].

Corollary 4 (Large model) *Let $\mathscr{I}$ be a global static specification, and let I be its interconnection relation. For all $(k,\ell) \in I$, let $(S_{k\ell}^0, P_k^\ell \| P_\ell^k)$ be a pair-program for (k,ℓ), so that $M_{k\ell}, S_{k\ell}^0 \models spec_{k\ell}$. Let $(S_I^0, P_{i_1} \| \dots \| P_{i_K})$ be the global static program synthesized (Def. 14) by our method from $\mathscr{I}$ via the $(S_{k\ell}^0, P_k^\ell \| P_\ell^k)$. Let M_I be the I-structure (Def. 15) of $(S_I^0, P_{i_1} \| \dots \| P_{i_K})$. Also let $glob{-}spec$ be some formula of* ACTL$^-$. *If all the following hold:*

1. *for all $(k,\ell) \in I$, the static liveness condition (Def. 27) holds for $(S_{k\ell}^0, P_k^\ell \| P_\ell^k)$,*
2. *for every reachable I-state u, $W_I(u)$ is supercycle-free*
3. *$(\bigwedge_{ij} spec_{ij}) \vdash_{\mathrm{CTL}} glob{-}spec$,*

then

$$M_I, S_I^0 \models_\Phi glob{-}spec.$$

Proof By Corollary 3, we have $M_I, S_I^0 \models_\Phi \bigwedge_{ij} spec_{ij}$. From $(\bigwedge_{ij} spec_{ij}) \vdash_{\mathrm{CTL}} glob{-}spec$ and soundness of the CTL deductive system, any model of $\bigwedge_{ij} spec_{ij}$ is also a model of $glob{-}spec$. Hence $M_I, S_I^0 \models_\Phi glob{-}spec$. □

5.5 Complexity of the method for synthesis of global static programs

Let the size (number of local states) of each process be $O(N)$. Then checking $M_{k\ell}, S^0_{k\ell} \models spec_{k\ell})$ is in $O(N^2)$ using the model-checking algorithm of Clarke, Emerson, and Sistla [21]. Checking the static liveness condition for every $(S^0_{ij}, P^j_i \,\|\, P^i_j)$ is also $O(N^2)$. Roughly, this requires finding all the strongly connected components CC in M_{ij} which contain no transition by P^i_j, since these include every infinite path along which P^j_i executes forever while P^i_j does not execute. We then check that all ready arcs of P^i_j are enabled in every state of CC (aen_j). Finding the strongly connected components can be done in time linear in the size of M_{ij}, using Tarjan's algorithm [53]. Checking the needed enablements is also clearly linear in the size of M_{ij}. Since there are $O(|I|)$ pair-structures, we obtain a total complexity of $O(|I| \cdot N^2)$.

Checking supercycle-freedom requires applying the methods in Attie and Emerson [5], Attie and Chockler [12], or Attie et. al. [9]. Attie and Chockler [12] give two methods, with running times $O(K^3N^3n)$ and $O(K^4N^4)$, where n is the maximum degree of the graph of some process. The method of Attie et. al. [9] starts with a small subsystem and attempts to verify a criterion for supercycle-freedom. If the check fails, then the subsystem is made larger (which reduces false negatives, in general) and the check repeated. The method often succeeds for small subsystems.

5.6 Correctness of the two phase commit protocol

Let $M_I = (S^0_I, S_I, R_I)$ be the I-structure for P, the concurrent program synthesized by our method for the two-phase commit example, and given in Figure 7. A key correctness property of two-phase commit is *consistent commit*: if the coordinator commits, then so does every participant. Figure 8 presents a proof which derives the consistent commit property from the relevant pair-properties. The first column gives line numbers. The second column is the formula f that is shown to hold in M_I, i.e., $M_I, S^0_I \models f$. The third column gives the justification for each formula, which is "Pair-property" for pair-properties, and $\vdash_{\mathrm{CTL}}$ for formulae deduced from formulae established earlier in the proof. We model-checked each pair-property formula $f_{k\ell}$ in its respective pair-structure $M_{k\ell}$, using the Eshmun [10] model checker and repairer. This gives us $M_{k\ell}, S^0_{k\ell} \models f_{k\ell}$. We then have $M_I, S^0_I \models f_{k\ell}$ by Corollary 3. For a deduced formula, we have $M_I, S^0_I \models f_{k\ell}$ by Corollary 4. Validity of the deductions of Formulae 9 and 11 was also checked using Eshmun, which implements the CTL decision procedure, and so can check validity of CTL formulae. The deduction of Formulae 10 and 15 was manual, as described below. The fourth column (sometimes continued on the following line) describes the meaning of the formula.

Several deduction steps rely on the transitivity of temporal implication $\longrightarrow$: $(f \longrightarrow g \wedge g \longrightarrow h) \Rightarrow f \longrightarrow h$. Intuitively this is straightforward, and we also verified it in Eshmun by checking the validity of $(\mathsf{A}[(f \Rightarrow \mathsf{AF}g)\mathsf{W}g] \wedge \mathsf{A}[(g \Rightarrow \mathsf{AF}h)\mathsf{W}h]) \Rightarrow \mathsf{A}[(f \Rightarrow \mathsf{AF}h)\mathsf{W}h]$.

Formula 10 follows from Formula 9, and the observation that, if cm_i occurs along a path π, then, by Formulae 6 and 7, ab_i cannot also occur along π. Hence if $\mathsf{A}[sb_{i+1}\mathsf{U}(sb_{i+1} \wedge (cm_i \vee ab_i))$ holds in some state s along π, it follows that $\mathsf{A}[sb_{i+1}\mathsf{U}(sb_{i+1} \wedge cm_i)]$ also holds in s. Formula 15 follows from Formula 14 and the observation that commitment is stable (Formula 7). Formula 15 gives a main correctness property of two phase commit: if the coordinator P_0 commits, then all processes commit together.

In a similar manner, we deduce, using Corollary 3, both $\bigwedge_{1 \leq i < n}(ab_{i-1} \longrightarrow ab_i)$, i.e., abort of a process implies the abort of its successor, and $\bigwedge_{0 \leq i < n} \mathsf{AG}(ab_i \Rightarrow \mathsf{AG}\, ab_i)$, i.e., abort is stable. From these, $ab_0 \longrightarrow \bigwedge_{0\le i<n} ab_i$ follows logically. That is, if the coordinator aborts, then so does every participant. Hence, by Corollary 4, P satisfies this property, which is another main correctness property of two phase commit. We also establish $\mathsf{AF}(cm_0 \vee ab_0)$, the coordinator eventually decides, using Corollary 3. We could also establish $\bigwedge_{1\le i<n} \mathsf{AG}(st_i \Rightarrow \mathsf{EX}_i ab_i)$, every participant can abort unilaterally. This last formula is not in ACTL^-_{ij}, but it was shown to be preserved in Attie & Emerson [5], and it is straightforward to extend the proof there to the setting of this paper. We omit the details.

6 Synthesis of dynamic concurrent programs

6.1 Global dynamic specifications

A global static specification is a finite set of pair-specifications. We generalize this idea by allowing pair-specifications to be added dynamically, at run time. Once a pair-specification $(\{i,j\}, spec_{ij})$ has been added to the global dynamic specification, we say that $(\{i,j\}, spec_{ij})$ is *in force*. This leads to our second synthesis method, which caters to such global *dynamic* specifications by adding appropriate pair-programs to the synthesized global program, at run time. Since the interconnection relation I is, in effect, dynamically changing, we drop the I subscript, e.g., in $M_I, S^0_I, S_I, R_I, W_I(s)$.

Definition 28 (Global dynamic specification)
A *global dynamic specification* $\mathscr{D}$ consists of:

1. A *universal* set $\mathscr{PS}$ of *pair-specifications*.
2. A finite set $\mathscr{PS}_0 \subseteq \mathscr{PS}$, which gives the pair-specifications which are *initially in force*.
3. A mapping $\mathsf{create} : 2^{\mathscr{PS}} \mapsto 2^{\mathscr{PS}}$ which determines which new pair-specifications (in $\mathscr{PS}$) can be added to those that are in-force. If $\mathscr{I}$ is the set of pair-specifications that are in-force and $(\{i,j\}, spec_{ij}) \in \mathsf{create}(\mathscr{I})$, then $\mathscr{I} \cup \{(\{i,j\}, spec_{ij})\}$ is a possible next value for the set of pair-specifications that are in-force.

We show in the sequel that the synthesized global 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.

6.2 Running example—the eventually serializable data service

The eventually-serializable data service (ESDS) of Fekete et. al. [35] and Ladin et. al. [42] 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 $\mathscr{O}$ be the (countable) set of all operations, $\mathscr{R}$ the set of all replicas $client(x)$ be the

1.	$cm_0 \longrightarrow sb_{n-1}$	Pair-property	P_0 commits only if P_{n-1} submits
2.	$\bigwedge_{1 \le i < n}(sb_i \longrightarrow sb_{i-1})$	Pair-property	P_i submits only if P_{i-1} submits
3.	$\bigwedge_{1 \le i < n}(cm_0 \longrightarrow sb_i)$	1, 2, $\longrightarrow$ is transitive	P_0 commits only if P_i submits
4.	$\bigwedge_{1 \le i < n}(cm_i \longrightarrow cm_{i-1})$	Pair-property	P_i commits only if P_{i-1} commits
5.	$\bigwedge_{0 \le i < n} \bigwedge_{1 \le j < n}(cm_i \longrightarrow sb_j)$	3, 4, $\longrightarrow$ is transitive	P_i commits only if P_j submits
6.	$\bigwedge_{0 \le i < n} \mathsf{AG}(\neg cm_i \vee \neg ab_i)$	Pair-property	no process both commits and aborts
7.	$\bigwedge_{0 \le i < n} \mathsf{AG}(cm_i \Rightarrow \mathsf{AG}(cm_i))$	Pair-property	commitment is stable
8.	$\bigwedge_{1 \le i < n} \mathsf{AG}[sb_i \Rightarrow \mathsf{A}[sb_i \mathsf{U}(sb_i \wedge (cm_{i-1} \vee ab_{i-1}))]]$	Pair-property	if P_{i+1} submits, then it remains in the submit state until P_i decides
9.	$\bigwedge_{1 \le i < n}(cm_{i-1} \longrightarrow \mathsf{A}[sb_i \mathsf{U}(sb_i \wedge (cm_{i-1} \vee ab_{i-1}))])$	5, 8, $\vdash_{\text{CTL}}$	if P_{i-1} commits, then P_i submits and remains in the submit state until P_{i-1} decides
10.	$\bigwedge_{1 \le i < n}(cm_{i-1} \longrightarrow \mathsf{A}[sb_i \mathsf{U}(sb_i \wedge cm_{i-1})])$	6, 7, 9, $\vdash_{\text{CTL}}$	if P_{i-1} commits, then P_i submits and remains in the submit state until P_{i-1} commits
11.	$\bigwedge_{1 \le i < n}(cm_{i-1} \longrightarrow (cm_{i-1} \wedge sb_i))$	10, $\vdash_{\text{CTL}}$	if P_{i-1} commits, then P_{i-1} commits and P_i submits
12.	$\bigwedge_{1 \le i < n}((cm_{i-1} \wedge sb_i) \longrightarrow cm_i)$	Pair-property	if P_{i-1} commits and P_i submits, then P_i commits
13.	$\bigwedge_{1 \le i < n}(cm_{i-1} \longrightarrow cm_i)$	11, 12, $\longrightarrow$ is transitive	if P_{i-1} commits then P_i commits
14.	$\bigwedge_{1 \le i < n}(cm_0 \longrightarrow cm_i)$	13, $\longrightarrow$ is transitive	if P_0 commits then so does P_i
15.	$cm_0 \longrightarrow (\bigwedge_{0 \le i < n} cm_i)$	7, 14, $\vdash_{\text{CTL}}$	if P_0 (the coordinator) commits, then so does every participant

Fig. 8 Derivation of the consistent commit correctness property

client issuing operation x, $replica(x)$ be the replica that handles operation x. We use x, x', x'' to index over operations, c to index over clients, and r, 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 that it issues. As the client issues operations, these processes are created dynamically. Likewise a replica consists of many processes, one for each operation that it handles. 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.

