Lazy Compositional Verification*

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (415) 859-5272 Fax: +1 (415) 859-2844

Abstract. Existing methodologies for the verification of concurrent systems are effective for reasoning about global properties of small systems. For large systems, these approaches become expensive both in terms of computational and human effort. A *compositional* verification methodology can reduce the verification effort by allowing global system properties to be derived from local component properties. For this to work, each component must be viewed as an open system interacting with a well-behaved environment. Much of the emphasis in compositional verification has been on the *assume-guarantee* paradigm where component properties are verified contingent on properties that are assumed of the environment. We highlight an alternate paradigm called *lazy composition* where the component properties are proved by composing the component with an abstract environment. We present the main ideas underlying lazy composition along with illustrative examples, and contrast it with the assume-guarantee approach. The main advantage of lazy composition is that the proof that one component meets the expectations of the other components, can be delayed till sufficient detail has been added to the design.

1 Introduction

In the last two decades, there has been considerable progress in the verification of concurrent, reactive systems. Much of the research has been devoted to the development of formalisms such as temporal logics [Eme90,Lam94,MP92,CM88] and

* Supported by the Air Force Office of Scientific Research under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931 and CCR-9300444. Based on earlier work [Sha93b] funded by Naval Research Laboratory (NRL) under contract N00015-92-C-2177. Connie Heitmeyer, Ralph Jeffords, and Pierre Collette gave useful feeback on the work cited above. John Rushby, Sam Owre, and Nikolaj Bjørner provided detailed comments on drafts of this paper. Presentations of earlier versions of this work at the meetings of IFIP Working Group 2.3 and at COMPOS'97 yielded valuable insights and criticisms. Martín Abadi and Leslie Lamport prompt and helpful in their responses to various technical queries and with feedback on earlier drafts.

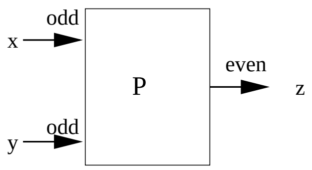

Fig. 1. Even number generator

process algebras [Hoa85,Mil80], and verification methods [Bar85,dBdRR90,dB-dRR94, Sha93a] based on deduction [Eme90, Lam94, MP92, CM88] and model checking [CES86, Kur93, Hol91]. While these techniques are effective on small examples—mutual exclusion, basic cache consistency algorithms, and simple communication protocols—the difficult problem of scaling these techniques up to large and realistic systems has remained largely unsolved.

Large-scale concurrent systems are usually defined by composing together a number of components or subsystems. The typical verification methods are non-compositional and require a global examination of the entire system. In the *deductive* approach to verification, this means that a property such as an invariant has to be verified with respect to each transition of all of the components in the system. Verification approaches based on *model checking* also fail to scale up gracefully since the global state space that has to be explored can grow exponentially in the number of components [GL94]. The purpose of a *compositional* verification approach is therefore to shift the burden of verification from the global level to the local, component level so that global properties are established by composing together independently verified component properties.

To motivate compositional verification, we can consider a very simple example of an adder component P shown in Figure 1 that adds two input numbers x and y and places the output in z. Here x, y, and z can be program variables, signals, or latches depending on the chosen model of computation. The system containing P as a component might require its output z to be an even number, but obviously P cannot unconditionally guarantee this property of the output z. It might be reasonable to assume that the environment always provides odd number inputs at x and y, so that with this assumption it is easy to show that the output numbers at z are always even. Only local reasoning in terms of P is needed to establish that z is always even when given odd number inputs at x and y.

If, as is shown in Figure 2, P is now composed with another component Q that generates the inputs at x and y, then to preserve the property that only even numbers are output at z, Q must be shown to output only odd numbers at x and y. However, the demonstration that Q provides only odd numbers as outputs at x and y might require assumptions on the inputs taken by Q, where z itself might be such an input. If in showing that Q produces odd outputs at x and

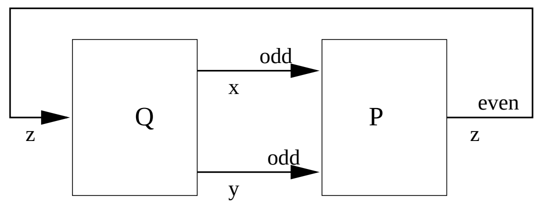

Fig. 2. Odd and even number generators

y, one has to assume that the z input is always even, then we have an obvious circularity and nothing can be concluded about the oddness or evenness of x, y, and z. If this circularity can somehow be broken, we then have a form of well-founded mutual recursion between P and Q that admits a proof by simultaneous induction that x and y are always odd and z is always even. The circularity can be broken by noting that that a z output for P is even as long as the preceding x and y inputs are odd, and the x and y outputs for Q are odd as long as the preceding z input is even.