We define the following atomic predicates for operation x:

- *in* is the initial state.
- *wt* means that x is submitted but not yet done.
- *dn* means that x is done, i.e., the response to x has been computed.
- *st* means that x is stable.
- *snt* means that the result of x has been sent to the client.

The pair-specifications are as follows.

Local structure of clients C_c^x

in_c^x: x is initially pending

$\mathsf{AG}(in_c^x \Rightarrow (\mathsf{AX}_c wt_c^x \wedge \mathsf{EX}_c wt_c^x)) \wedge \mathsf{AG}(wt_c^x \Rightarrow \mathsf{AX}_c dn_c^x) \wedge \mathsf{AG}(dn_c^x \Rightarrow (\mathsf{AX}_c dn_c^x \wedge \mathsf{EX}_c dn_c^x))$: 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.

$\mathsf{AG}((in_c^x \equiv \neg(wt_c^x \vee dn_c^x)) \wedge (wt_c^x \equiv \neg(in_c^x \vee dn_c^x)) \wedge (dn_c^x \equiv \neg(in_c^x \vee wt_c^x)))$: 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 9, 10, and 11. We omit the temporal logic formulae, as they are obvious, and are constructed in an analogous manner to those for the clients.

Pair-specification for $C_c^x \| R_r^x$ (client-replica interaction) where $x \in \mathscr{O}$, $c = client(x)$, $r = replica(x)$

Local structure specifications for C_c^x and R_r^x

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

$\mathsf{A}[(wt_c^x \Rightarrow \mathsf{AF} wt_r^x)\mathsf{W} wt_r^x]$: every submitted x is 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 at its client, it remains done

$\mathsf{AG}(dn_c^x \Rightarrow snt_r^x)$: a client does not receive a result before it is sent

Pair-specification for $R_r^x \| R_{r'}^{x'}$ (CSC constraints) where $x \in \mathscr{O}$, $x' \in x.prev$, $r = replica(x)$, $r' = replica(x')$

Local structure specifications for R_r^x and $R_{r'}^{x'}$

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

Pair-specification for $R_r^x \| R_{r'}^x$, where $x \in \mathscr{O}$, $x.strict$, $r = replica(x)$, $r' \in \mathscr{R} - \{replica(x)\}$

$\mathsf{AG}(snt_r^x \Rightarrow st_{r'}^x)$: the result of a strict operation is not sent to the client until it is stable at all replicas (strictness constraints)

$\mathsf{AG}(snt_r^x \Rightarrow \mathsf{AG} snt_r^x)$: once operation results are sent, they remain sent

$\mathsf{AG}(wt_r^x \Rightarrow \mathsf{AF} st_{r'}^x)$: every submitted operation eventually stabilizes

6.3 Conjunctive overlay of pair-programs

Our second synthesis method produces a global dynamic program $\mathscr{P}$. $\mathscr{P}$ consists of the *conjunctive overlay* of a dynamically and monotonically increasing set of *pair-programs*.

Definition 29 (Conjunctive overlay, $P_i^j \otimes P_i^k$) Let P_i^j and P_i^k be pair-processes for i such that $graph(P_i^j) = graph(P_i^k)$. Then,

$P_i^j \otimes P_i^k$ contains an arc from s_i to t_i with label $(\bigoplus_{\ell \in [1:n_j]} B_{i,\ell}^j \to A_{i,\ell}^j) \otimes (\bigoplus_{\ell \in [1:n_k]} B_{i,\ell}^k \to A_{i,\ell}^k)$
iff
P_i^j contains an arc from s_i to t_i with label $\bigoplus_{\ell \in [1:n_j]} B_{i,\ell}^j \to A_{i,\ell}^j$ and
P_i^k contains an arc from s_i to t_i with label $\bigoplus_{\ell \in [1:n_k]} B_{i,\ell}^k \to A_{i,\ell}^k$.

Note that the $\otimes$ operator is now overloaded, and applies to both pair-processes and to guarded commands. When applied to guarded commands, $\otimes$ denotes the "conjunction" of guarded commands, as already defined for the static case. That is, an arc with label $(\bigoplus_{\ell \in [1:n_j]} B_{i,\ell}^j \to A_{i,\ell}^j) \otimes (\bigoplus_{\ell \in [1:n_k]} B_{i,\ell}^k \to A_{i,\ell}^k)$ can only be executed in a state in which $B_{i,\ell}^j$ holds for some $\ell \in [1:n_j]$ and $B_{i,\ell'}^k$ holds for some $\ell' \in [1:n_k]$. Execution then involves the parallel execution of the corresponding $A_{i,\ell}^j$ and $A_{i,\ell'}^k$. We also note that the overloaded version of $\otimes$ remains commutative and associative, and so we can use the n-ary version of $\otimes$.

Given a global dynamic program $\mathscr{P}$, a new pair-program $(S_{ij}^0, P_i^j \parallel P_j^i)$ is dynamically added at run-time as follows. If $\mathscr{P}$ already contains a process P_i, then P_i is modified by taking the conjunctive overlay with P_i^j, i.e., $P_i := P_i \otimes P_i^j$. If $\mathscr{P}$ 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_i^j, i.e., $P_i := P_i^j$. Hence, in every reachable state, $P_i = \bigotimes_{j \in I(i)} P_i^j$, where I is the "current" (dynamically changing) interconnection relation. That is, each process P_i is built up by successive conjunctive overlays of pair-processes. The synchronization skeleton code of the dynamic program thus changes at run time, as pair-programs are added, and we say that a pair-program $(S_{ij}^0, P_i^j \parallel P_j^i)$ is *active* once it has been added.

For $P_i = \bigotimes_{j \in I(i)} P_i^j$ to be well-defined, we require that $graph(P_i) = graph(P_i^j)$ for all $j \in I(i)$. To assure this, we require, in the sequel, that the following hold in all reachable states:

Definition 30 (Dynamic process graph consistency assumption) For active pair-programs $(S_{ij}^0, P_i^j \parallel P_j^i)$ and $(S_{ik}^0, P_i^k \parallel P_k^i)$: $graph(P_i^j) = graph(P_i^k)$.

We emphasize that different pair-programs can be non-isomorphic and can have different functionality, since the guarded commands which label the arcs of P_i^j and P_i^k can be different. Pair-programs are added only when a new pair-specification comes into force, and are the means of satisfying the new pair-specification. Thus, the transitions of $\mathscr{P}$ are of two kinds: (1) *normal* transitions, which are atomic transitions (as described in Section 2.1) arising from execution of the conjunctive overlay of all active pair-programs, and (2) *create* transitions, which correspond to making a new pair-specification $(\{i,j\}, spec_{ij})$ in-force, according to the create mapping. To satisfy $(\{i,j\}, spec_{ij})$, we dynamically create a new pair-program $(S_{ij}^0, P_i^j \parallel P_j^i)$ such that $(S_{ij}^0, P_i^j \parallel P_j^i)$ satisfies $spec_{ij}$, and incorporate it into the existing global dynamic program by performing a conjunctive overlay with the currently active P_i and P_j.

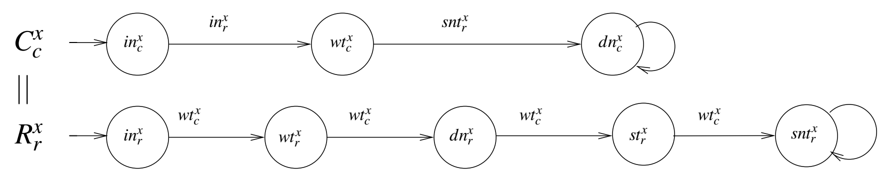

Fig. 9 Pair-program $C_c^x \,||\, R_r^x$, $r = replica(x)$: client-replica interaction.

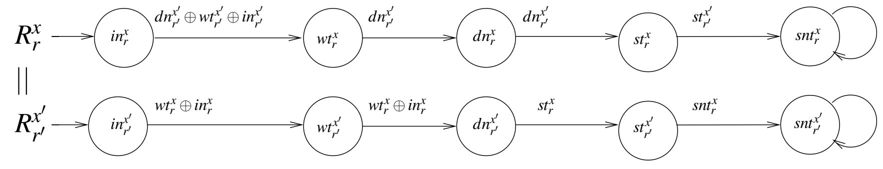

Fig. 10 Pair-program $R_r^x \,||\, R_{r'}^{x'}$, $r = replica(x)$, $x' \in x.prev$, $r' = replica(x')$: *CSC* constraints.

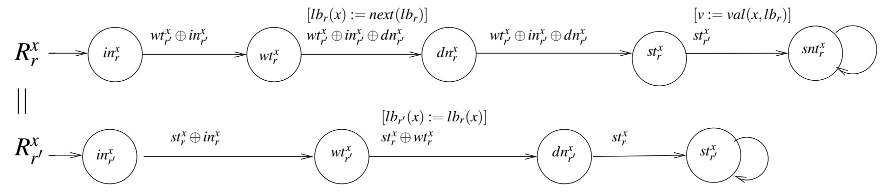

Fig. 11 Pair-program $R_r^x \,||\, R_{r'}^x$, when x is strict, $r = replica(x)$, $r' \in \mathscr{R} - \{replica(x)\}$: strictness constraints and eventual stabilization.

6.4 ESDS Pair-programs

The pair-programs for ESDS, synthesized from the above pair-specifications for ESDS, are given in Figures 9, 10, and 11. Recall that we gave pair-specifications 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 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. Figure 12 gives the pair-program $R_r^x \,||\, R_{r'}^x$ when x is not strict. Note the new state $sntu_r^x$ (response sent in an *u*n-strict manner). These pair-programs were produced using the Eshmun [10] tool, by starting with a trivial Kripke structure (all transitions present), then repairing w.r.t. the relevant pair-specification, and finally extracting the pair-program from the repaired Kripke structure, by projecting it onto the individual process indices [6, 30].

Out treatment of strict/non-strict operations illustrates the locality and modifiability [43] of the programs produced by our synthesis method. To change an operation x from strict to non-strict, or vice-versa, we only need to modify the pair-programs that handle x. Pair-programs in which x is not mentioned do not need to be modified.

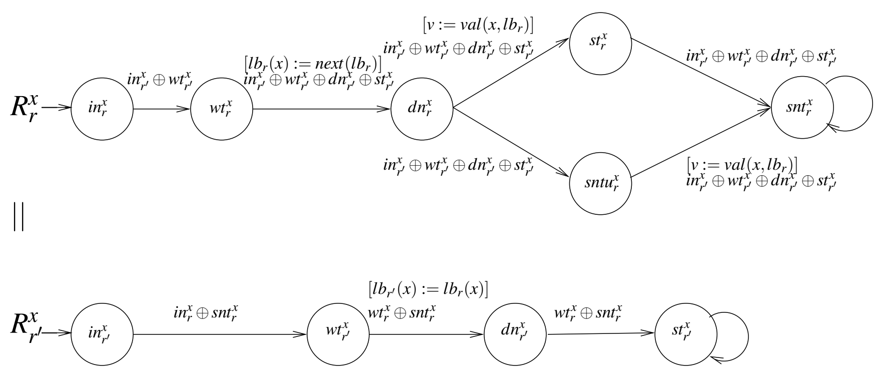
R^x_r
in^x_r
$in^x_{r'} \oplus wt^x_{r'}$
wt^x_r
$[lb_r(x) := next(lb_r)]$
$in^x_{r'} \oplus wt^x_{r'} \oplus dn^x_{r'} \oplus st^x_{r'}$
dn^x_r
$[v := val(x, lb_r)]$
$in^x_{r'} \oplus wt^x_{r'} \oplus dn^x_{r'} \oplus st^x_{r'}$
st^x_r
$in^x_{r'} \oplus wt^x_{r'} \oplus dn^x_{r'} \oplus st^x_{r'}$
$sntu^x_r$
$in^x_{r'} \oplus wt^x_{r'} \oplus dn^x_{r'} \oplus st^x_{r'}$
$[v := val(x, lb_r)]$
$in^x_{r'} \oplus wt^x_{r'} \oplus dn^x_{r'} \oplus st^x_{r'}$
snt^x_r
$\|$
$R^x_{r'}$
$in^x_{r'}$
$in^x_r \oplus snt^x_r$
$wt^x_{r'}$
$[lb_{r'}(x) := lb_r(x)]$
$wt^x_r \oplus snt^x_r$
$dn^x_{r'}$
$wt^x_r \oplus snt^x_r$
$st^x_{r'}$

Fig. 12 Pair-program $R^x_r \,\|\, R^x_{r'}$, when x is not strict: eventual stabilization.

6.5 Technical preliminaries for dynamic synthesis

In the sequel, we re-define the identifier $\mathscr{I}$ to indicate the "current" set of pair-specifications, which in general increases monotonically as execution proceeds. Since $\mathscr{I}$ as given in Definition 7 indicates a fixed static set of pair-specifications, this re-definition captures our intent that the set of pair-specifications is now dynamically changing. Define, as before, $\mathscr{I}.pairs = \{\{i,j\} \mid \exists \mathsf{PS} \in \mathscr{I} : \{i,j\} = \mathsf{PS}.procs\}$, and $\mathscr{I}.procs = \{i \mid \exists j : \{i,j\} \in \mathscr{I}.pairs\}$. Processes i and j are *neighbors* when $\{i,j\} \in \mathscr{I}.pairs$.

Definition 31 (Configuration, consistent configuration) A *configuration* s is a tuple $(\mathscr{I}, \mathscr{A}, \mathscr{S})$, where $\mathscr{I} \subseteq \mathscr{PS}$ is a set of pair-specifications, $\mathscr{A}$ is a set of pair-programs $(S^0_{ij}, P^j_i \,\|\, P^i_j)$, one for each $\{i,j\} \in \mathscr{I}.pairs$, and $\mathscr{S}$ is a mapping from each $\{i,j\} \in \mathscr{I}.pairs$ to an ij-state of $(S^0_{ij}, P^j_i \,\|\, P^i_j)$. We refer to the components of s as $s.\mathscr{I}$, $s.\mathscr{A}$, $s.\mathscr{S}$. We write $s.procs$ for $s.\mathscr{I}.procs$, and $s.pairs$ for $s.\mathscr{I}.pairs$.

A *consistent configuration* satisfies the constraint that all pair-states assign the same local state to all common processes, i.e., for all $\{i,j\},\{i,k\} \in \mathscr{I}.pairs$, if $\mathscr{S}(\{i,j\}) = (s_i, s_j, v^1_{ij}, \ldots, v^m_{ij})$ and $\mathscr{S}(\{i,k\}) = (s'_i, s_k, v^1_{ik}, \ldots, v^m_{ik})$, then $s_i = s'_i$. We assume henceforth that configurations are consistent, and our definitions will respect this constraint.

Definition 32 (Dynamic interconnection relation) Define $I_s \overset{\text{df}}{=\!=} \{(i,j) \mid \{i,j\} \in s.\mathscr{I}.pairs\}$, that is $I_s = \{(i,j) \mid \exists \mathsf{PS} \in s.\mathscr{I} : \{i,j\} = \mathsf{PS}.procs\}$.

The notation I_s is intended to suggest an interconnection relation that varies with the current configuration s. We define $I_s(i) = \{j \mid i I_s j\}$, and $\hat{I}_s(i) = \{i\} \cup \{j \mid i I_s j\}$. If $I_s \neq \emptyset$, then $I_s(i) \neq \emptyset$ for all $i \in s.\mathscr{I}.procs$, by definition. Thus, every process always has at least one neighbor.

Definition 33 (Dynamic spatial modality) We introduce the *dynamic spatial modality* $\bigwedge^s_{ij}$ which quantifies over all pairs (i,j) in I_s. Thus, $\bigwedge^s_{ij} spec_{ij}$ is equivalent to $\forall (i,j) \in I_s : spec_{ij}$.

Definition 34 (Projection) We extend the state projection operator $\upharpoonright$ to configurations. Let $s = (\mathscr{I}, \mathscr{A}, \mathscr{S})$. For projection of s onto a single process: if $i \in s.procs$, then $s{\upharpoonright}i =$

$\mathscr{S}(\{i,j\})\upharpoonright i$, where $\{i,j\} \in I_s$. This is unique because configurations are consistent. For projection of s onto a pair-program: if $\{i,j\} \in I_s$, then $s\upharpoonright ij = \mathscr{S}(\{i,j\})$. If $\{i,j\} \notin I_s$, then $s\upharpoonright ij$ is undefined. If J is a set of pairs such that $J \subseteq I_s$, then we define the projection of s onto J: $s\upharpoonright J$ is the restriction of $s.\mathscr{S}$ to J.

For configuration s, $i \in s.procs$, and atomic proposition $p_i \in AP_i$, we define $s(p_i) = s\upharpoonright i(p_i)$. That is, a configuration s inherits the assignments to atomic propositions in AP_i that are made by its i-states $s\upharpoonright i$. As in the static case, we require at least one initial global state.

Definition 35 (Dynamic initial configuration assumption) Let $S^0 = \{s \mid \forall (i,j) \in \mathscr{PS}_0 : s\upharpoonright ij \in S^0_{ij}\}$. That is, S^0 is the set of configurations that project onto the initial states S^0_{ij} of all the pair-programs that are active initially. We assume that these S^0_{ij} are such that $S^0 \neq \emptyset$, i.e., there exists some initial configuration.

6.6 Formal definition of the method for synthesis of global dynamic programs

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

1. Initially, $\mathscr{P}$ consists of the conjunctive overlay of the pair-programs corresponding to the pair-specifications in $\mathscr{PS}_0$.
2. When a pair-specification $(\{i,j\}, spec_{ij})$ is added, as permitted by the create mapping, synthesize a *pair-program* $(S^0_{ij}, P^j_i \parallel P^i_j)$ using $spec_{ij}$ as the specification, and add it to $\mathscr{P}$ as discussed in Section 6.3 above.

To synthesize pair-programs, any synthesis method which produces static concurrent programs for the synchronization skeleton model of concurrency can be used, for example the methods given in [6, 11, 30].

Since the create transitions affect the actual code of $\mathscr{P}$, we define them first. The create transitions are determined by the intended meaning of the create mapping, together with the constraint that creating a new pair-program does not change the current state of existing pair-programs. Syntactically, the create mapping is specified as a *creation rule*, which is an input to the synthesis method.

Definition 36 (Create transitions) Let s,t be configurations. Then (s, create, t) is a *create transition* iff there exists $\{i,j\} \notin I_s$ such that

1. $(\{i,j\}, spec_{ij}) \in create(s.\mathscr{I})$, i.e., the rule for adding new pair-specifications allows the pair-specification $(\{i,j\}, spec_{ij})$ to be added in configuration s.
2. $t.\mathscr{I} = s.\mathscr{I} \cup (\{i,j\}, spec_{ij})$, and $t.\mathscr{A} = s.\mathscr{A} \cup \{(S^0_{ij}, P^j_i \parallel P^i_j)\}$ for some $(S^0_{ij}, P^j_i \parallel P^i_j)$ such that $(S^0_{ij}, P^j_i \parallel P^i_j) \models spec_{ij}$.
3. $t\upharpoonright ij$ is a reachable state of $(S^0_{ij}, P^j_i \parallel P^i_j)$, and if $i \in s.procs$ then $t\upharpoonright i = s\upharpoonright i$, and if $j \in s.procs$ then $t\upharpoonright j = s\upharpoonright j$
4. for all $\{k,\ell\} \in I_s : s.\mathscr{S}(\{k,\ell\}) = t.\mathscr{S}(\{k,\ell\})$

That is, the result of a create transition (s, create, t) is a final configuration t with one more pair-program than the initial configuration s. This new pair-program starts execution in one of its reachable states. The consituent processes of the newly created pair-program may or may not be present in s. If present, they must have the same local state in s as in t. Existing pair-programs do not change state. 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 37 (Pairwise synthesis of global dynamic programs) In configuration s, the synthesized program $\mathscr{P}$ is given by $\mathscr{P} = \|_{i \in s.procs} P_i$, where $P_i = \bigotimes_{j \in I_s(i)} P_i^j$.

The set of initial configurations S^0 of $\mathscr{P}$ consists of all s such that

1. $s.\mathscr{I} = \mathscr{P}\mathscr{S}_0$, i.e., initially the pair-specifications in force are those in $\mathscr{P}\mathscr{S}_0$
2. $s.\mathscr{A}$ contains exactly one pair-program $(S^0_{ij}, P_i^j \,\|\, P_j^i)$ for each $(\{i,j\}, spec_{ij}) \in \mathscr{P}\mathscr{S}_0$, i.e., each pair-specification has a corresponding pair-program
3. $(S^0_{ij}, P_i^j \,\|\, P_j^i) \models spec_{ij}$, i.e., each pair-program satisfies its corresponding pair-specification, and
4. $s.\mathscr{S}(\{i,j\}) \in S^0_{ij}$ for all $\{i,j\} \in I_s$, i.e., the state of every pair-program $(S^0_{ij}, P_i^j \,\|\, P_j^i)$ in configuration s is one of its initial states.

Another way to characterize process P_i of $\mathscr{P}$ is that P_i contains an arc from s_i to t_i with label $\bigotimes_{j \in I_s(i)} \bigoplus_{\ell \in [1:n_j]} B^j_{i,\ell} \to A^j_{i,\ell}$ iff $\forall j \in I_s(i) : P_i^j$ contains an arc from s_i to t_i with label $\bigoplus_{\ell \in [1:n_j]} B^j_{i,\ell} \to A^j_{i,\ell}$.

Definition 37 gives the initial configurations S^0 of $\mathscr{P}$, and the code of $\mathscr{P}$ as a function of the $s.\mathscr{I}$ and $s.\mathscr{A}$ components of the current configuration s. The code of $\mathscr{P}$ does not depend on the $s.\mathscr{S}$ component of s, which gives the values of the atomic propositions and shared variables, i.e., the state. Definition 36 shows how $s.\mathscr{I}$ and $s.\mathscr{A}$ are changed by create transitions. Since a configuration of $\mathscr{P}$ 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, as follows.

Definition 38 (Normal transitions) Let s,t be configurations and $i \in s.procs$. Then (s,i,t) is a *normal transition* iff

1. there exists an arc in P_i from $s{\upharpoonright}i$ to $t{\upharpoonright}i$ with label $\bigotimes_{j \in I_s(i)} \bigoplus_{\ell \in [1:n_j]} B^j_{i,\ell} \to A^j_{i,\ell}$, such that
$$\forall j \in I_s(i), \exists m \in [1:n_j] : s{\upharpoonright}ij(B^j_{i,m}) = true \text{ and } \langle (s{\upharpoonright}ij){\upharpoonright}SH_{ij} \rangle A^j_{i,m} \langle (t{\upharpoonright}ij){\upharpoonright}SH_{ij} \rangle$$
2. $\forall j \in s.procs - \{i\} : s{\upharpoonright}j = t{\upharpoonright}j$, and
3. $\forall (j,k) \in I_s,\ i \notin \{j,k\} : s{\upharpoonright}jk = t{\upharpoonright}jk$.
4. $s.\mathscr{I} = t.\mathscr{I}$ and $s.\mathscr{A} = t.\mathscr{A}$

$\langle (s{\upharpoonright}ij){\upharpoonright}SH_{ij} \rangle A^j_{i,m} \langle (t{\upharpoonright}ij){\upharpoonright}SH_{ij} \rangle$ is Hoare triple notation [39] for total correctness, which in this case means that execution of $A^j_{i,m}$ always terminates,[9] and, when the shared variables in SH_{ij} have the values assigned by $s{\upharpoonright}ij$, leaves these variables with the values assigned by $t{\upharpoonright}ij$. $s{\upharpoonright}ij(B^j_{i,m}) = true$ states that the value of guard $B^j_{i,m}$ in state $s{\upharpoonright}ij$ is *true*.

6.7 Synthesized global dynamic concurrent program for ESDS

The pair-specifications for ESDS are given in Section 6.2 above, and the resulting synthesized pair-programs are given in Figures 9, 10, and 11. It remains to give the creation rule, which is as follows.

Rule for Dynamic process creation

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:

- $C^x_c \,\|\, R^x_r$ where $r = replica(x)$,

[9] Termination is obvious, since A is a parallel assignment and the right-hand side of $A^j_{i,m}$ is a list of constants.

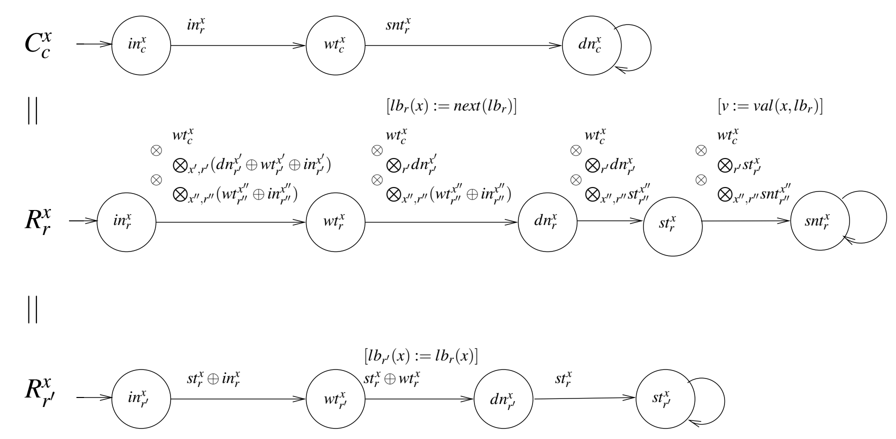
C_c^x
in_c^x
in_r^x
wt_c^x
snt_r^x
dn_c^x
$||$
$[lb_r(x) := next(lb_r)]$
$[v := val(x, lb_r)]$
$wt_c^x \otimes \bigotimes_{x',r'}(dn_{r'}^{x'} \oplus wt_{r'}^{x'} \oplus in_{r'}^{x'}) \otimes \bigotimes_{x'',r''}(wt_{r''}^{x''} \oplus in_{r''}^{x''})$
$wt_c^x \otimes \bigotimes_{r'} dn_{r'}^{x'} \otimes \bigotimes_{x'',r''}(wt_{r''}^{x''} \oplus in_{r''}^{x''})$
$wt_c^x \otimes \bigotimes_{r'} dn_{r'}^{x} \otimes \bigotimes_{x'',r''} st_{r''}^{x''}$
$wt_c^x \otimes \bigotimes_{r'} st_{r'}^{x} \otimes \bigotimes_{x'',r''} snt_{r''}^{x''}$
R_r^x
in_r^x
wt_r^x
dn_r^x
st_r^x
snt_r^x
$||$
$[lb_{r'}(x) := lb_r(x)]$
$R_{r'}^x$
$in_{r'}^x$
$st_r^x \oplus in_r^x$
$wt_{r'}^x$
$st_r^x \oplus wt_r^x$
$dn_{r'}^x$
st_r^x
$st_{r'}^x$

Fig. 13 The synthesized ESDS program. $c = client(x)$, $r = replica(x)$. x' ranges over $x.prev$, and r' ranges over $\mathscr{R}' = \mathscr{R} - \{replica(x)\}$ in $\bigotimes_{r'}$. x'' ranges over all operations such that $x \in x''.prev$, and $r'' = replica(x'')$. To avoid clutter, we omit these ranges from the $\bigotimes$ operators.

– $R_r^x \,||\, R_{r'}^{x'}$ where $x' \in x.prev$, $r' = replica(x')$, and
– $R_r^x \,||\, R_i^x$ where $r = replica(x)$, $i \in \mathscr{R}$.

We now apply Definition 37 to synthesize the ESDS program with a dynamic number of clients and replicas, shown in Figure 13. Note that we use x'' to range over all operations such that $x \in x''.prev$, and that $r'' = replica(x'')$. The guarded commands corresponding to such x'' come from the pair-process $R_{r'}^{x'}$ in Figure 10, since x' there plays the role of the operation that must be executed first ($x' \in x.prev$), and this gives the template for the case when x itself is the operation that must be executed first. The ESDS program, and the pair-program $R_r^x \,||\, R_{r'}^x$ of Figure 11 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.

6.8 Semantics of the synthesized global dynamic program

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

Definition 39 (Global state transition diagram of $\mathscr{P}$) The semantics of $\mathscr{P}$ is given by the structure $M_{\mathscr{P}} = (S^0, S, R_n, R_c)$[10] where

1. S^0 is given by Definition 37
2. S is the smallest set of 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 $\mathscr{P}$, as given by Definition 38.
4. $R_c \subseteq S \times \{\mathsf{create}\} \times S$ is a transition relation consisting of the create transitions of $\mathscr{P}$, as given by Definition 36.

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 *creation protocol* which synchronizes the "activation" of $(S^0_{ij}, P^j_i \parallel P^i_j)$ with the current computation of the existing dynamic program. The details of such a protocol are a topic for future work.

7 Soundness of the method for synthesis of global dynamic programs

Recall that a program P satisfies a correctness property expressed as a CTL* formua f iff f holds in all initial states of the global state transition diagram of P.

7.1 Projection onto subprograms

We extend the notion of path projection to global dynamic programs. The key consideration is that the J-subprogram being projected onto must "exist" in every state of the path being projected. Modify the definition of J-block given in Section 5.1 so that J-blocks include create transitions. That is, a J-block of path π is a maximal subsequence of π that starts and ends in a state and does not contain a transition by any P_i such that $i \in dom(J)$. It does contain transitions by processes other than those in $dom(J)$, as well as create transitions.

Definition 40 (Path projection in dynamic programs) Let π be a computation path of $\mathscr{P}$, i.e., π is a path in $M_{\mathscr{P}}$. Let $J \subseteq \mathsf{Pids} \times \mathsf{Pids}$ be such that $J \neq \emptyset \wedge J \subseteq I_s$ for all s along π. This restriction on J means that π can be partitioned into J-blocks. Hence, write π as $B^1 \xrightarrow{d_1} \cdots B^n \xrightarrow{d_n} B^{n+1} \cdots$ where B^n is a J-block for all $n \geq 1$. Then, the *path-projection* of π onto J, denoted $\pi{\restriction}J$, is given by:

$$\pi{\restriction}J = B^1{\restriction}J \xrightarrow{d_1} \cdots B^n{\restriction}J \xrightarrow{d_n} B^{n+1}{\restriction}J \cdots$$

Define $M_J = (S^0_J, S_J, R_J)$ to be $M_{\mathscr{P}}$ for the case when $\mathscr{P}\mathscr{S}_0 = J$, and no create transitions occur, i.e., the set of active pair-programs is always J. Recall that $M_{ij} = (S^0_{ij}, S_{ij}, R_{ij})$ is the global state transition diagram of $(S^0_{ij}, P^j_i \parallel P^i_j)$, as given by Definition 6. M_{ij} and $M_{\mathscr{P}} = (S^0, S, R_n, R_c)$ 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 $M_{\mathscr{P}}$ gives the semantics of $\mathscr{P}$. Our main soundness

[10] Since the interconnection relation I is changing dynamically, we do not use I as a subscript. We use $\mathscr{P}$ as a subscript of $M_{\mathscr{P}}$ as a reminder that $M_{\mathscr{P}}$ is the global state transition diagram of $\mathscr{P}$.

result below (the large model theorem) relates the ACTL formulae that hold in $M_{\mathscr{P}}$ to those that hold in M_{ij}.

We characterize transitions in $M_{\mathscr{P}}$ as compositions of transitions in all the relevant M_{ij}, i.e., P_i can execute a transition from configuration s to configuration t if and only if, for every $(i,j) \in I_s$, P_i^j can execute a corresponding transition from $s{\upharpoonright}ij$ to $t{\upharpoonright}ij$, and all other processes do nothing.

Lemma 4 (Transition mapping) *For all configurations $s,t \in S$ and $i \in s.procs$:*

$$s \xrightarrow{i} t \in R_n \textit{ iff } ((\forall j \in I_s(i) : s{\upharpoonright}ij \xrightarrow{i} t{\upharpoonright}ij \in R_{ij}) \textit{ and } (\forall (j,k) \in I_s, i \notin \{j,k\} : s{\upharpoonright}jk = t{\upharpoonright}jk)).$$

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 $(i,j) \in I_s$. If all such pairs permit a transition, i.e., $(\forall j \in I_s(i) : s{\upharpoonright}ij \xrightarrow{i} t{\upharpoonright}ij \in R_{ij})$, and if all pair-programs in which P_i is not a member do not execute a transition, i.e., $(\forall (j,k) \in I_s, i \notin \{j,k\} : s{\upharpoonright}jk = t{\upharpoonright}jk)$, then P_i can indeed execute the transition $s \xrightarrow{i} t$, according to the semantics of M_{ij} and $M_{\mathscr{P}}$, i.e., Definitions 6 and 39, respectively. The other direction follows by similar reasoning. The technical details of this argument follow exactly the same lines as the proof of Lemma 6.4.1 in Attie & Emerson [5]. □

Corollary 5 (Transition mapping) *For all configurations $s,t \in S$, $J \subseteq I_s$, and $i \in dom(J)$:*

$$\textit{if } s \xrightarrow{i} t \in R_n, \textit{ then } s{\upharpoonright}J \xrightarrow{i} t{\upharpoonright}J \in R_J.$$

Proof From $s \xrightarrow{i} t \in R_n$ and the forward direction of Lemma 4, we have

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

Since $J \subseteq I_s$, this implies:

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

Now apply the backward direction of Lemma 1 to the J-subprogram of $(S_I^0, P_{i_1} \parallel \ldots \parallel P_{i_K})$. Together with the above, we obtain

$$s{\upharpoonright}J \xrightarrow{i} t{\upharpoonright}J \in R_J,$$

and we are done. □

Lemma 5 (Path mapping) *Let π be a path in $M_{\mathscr{P}}$, and let $J \subseteq \mathsf{Pids} \times \mathsf{Pids}$ be such that $J \subseteq I_s$ for every configuration s along π. Then $\pi{\upharpoonright}J$ is a path in M_J.*

Proof By Defintion 40,

$$\pi{\upharpoonright}J = B^1{\upharpoonright}J \xrightarrow{d_1} \cdots B^n{\upharpoonright}J \xrightarrow{d_n} B^{n+1}{\upharpoonright}J \cdots$$

Now apply Corollary 5 to every transition $B^n{\upharpoonright}J \xrightarrow{d_n} B^{n+1}{\upharpoonright}J$ along $\pi{\upharpoonright}J$, and we conclude that every such transition is a transition in R_J. Hence $\pi{\upharpoonright}J$ is a path in M_J. Note in particular that create transitions pose no problem since they are always inside some J-block. □

In particular, when $J = \{(i,j),(j,i)\}$, Lemma 5 forms the basis for our soundness proof, since it relates computations of the synthesized program $\mathscr{P}$ to computations of the pair-programs.

Since we allow the creation of pair-programs in an arbitrary reachable state, it is possible to create several pair-programs, one after the other, so that their combined states would not be reachable if all of the pair-programs were initially present. Hence we must take as an assumption the analogue of Corollary 2, the state mapping corollary for the static case.

Definition 41 (State mapping assumption) Let s be a reachable configuration of $M_{\mathscr{P}}$, and let $J \subseteq I_s$. Then $s{\upharpoonright}J$ is a reachable state of M_J.

7.2 Deadlock-freedom of global dynamic programs

In our dynamic model, the definition of wait-for graph is essentially the same as the static case (Definition 19), except that the set of process nodes is a function of the current configuration.

Definition 42 (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 $\{P_i \mid i \in s.procs\}$, and
   (b) the arcs $\{a_i \mid i \in s.procs$ and $a_i \in P_i$ and $s{\upharpoonright}i = a_i.start\}$
2. there is an edge from P_i to every node a_i such that $a_i \in P_i$ and $s{\upharpoonright}i = a_i.start$, and
3. there is an edge from a_i to P_j if and only if $(i,j) \in I_s$ and $s{\upharpoonright}ij(a_i.guard_j) = \mathit{false}$.

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. As before, we characterize a deadlock as the occurrence in the wait-for graph of a *supercycle*:

Definition 43 (Supercycle) SC is a *supercycle* in $W(s)$ if and only if:

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

The methods for checking deadlock freedom given in [5, 9, 12] were all formulated for global static programs. Extending them to global dynamic programs is outside the scope of this paper, and is a topic for future work. We assume, in the sequel, that every reachable configuration is supercycle-free.

7.3 Liveness of global dynamic programs

To assure liveness properties of the synthesized program $\mathscr{P}$, 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 normal transition in $M_{\mathscr{P}}$ iff that transition results from executing P_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 normal transitions of $M_{\mathscr{P}}$, i.e., those that are drawn from R_n. Let π be a fullpath of $M_{\mathscr{P}}$. Recall also from Section 5.3 the meanings of $blk_i, en_i, ex_i, pnd_{ij}, aen_j$. To define versions of weak blocking fairness and weak eventuality fairness for the dynamic case, we must quantify over all processes that are created along some path π. Since processes are created dynamically as execution proceeds along π, we cannot define fairness as a PTL formula Φ for two reasons: (1) in general, Φ would be a conjunction over a countably infinite number of proccesses, and so would be infinitely long, and (2) the truth values of the atomic propositions AP_i of a process P_i that is created by some create transition $s \overset{\mathsf{create}}{\rightarrow} t$ along π, are not well-defined before s. Hence the truth value of a conjunction over all processes created along π would not, in general, be well-defined in any particular configuration of π. We deal with this by redefining fairness as a predicate over paths. An "initial" version of the predicate handles all processes and pair-programs that are alive in the first configuration of π. We then quantify this initial version over all suffixes of π. We use this approach for both weak blocking fairness and weak eventuality fairness. Recall that $\models$ for fullpaths and path-formulae is defined in Section 2.2.

Definition 44 (Dynamic weak blocking fairness Φ_b) Let π be a fullpath in $M_{\mathscr{P}}$, and let s be the first configuration along π. Then define the predicate Φ_b^{init} over fullpaths:

$$\Phi_b^{init}(\pi) \text{ holds iff } \pi \models \textstyle\bigwedge_{i \in s.procs} \overset{\infty}{\mathsf{G}}(blk_i \wedge en_i) \Rightarrow \overset{\infty}{\mathsf{F}} ex_i.$$

That is, processes that are initially alive are treated fairly. Also define the predicate Φ_b:

$$\Phi_b(\pi) \text{ holds iff for every suffix } \rho \text{ of } \pi\text{: } \Phi_b^{init}(\rho)$$

That is, once a process is created, it is subsequently treated fairly.

This gives the dynamic analogue of Definition 23, weak blocking fairness for global static programs.

Definition 45 (Dynamic weak eventuality fairness, Φ_ℓ) Let π be a fullpath in $M_{\mathscr{P}}$, and let s be the first configuration along π. Then define the predicate Φ_ℓ^{init} over fullpaths:

$$\Phi_\ell^{init}(\pi) \text{ holds iff } \pi \models \textstyle\bigwedge_{(i,j) \in I_s} (\overset{\infty}{\mathsf{G}} en_i \vee \overset{\infty}{\mathsf{G}} en_j) \wedge \overset{\infty}{\mathsf{G}} pnd_{ij} \Rightarrow \overset{\infty}{\mathsf{F}}(ex_i \vee ex_j).$$

That is, pair-programs that are initially alive are treated fairly. Also define the predicate Φ_ℓ:

$$\Phi_\ell(\pi) \text{ holds iff for every suffix } \rho \text{ of } \pi\text{: } \Phi_\ell^{init}(\rho)$$

That is, once a pair-program is created, it is subsequently treated fairly.

This gives the dynamic analogue of Definition 25, weak eventuality fairness for global static programs. Finally, a fullpath π satisfies *creation fairness* iff it contains an infinite number of normal transitions:

Definition 46 (Creation fairness, Φ_c) $\Phi_c \overset{\text{df}}{=} \overset{\infty}{\mathsf{F}} normal.$

Our overall fairness notion Φ is thus the conjunction of dynamic weak blocking fairness, dynamic weak eventuality fairness, and creation fairness:

Definition 47 (Global dynamic fairness, Φ) We define the predicate Φ over fullpaths as follows:

$$\Phi(\pi) \text{ holds iff } \Phi_b(\pi) \text{ and } \Phi_\ell(\pi) \text{ and } M_{\mathscr{P}}, \pi \models \Phi_c.$$

Hence a fullpath π is fair iff π contains an infinite number of normal transitions, and every pair-program that is created along π is, from the point of its creation, treated fairly according to weak blocking and weak eventuality fairness.

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 wait-for 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, in configuration s, block the pair-program $(S_{ij}^0, P_i^j \parallel P_j^i)$ from executing some arc of P_i^j or P_j^i.

Definition 48 (Liveness condition for global dynamic programs) The liveness condition for global dynamic programs holds if and only if, for every reachable configuration s of $M_{\mathscr{P}}$, both the following conditions hold:

1. For every $(i,j) \in I_s$: $M_{ij}, S^0_{ij} \models \mathsf{AGA}(\mathsf{G}ex_i \Rightarrow \overset{\infty}{\mathsf{G}}aen_j)$
2. For every $(i,j) \in I_s$ such that $s \models pnd_{ij}$,
   there exists a finite $W \subseteq \mathsf{Pids}$ such that,
   for all t reachable from s along a finite path ρ such that
   (a) pnd_{ij} holds in all configurations of ρ, and
   (b) neither P_i nor P_j are executed along ρ,
   it must be that $Wt_{ij}(t) \subseteq W$.

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. It is the dynamic analogue of the liveness condition for the static case. The second condition is "global": it requires, in effect, that a process is not forever delayed because new processes which block it are constantly being created.

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 in M_{ij}. Recall that $\models_\Phi$ is the satisfaction relation of CTL* when the path quantifiers A and E are restricted to fullpaths π such that $\Phi(\pi)$.

Lemma 6 (Progress for global dynamic programs) *Let $\mathscr{D}$ be a global dynamic specification, let $\mathscr{P}$ be a global dynamic program synthesized by our method from $\mathscr{D}$ (Def. 37) and let $M_{\mathscr{P}}$ be the global state transition diagram of $\mathscr{P}$ (Def. 39). Let s be an arbitrary reachable configuration of $M_{\mathscr{P}}$ and let $(i,j) \in I_s$. If the following assumptions all hold*

1. *the state mapping assumption (Def. 41),*
2. *the dynamic liveness condition (Def. 48),*
3. *for every reachable configuration u of $M_{\mathscr{P}}$, $W(u)$ is supercycle-free, and*
4. *$M_{ij}, s{\upharpoonright}ij \models \neg h_{ij} \wedge \mathsf{AF}h_{ij}$ for some $h_{ij} \in CL(spec_{ij})$,*

then

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

Proof By Assumption 3 above and Proposition 1 (deadlock-freedom), in every reachable configuration, there is some process with an enabled arc. Hence every fullpath in $M_{\mathscr{P}}$ is infinite. Let π be an arbitrary Φ-fair fullpath starting in s. If $M_{\mathscr{P}}, \pi \models \mathsf{F}(ex_i \vee ex_j)$, then we are done. Hence we assume

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

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 48), $Wt_{ij}(t) \subseteq W$ for some finite $W \subseteq \mathsf{Pids}$. Hence, there 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.

Let P_J be the static concurrent program with initial state set $\{v{\upharpoonright}J\}$, i.e., a single initial state, $v{\upharpoonright}J$, and the interconnection relation $J = I_v \cap \{(k,l) \mid \{k,l\} \cap Wt_{ij}(v) \neq \emptyset\}$. That is, the pair-programs in P_J are those that (1) exist in configuration v, and (2) have some component process which is in a wait-for path of $W(v)$ that starts from P_i or P_j, i.e., some process that blocks P_i or P_j. Since v is a reachable configuration of $M_{\mathscr{P}}$, we have from Assumption 1 that $v{\upharpoonright}J$ is a reachable J-state of M_J. By applying Lemma 3 (progress for global static programs) to P_J, we conclude

$$M_J, v{\upharpoonright}J \models_{\Phi} \mathsf{AF}(ex_i \vee ex_j). \tag{b}$$

Now let $\rho_J = \pi^v{\upharpoonright}J$, where π^v is the infinite suffix of π starting in v. From Lemma 5, ρ_J is a path in M_J. We now establish

$$\rho_J \text{ is a fullpath in } M_J \tag{c}$$

given the assumption that (a) holds. From (a) and weak eventuality fairness (Definition 45), we see that $Wt_{ij}(w)$ is nonempty for infinitely many configurations w along π, since otherwise one of P_i, P_j would be executed. By definition, there is no wait-for path in $W(t)$ from a process in $Wt_{ij}(t)$ to a process outside $Wt_{ij}(t)$. Hence, by Assumption 3 and Proposition 1 (deadlock-freedom), there exists some $P_k \in Wt_{ij}(t)$ such that P_k has an enabled arc in configuration t. Since this holds for all configurations t along π, we conclude by weak blocking fairness (Definition 44) and creation fairness (Definition 46), that infinitely often along π, some process in $Wt_{ij}(v)$ is executed. Hence, by Definition 40 (path projection) and the definition of J, ρ_J is infinite. Hence, ρ_J is a fullpath in M_J, and (c) is established.

By Definition 40, the first state of ρ_J is $v{\upharpoonright}J$. Hence, by (b) above, we have $\rho_J \models \mathsf{F}(ex_i \vee ex_j)$. From $\rho_J = \pi^v{\upharpoonright}J$ and Definition 40, 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. $\square$

7.4 The large model theorem for dynamic programs

The main technical result for our dynamic synthesis method is that it is sound. This is given by the large model theorem, which 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 global dynamic program $\mathscr{P}$ that our method synthesizes.

Theorem 2 (Large model) *Let $\mathscr{D}$ be a global dynamic specification, let $\mathscr{P}$ be the global dynamic program synthesized by our method from $\mathscr{D}$ (Def. 37) and let $M_{\mathscr{P}}$ be the global state transition diagram of $\mathscr{P}$ (Def. 39). Let s be an arbitrary reachable configuration of $M_{\mathscr{P}}$ and let $(i,j) \in I_s$, so that $(\{i,j\}, spec_{ij}) \in s.\mathscr{I}$. If all the following assumptions hold*

1. *the state mapping assumption (Def. 41),*
2. *the dynamic liveness condition (Def. 48),*
3. *for every reachable configuration u of $M_{\mathscr{P}}$, $W(u)$ is supercycle-free, and*
4. *$M_{ij}, s{\upharpoonright}ij \models f_{ij}$ for some $f_{ij} \in CL(spec_{ij})$,*

then

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

Proof We first overview the proof, which is quite similar to the proof of Theorem 1, the large model theorem for the static case. The only difference in the proof is in dealing with create transitions. This is straightforward, since $(S^0_{ij}, P^j_i \parallel P^i_j)$ is created with its current state set to one of its reachable states, and so the same projection relationships hold between $M_{\mathscr{P}}$ and M_{ij} in the dynamic case as between M_I and M_{ij} in the static case. In particular, Lemma 5 provides the exact dynamic analogue for Lemma 2, 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 $(S^0_{ij}, P^j_i \parallel P^i_j)$ is created. Since we do not require computation paths to start from an initial state, this does not pose a problem.

Now for the formal proof, which is by induction on the structure of f_{ij}. Recall that f_{ij} is a formula of ACTL^-_{ij}. Throughout, let $s_{ij} = s{\upharpoonright}ij$.

$f_{ij} = p_i$, or $f_{ij} = \neg p_i$, where $p_i \in AP_i$, i.e., p_i is an atomic proposition.
By definition of ${\upharpoonright}ij$, s and $s{\upharpoonright}ij$ agree on all atomic propositions in $AP_i \cup AP_j$. 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 $M_{\mathscr{P}}, s \models_{\Phi} g_{ij}$ and $M_{\mathscr{P}}, s \models_{\Phi} h_{ij}$. So by CTL* semantics we get $M_{\mathscr{P}}, s \models_{\Phi} (g_{ij} \wedge h_{ij})$.

$f_{ij} = g_{ij} \vee h_{ij}$. 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 $M_{\mathscr{P}}, s \models_{\Phi} g_{ij}$ or $M_{\mathscr{P}}, s \models_{\Phi} h_{ij}$. So by CTL* semantics we get $M_{\mathscr{P}}, s \models_{\Phi} (g_{ij} \vee h_{ij})$.

$f_{ij} = \mathsf{A}[g_{ij} \mathsf{W} h_{ij}]$. Let π be an arbitrary Φ-fair fullpath starting in s. We establish $\pi \models [g_{ij} \mathsf{W} h_{ij}]$. By Definition 40 (path projection), $\pi{\upharpoonright}ij$ starts in $s{\upharpoonright}ij = s_{ij}$. Hence, by CTL semantics, $\pi{\upharpoonright}ij \models [g_{ij} \mathsf{W} h_{ij}]$ (note that this holds even if $\pi{\upharpoonright}ij$ is not a fullpath, i.e., is a finite path). Since $f_{ij} \in CL(spec_{ij})$, we have $g_{ij} \in CL(spec_{ij})$ and $h_{ij} \in CL(spec_{ij})$. We have two cases.

Case 1: $\pi{\upharpoonright}ij \models \mathsf{G} g_{ij}$. Let t be an arbitrary state along π. By Definition 40, $t{\upharpoonright}ij$ 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 $\pi \models [g_{ij} \mathsf{W} h_{ij}]$ by CTL*semantics.

Case 2: $\pi{\upharpoonright}ij \models [g_{ij} \mathsf{U} h_{ij}]$. Let $s^{m'}_{ij}$ be the first state along $\pi{\upharpoonright}ij$ that satisfies h_{ij}[11]. By Definition 40, there exists at least one state t along π such that $t{\upharpoonright}ij = s^{m'}_{ij}$. 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 40, $s^n{\upharpoonright}ij$ lies along the portion of $\pi{\upharpoonright}ij$ up to, and possibly including, $s^{m'}_{ij}$. That is, $s^n{\upharpoonright}ij = s^m_{ij}$, where $0 \le m \le m'$. Now suppose $s^n{\upharpoonright}ij = s^{m'}_{ij}$ (i.e., $m = m'$). Then, by $s^{m'}_{ij} \models h_{ij}$ 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 $s^{m'}_{ij}$ 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 $s^m_{ij} \models g_{ij}$ by CTL*semantics. From $s^n{\upharpoonright}ij = s^m_{ij}$ 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 $\pi \models [g_{ij} \mathsf{W} h_{ij}]$ by CTL*semantics.

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

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

$f_{ij} = \mathsf{A}[g_{ij} \mathsf{U} h_{ij}]$. 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. 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 $s_{ij} \models \neg h_{ij} \land \mathsf{AF} h_{ij}$. Let π be an arbitrary Φ-fair fullpath starting in s. By Proposition 1 (deadlock-freedom), 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 40, $\rho \upharpoonright ij$ is a path from s_{ij} to $t \upharpoonright ij$. By Lemma 5 (path mapping), $\rho \upharpoonright ij$ is a path in M_{ij}. Suppose $\rho \upharpoonright ij$ contains a state u_{ij} such that $u_{ij} \models h_{ij}$. By Definition 40, 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 $t \upharpoonright ij \models \neg h_{ij} \land \mathsf{AF} h_{ij}$ by CTL semantics. Let π' be the suffix of π starting in t. Since $t \upharpoonright ij \models \neg h_{ij} \land \mathsf{AF} h_{ij}$ and $h_{ij} \in CL(spec_{ij})$, we can apply Lemma 6 (progress) to conclude $M_{\mathscr{P}}, t \models_\Phi \mathsf{AF}(ex_i \lor ex_j)$. Since t is an arbitrary state along π, we conclude $M_{\mathscr{P}}, \pi \models \overset{\infty}{\mathsf{F}}(ex_i \lor ex_j)$. Hence, by Definition 40, $\pi \upharpoonright ij$ is a fullpath, since it contains an infinite number of P_i or P_j transitions. By Lemma 5, $\pi \upharpoonright ij$ 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 40, 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 $s_{ij} \models \mathsf{A}[g_{ij} \mathsf{W} h_{ij}]$. From the above proof case for $\mathsf{A}[g_{ij} \mathsf{W} h_{ij}]$, we have $s \models_\Phi \mathsf{A}[g_{ij} \mathsf{W} h_{ij}]$. Hence $\pi \models_\Phi [g_{ij} \mathsf{W} h_{ij}]$, 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 $s \models_\Phi \mathsf{A}[g_{ij} \mathsf{U} h_{ij}]$. □

To establish similar corollaries for the Large Model Theorem in the dynamic case, we need to restrict the pair specifications $spec_{ij}$ to be of the form $\mathsf{AG} f_{ij}$. This is because dynamically added pair-programs may start executing in any reachable state, rather than just an initial state. For this same reason, we state the corollaries in terms of an arbitrary reachable configuration, rather than an initial configuration. Finally, note that, if the liveness condition does not hold, then we can still verify safety properties of $\mathscr{P}$, since the cases for atomic propositions, $g_{ij} \land h_{ij}$, $g_{ij} \lor h_{ij}$, and $\mathsf{A}[g_{ij} \mathsf{W} h_{ij}]$ above still apply.

Corollary 6 (Large model) *Let $\mathscr{D}$ be a global dynamic specification, let $\mathscr{P}$ be the global dynamic program synthesized by our method from $\mathscr{D}$ (Def. 37), so that, for all reachable configurations v of $M_{\mathscr{P}}$, $(\forall (i,j) \in I_v : M_{ij}, S^0_{ij} \models spec_{ij})$, where $spec_{ij}$ is an ACTL^-_{ij} formula of the form $\mathsf{AG} f_{ij}$, and $M_{\mathscr{P}}$ is the global state transition diagram of $\mathscr{P}$ (Def. 39). Let s be an arbitrary reachable configuration of $M_{\mathscr{P}}$. If the following assumptions all hold*

1. *the state mapping assumption (Def. 41),*
2. *the dynamic liveness condition (Def. 48),*
3. *for every reachable configuration u of $M_{\mathscr{P}}$, $W(u)$ is supercycle-free*

then

$$M_{\mathscr{P}}, s \models_\Phi \textstyle\bigwedge^s_{ij} spec_{ij}.$$

Proof Let (i,j) be an arbitrary pair in I_s. Since s is a reachable configuration in $M_{\mathscr{P}}$, it follows by Assumption 1 that $s \upharpoonright ij$ is a reachable state in M_{ij}. Since $M_{ij}, S^0_{ij} \models spec_{ij}$ and $spec_{ij} = \mathsf{AG} f_{ij}$, we have, by CTL semantics, that $M_{ij}, s \upharpoonright ij \models spec_{ij}$. By applying Theorem 2, we obtain $M_{\mathscr{P}}, s \models_\Phi (\bigwedge_{(i,j) \in I_s} spec_{ij})$, since (i,j) is an arbitrarily chosen pair in I_s. But this is the same as $M_{\mathscr{P}}, s \models_\Phi \bigwedge^s_{ij} spec_{ij}$. □

Corollary 7 (Large model) *Let $\mathscr{D}$ be a global dynamic specification, let $\mathscr{P}$ be the global dynamic program synthesized by our method from $\mathscr{D}$ (Def. 37), so that, for all reachable configurations v of $M_{\mathscr{P}}$, $(\forall (i,j) \in I_v : M_{ij}, S^0_{ij} \models spec_{ij})$, where $spec_{ij}$ is an ACTL^-_{ij} formula of the form $\mathsf{AG} f_{ij}$, and $M_{\mathscr{P}}$ is the global state transition diagram of $\mathscr{P}$ (Def. 39). Let s be an arbitrary reachable configuration of $M_{\mathscr{P}}$. If the following assumptions all hold*

1. *the state mapping assumption (Def. 41),*
2. *the dynamic liveness condition (Def. 48),*
3. *for every reachable configuration u of $M_{\mathscr{P}}$, $W(u)$ is supercycle-free*
4. *$(\bigwedge^s_{ij} spec_{ij}) \vdash_{\mathrm{CTL}} glob{-}spec$*

then

$$M_{\mathscr{P}}, s \models_{\Phi} glob{-}spec.$$

Proof By Corollary 6, we have $M_{\mathscr{P}}, s \models_{\Phi} \bigwedge^s_{ij} spec_{ij}$. From $(\bigwedge^s_{ij} spec_{ij}) \vdash_{\mathrm{CTL}} glob{-}spec$ and soundness of the CTL deductive system, any model of $\bigwedge^s_{ij} spec_{ij}$ is also a model of $glob{-}spec$. Hence $M_{\mathscr{P}}, s \models_{\Phi} glob{-}spec$. $\square$

7.5 Correctenss of the synthesized ESDS global dynamic program

Correctness of the ESDS program (Figure 13) follows immediately from Theorem 2 and Corollary 6, since all specified properties are over pairs, and so no global proof (as in Figure 8) is needed. Local structure 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 Attie and Emerson [5], and the proof given there can be adapted to the setting in this paper.

8 Discussion and further work

In addition to avoiding state-explosion in the number of processes, another major benefit of pairwise form is that it is modular and compositional, since the code expressing the interaction of each pair of processes is isolated from the remaining code. This provides locality and modifiability [43]. Suppose we have a global program P that satsifies $glob{-}spec$, and we wish to modify P so that it also satisfies a new global property f, i.e., we want a new program P' that satisfies $glob{-}spec \wedge f$. We determine a set of pair-properties $\{f_{ij} \mid (i,j) \in \varphi\}$, where $\varphi \subseteq I$, such that $\bigwedge_{(i,j)\in\varphi} f_{ij} \vdash_{\mathrm{CTL}} f$. We then modify the pair-programs $P^j_i \parallel P^i_j$ for all $(i,j) \in \varphi$ so that they satisfy f_{ij}, in addition to the previous pair-specification $spec_{ij}$. We now synthesize concurrent program P' from the modified ($(i,j) \in \varphi$) and unmodified ($(i,j) \notin \varphi$) pair-programs. P' now satisfies $glob{-}spec \wedge f$, by Corollary 4 or 7, depending on whether P is static or dynamic. Futhermore, the needed modification of P was effected by modifying only the pair-programs whose pair-properties contributed directly to the satisfaction of f, via $\bigwedge_{(i,j)\in\varphi} f_{ij} \vdash_{\mathrm{CTL}} f$. Modification of the pair-programs $P^j_i \parallel P^i_j$ can be effected by re-synthesizing them [6,11,30], using $spec_{ij} \wedge f_{ij}$ as the specification, or by repairing them with respect to f_{ij}, while maintaining $spec_{ij}$ [10,20]. Exploration of this aspect of pairwise form is a topic for future work.

Pairwise synthesis as presented in this paper is restricted to finite-state pair-programs, and to the global programs (static and dynamic) that result from composing these. A key issue is the applicability to realistic concurrent programs, which in practice, contain data such as 32-bit integers (bounded but large state space, i.e., $2^{31}-1$ to -2^{31}), arrays (a given instance has fixed size, but no bound on the size of new instances at creation time), and pointer-based structures (no bound on the size of an instance at any time). Such data variables can occur in the specifications of concurrent programs, and can therefore be used to make decisions that affect the interaction of the component processes, e.g., when to request a critical section, and when to release it. We first note that the large model theorems do not rely *per se* on the finite-state assumption; the execution projection arguments that are used in establishing the large model theorems work equally well for infinite-state systems cf. [45]. Likewise, the definitions of wait-for-graph and supercycle freedom are also applicable to infinite-state systems [9]. The finite-state assumption is used only to simplify the model of concurrency (finite-state synchronization skeletons) and to enable the proof obligations of the large model theorems (pair-structure satisfies pair specification, liveness condition, supercycle-freedom condition) to be discharged using model checking. Hence, generalization to infinite state concurrent programs can be achieved by replacing (propositional) CTL by first-order CTL [29], and replacing the model-checking of pair-properties by more general verification methods, using first-order verification obligations [3,51,55]. We leave this as a topic for future work. While first-order verification obligations are semi-decidable, in general, the use of pairwise synthesis may still be beneficial, since it removes the universal quantifier "for all processes ..." that is typically used in stating global properties of concurrent porgrams, and also reduces their size and complexity. This may make first-order pairwise proof obligations easier to check, using e.g., theorem provers and SMT solvers. Extension to a first order setting is another topic for future work.

A third important topic of future work is to show how the synthesized programs can be efficiently implemented using realistic primitives. Attie and Emerson [5] show how global static programs can be syntactically transformed (by introducing "pair-controller processes") into programs that use *multiparty interactions* for process comunication and synchronization. Efficient implementations of multiparty interactions (using point to point message passing) are provided by the BIP framework and toolset [15, 16], where example implementations compare favorably (in efficiency) with low-level implementations. Furthermore, in our synthesized programs, a process P_i atomically inspects and updates the shared variables x_{ij} ("pairwise shared state") that it shares with all of its neighbors P_j, and so the degree of atomicity required is the degree of the interconnection relation I, viewed as a graph. When I has small degree, this confers a "spatial locality" property, which is helpfull in designing efficient implementations.

Another issue is how to seamlessly integrate dynamically created pair-programs into an existing global dynamic program. We currently assume that a dynamically created pair-program $(S^0_{ij}, P^j_i \parallel P^i_j)$ is added in a configuration s such that $s{\restriction}ij$ is a reachable state of $(S^0_{ij}, P^j_i \parallel P^i_j)$ (state mapping assumption, Def. 41). How to ensure that this assumption holds, and how then to actually integrate, on-the-fly, $(S^0_{ij}, P^j_i \parallel P^i_j)$ into the global dynamic program, are topics for future work. This issue has implications for the semantics of dynamic process creation in general, cf. Attie and Lynch [7].

We showed above how to take an infinite-state system, the ESDS system, and to decompose it into an infinite set of finite-state processes, that are added dynamically, as needed. Our ESDS example above shows this in a single case, but does not give a general method. Development of this idea could provide a new way to synthesize and verify infinite-state concurrent programs, which would, in some cases, obviate the use of first-order proof obligations, as discussed above.

We have not handled fault-tolerance in this paper. Integration of the work here with that of [11] would enable synthesis of fault-tolerant concurrent programs in pairwise form.

In [11], we model the faults to be tolerated as a set of *fault actions*, which perturb the state, and possibly cause the program to enter a state this is unreachable under normal (fault-free) operation. To ensure fault-tolerance, we have to extend the large model theorem to apply to these "perturbed states". If faults perturb the state of at most two processes at a time, this is straightforward, since we can add the fault actions to the relevant pair-structures, and then proceed as usual. If faults perturb the state of more than two processes, then the problem is more challenging. A possible attack is to decompose the given set of fault actions into a set of "pairwise fault actions" that do perturb the state of at most two processes at a time, and to show that the pairwise faults generate the same set of perturbed states as the original faults. We leave this intriguing problem as future work. A boundary case of this model of fault tolerance is that of *self stabilization*, where the set of perturbed states consists of every possible state of the program. In this case, we must design a set of pairwise fault actions which can generate any possible state as a perturbed state.

Finally, remaining topics for future work are those mentioned above for dynamic synthesis: checking deadlock freedom and expressiveness of pairwise form in the dynamic case.

9 Conclusions

We presented two synthesis methods, that produce respectively static concurrent programs (fixed set of component processes) and dynamic concurrent programs (processes can be added dynamically). Our methods do not incur the exponential overhead due to state-explosion, apply to any process interconnection scheme, do not make any assumption of similarity among the component processes, and preserve all pairwise correctness properties expressed as nexttime-free formulae of ACTL, the universal fragment of CTL. Furthermore, we show in [8] that no loss of expressiveness is incurred in the static case: any static finite-state shared-memory concurrent program can be rewritten (up to strong bisimulation) in pairwise form. We also showed how to use CTL deduction to increase the set of global specifications that can be dealt with. To the best of our knowledge, our method is the first that deals with the synthesis of dynamic concurrent programs, in which processes can be created at run time.

A Glossary of major symbols

$\models$	Satisfies relation of CTL*, *Sec.* 2.2
$\models_\Phi$	Satisfies relation of CTL* relativized to fairness notion Φ, *Sec.* 2.3
Φ	CTL* path formula that specifies fairness, *Sec.* 2.3
$\otimes, \bigotimes_{j \in I(i)}$	"Conjunctive" guarded-command composition operator, *Def.* 14
$\oplus, \bigoplus_{\ell \in [1:n]}$	"Disjunctive" guarded-command composition operator, *Def.* 14
$\{\!\mid\}$	State to formula operator, *Def.* 21
$\mathsf{\Lambda}_{ij}$	Static spatial modality, *Def.* 9
$\mathsf{\Lambda}^s_{ij}$	Dynamic spatial modality, *Def.* 33
AP_i	The set of atomic propositions of process i, *Sec.* 2.1
AP	The set of all atomic propositions, *Sec.* 2.2
$\upharpoonright i$	State projection onto process i, *Def.* 5, *Def.* 34
$\upharpoonright SH_{ij}$	State projection onto the shared variables SH_{ij}, *Def.* 5
$\upharpoonright ij$	State or path projection onto pair-program $(S^0_{ij}, P^j_i \parallel P^i_j)$, *Def.* 12, *Def.* 34
$\upharpoonright J$	State or path projection onto a J-subprogram, *Def.* 12, *Def.* 18, *Def.* 34, *Def.* 40
PS	Pair-specification, *Def.* 3

P_i^j	Pair-process that represents process i in the pair-program consisting of processes i and j, *Def.* 4
a_i^j	Arc of process P_i^j, *Sec.* 5.3
s_i, t_i	Local state of process P_i^j or process P_i, *Sec.* 2.1
$(S_{ij}^0, P_i^j \parallel P_j^i)$	Pair-program consisting of processes i and j, *Def.* 4
SH_{ij}	Shared variables of $(S_{ij}^0, P_i^j \parallel P_j^i)$, *Def.* 4
S_{ij}^0	The set of initial states of $(S_{ij}^0, P_i^j \parallel P_j^i)$, *Def.* 4
S_{ij}	The set of states of $(S_{ij}^0, P_i^j \parallel P_j^i)$, *Def.* 6
R_{ij}	The transition relation of $(S_{ij}^0, P_i^j \parallel P_j^i)$, *Def.* 6
M_{ij}	Pair-structure of $(S_{ij}^0, P_i^j \parallel P_j^i)$, *Def.* 6
$\mathscr{I}$	Global static specification, *Def.* 7
$\mathscr{I}.pairs$	The pairs in $\mathscr{I}$, *Def.* 7
I	Interconnection relation, *Def.* 8
$dom(I)$	Domain of I, *Def.* 8
J	Subrelation of I; gives a subprogram of $(S_I^0, P_{i_1} \parallel \ldots \parallel P_{i_K})$, *Def.* 11
$dom(J)$	Domain of J, *Sec.* 5.1
$(S_I^0, P_{i_1} \parallel \ldots \parallel P_{i_K})$	The global static program synthesized from $\mathscr{I}$, *Def.* 14
P_i	Process i in $(S_I^0, P_{i_1} \parallel \ldots \parallel P_{i_K})$, *Def.* 14
a_i	Arc in process P_i, *Sec.* 5.2
s	I-state, *Def.* 11
S_I^0	The set of initial states of $(S_I^0, P_{i_1} \parallel \ldots \parallel P_{i_K})$, *Def.* 14
S_I	The set of states of $(S_I^0, P_{i_1} \parallel \ldots \parallel P_{i_K})$, *Def.* 15
R_I	The transition relation of $(S_I^0, P_{i_1} \parallel \ldots \parallel P_{i_K})$, *Def.* 15
M_I	I-structure of $(S_I^0, P_{i_1} \parallel \ldots \parallel P_{i_K})$, *Def.* 15
$W_I(s)$	The wait-for-graph for $(S_I^0, P_{i_1} \parallel \ldots \parallel P_{i_K})$ in I-state s, *Def.* 19
$\mathscr{D}$	Global dynamic specification, *Def.* 28
$\mathscr{PS}$	Universal set of pair-specifications, *Def.* 28
$\mathscr{PS}_0$	Initial set of pair-specifications, *Def.* 28
create	Create mapping, *Def.* 28
$\mathscr{I}.pairs$	The pairs in $\mathscr{I}$, *Def.* 7
s	Configuration, *Def.* 31
$s.\mathscr{I}$	Pair-specifications in configuration s, *Def.* 31
$s.\mathscr{A}$	Pair-programs in configuration s, *Def.* 31
$s.\mathscr{S}$	State-mapping of configuration s, *Def.* 31
$s.procs$	Processes in configuration s, *Def.* 31
$s.pairs$	Pairs in configuration s, *Def.* 31
I_s	Dynamic interconnection relation for configuration s, *Def.* 32
$\mathscr{P}$	The global dynamic program synthesized from $\mathscr{D}$, *Def.* 37
P_i	Process i in $\mathscr{P}$, *Def.* 37
a_i	Arc in process P_i, *Sec.* 7.2
S^0	The set of initial states of $\mathscr{P}$, *Def.* 37
S	The set of states of $\mathscr{P}$, *Def.* 39
R_n	The normal transitions of $\mathscr{P}$, *Def.* 39
R_c	The create transitions of $\mathscr{P}$, *Def.* 39
$M_{\mathscr{P}}$	Structure (transition diagram) of $\mathscr{P}$, *Def.* 39
$W(s)$	The wait-for-graph for $\mathscr{P}$ in configuration s, *Def.* 42

References

1. Shaull Almagor and Orna Kupferman. Latticed-ltl synthesis in the presence of noisy inputs. In Anca Muscholl, editor, *Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8412 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2014.

2. 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.
3. K. Apt and E. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 1997.
4. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, 2001.
5. P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Transactions on Programming Languages and Systems*, 20(1):51–115, January 1998.
6. P. C. Attie and E. A. Emerson. Synthesis of concurrent systems for an atomic read/write model of computation. *ACM Transactions on Programming Languages and Systems*, 23(2):187–242, March 2001. Extended abstract appears in Proceedings of the 15'th ACM Symposium of Principles of Distributed Computing (PODC), Philadelphia, May 1996, pp. 111–120.
7. P. C. Attie and N.A. Lynch. Dynamic input/output automata: a formal and compositional model for dynamic systems. *Information and Computation*, 2016.
8. Paul C. Attie. Finite-state concurrent programs can be expressed in pairwise normal form. *Theor. Comput. Sci.*, 619:1–31, 2016.
9. Paul C. Attie, Saddek Bensalem, Marius Bozga, Mohamad Jaber, Joseph Sifakis, and Fadi A. Zaraket. An abstract framework for deadlock prevention in BIP. In Dirk Beyer and Michele Boreale, editors, *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, volume 7892 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2013.
10. Paul C. Attie, Ali Cherri, Kinan Dak Al Bab, Mohamad Sakr, and Jad Saklawi. Model and program repair via SAT solving. In *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*, pages 148–157. IEEE, 2015. Tool download available at `http://eshmuntool.blogspot.com/`.
11. P.C. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems*, 26(1):125–185, January 2004. Extended abstract appears in Proceedings of the 17'th ACM Symposium of Principles of Distributed Computing (PODC), Puerto Vallarta, Mexico, pp. 173–182.
12. P.C. Attie and H. Chockler. Efficiently verifiable conditions for deadlock-freedom of large concurrent programs. In *Proceedings of VMCAI 2005: Verification, Model Checking and Abstract Interpretation*, Paris, France, January 2005.
13. Guy Avni and Orna Kupferman. Synthesis from component libraries with costs. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, volume 8704 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2014.
14. Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.
15. Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. From high-level component-based models to distributed implementations. In Luca P. Carloni and Stavros Tripakis, editors, *EMSOFT*, pages 209–218. ACM, 2010.
16. Borzoo Bonakdarpour, Marius Bozga, Mohamad Jaber, Jean Quilbeuf, and Joseph Sifakis. A framework for automated distributed implementation of component-based models. *Distributed Computing*, 25(5):383–409, 2012.
17. Borzoo Bonakdarpour, SandeepS. Kulkarni, and Fuad Abujarad. Symbolic synthesis of masking fault-tolerant distributed programs. *Distributed Computing*, 25(1):83–108, 2012.
18. M.C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.
19. F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing model checking in verification by AI techniques. *Artif. Intell.*, 1999.
20. George Chatzieleftheriou, Borzoo Bonakdarpour, Panagiotis Katsaros, and Scott A. Smolka. Abstract model repair. *Logical Methods in Computer Science*, 11(3), 2015.
21. 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.
22. 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.

23. Leonardo Mendonça de Moura, Sam Owre, Harald Rue, John M. Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer, 2004.
24. Xianghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 442–452, New York, NY, USA, 2002. ACM.
25. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.
26. E. W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*, pages 188–199. Springer-Verlag, New York, 1982.
27. Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.
28. 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.
29. 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.
30. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241 – 266, 1982.
31. E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *Conference on Automated Deduction*, pages 236–254, 2000.
32. E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8:275–306, 1987.
33. E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *CAV*, pages 87–98, 1996.
34. Fathiyeh Faghih and Borzoo Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. In Pascal Felber and Vijay K. Garg, editors, *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014, Paderborn, Germany, September 28 - October 1, 2014. Proceedings*, volume 8756 of *Lecture Notes in Computer Science*, pages 165–179. Springer, 2014.
35. 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.
36. Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.
37. Adria Gascón and Ashish Tiwari. Synthesis of a simple self-stabilizing system. In Krishnendu Chatterjee, Rüdiger Ehlers, and Susmit Jha, editors, *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014.*, volume 157 of *EPTCS*, pages 5–16, 2014.
38. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, May 1994.
39. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 583, 1969.
40. 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. Springer-Verlag, 2000.
41. O. Kupferman and M.Y. Vardi. Synthesis with incomplete information. In *2nd International Conference on Temporal Logic*, pages 91–106, Manchester, July 1997.
42. 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.
43. B. Liskov. *Program Development in Java*. Addison Wesley, 2001.
44. Yoad Lustig and Moshe Y. Vardi. Synthesis from component libraries. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FOSSACS '09, pages 395–409, Berlin, Heidelberg, 2009. Springer-Verlag.
45. N. A. Lynch. *Distribiuted Algorithms*. Morgan Kaufmann, 1996.
46. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984. Also appears in Proceedings of the Workshop on Logics of Programs, Yorktown-Heights, N.Y., Springer-Verlag Lecture Notes in Computer Science vol. 131 (1981).
47. 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.

48. 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.
49. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, 2001.
50. Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan, October 22-25, 2007, Proceedings*, volume 4762 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2007.
51. F.B. Schneider. *Verification of Sequential and Concurrent Programs*. Springer, 1997.
52. 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.
53. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
54. Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 184–193, New York, NY, USA, 1986. ACM.
55. Z.Manna and A. Pnueli. *Temporal Verification of Reactive Systems—Safety*. Springer, 1995.