The assume-guarantee paradigm is the best studied approach to compositional verification [AL93, AL95, AP93, CMP94, Col93, Hoo91, Jon83, MC81, PJ91, Pnu84, Sta85, XCC94, XdRH97, Zwi89]. In this approach, a property of a component is stated as a pair (A, C) consisting of a guarantee property C that the component will satisfy provided the environment to the component satisfies the assumption property A. The interpretation of (A, C) has to be carefully defined to be non-circular. Informally, a component P satisfies (A, C) if the environment to P violates A before the component fails to satisfy C. When two or more components, P_1 satisfying (A_1, C_1) and P_2 satisfying (A_2, C_2), are composed into a larger component $P_1 \| P_2$, the assumption A together with property C_1 of component P_1 must be used to show that P_1 does not violate assumption A_2, and correspondingly, A and C_2 must be used to show that P_2 does not violate A_2. Discharging these proof obligations allows one to conclude that the composite component $P_1 \| P_2$ has a similar property (A, C) where C follows from A, C_1, and C_2. The assume-guarantee technique as described informally still suffers from the earlier circularity. The formal details of the assume-guarantee technique are deferred to Section 2. The assume-guarantee approach has been more widely studied than actually used. The primary difficulty in applying this approach for compositional verification is that it requires component guarantee properties to be strong enough to entail any potential environment constraints. It is obviously not easy to anticipate all the potential constraints that might be placed on a component by the other components in a system.

The *lazy composition* approach advocated in this paper builds on conventional techniques while avoiding the difficulties associated with the assume-guarantee approach [Sha93b]. Lazy composition works at the level of the *specification* of component behavior. In lazy composition, a property C of a component specified as P is actually proved of the system $P\|E$ obtained by composing P with an abstract environment specification E that captures the expected behavior of the environment. When the component specification P is composed with another component specification Q, then C might no longer be a property of the specification $P\|Q$ since Q might not satisfy the constraint E. However, C is a property of the composition $P\|(Q \wedge E)$ obtained by strengthening Q to additionally satisfy E. This allows local properties such as C to be used as global properties of the specification of a larger system. If in fact the combined specification $P\|(Q \wedge E)$ can be simplified to $P\|Q$, then clearly the constraint E is redundant and can be eliminated. However, it is not imperative that (properties guaranteed by) Q already imply E as is the case with the assume-guarantee technique. While the assumed environment specification has eventually to be shown to hold of the other components in the system, this proof obligation can be discharged lazily as the system design is being refined. The demonstration that $P\|(Q \wedge E)$ is refined by $P\|Q$ uses inductive reasoning on computations so that any possible circularity between assumptions E and guarantees C is avoided. Thus lazy composition allows global properties to be proved by local component-wise reasoning combined with a one-time demonstration that each component satisfies the accumulated constraints imposed by the other components. There are several other tradeoffs between lazy composition and assume-guarantee reasoning that are discussed in Section 3.

The lazy composition approach is quite general and can be applied to a wide variety of synchronous and asynchronous computational models, but this paper considers only one such model, namely, asynchronous transition systems with interleaving composition.

We first present some background on compositional verification in Section 2. Lazy composition is introduced in Section 3. Some examples illustrating the use of lazy composition in verifying safety properties are presented in Section 4. The elimination of environment constraints by means of refinement proofs is described in Section 5. The verification of liveness properties using lazy composition is given in Section 6. A comparison between lazy composition and other compositional approaches is given in Section 7.

2 Background

The presentation in this paper is entirely at the semantic level where we are dealing with states, predicates (sets) and relations on states, computations as infinite sequences of states, and properties as sets of computations. We will also speak of sets of sequences and properties interchangeably.

Asynchronous Transition Systems. In its simplest form, an *asynchronous transition system* is a triple $\langle \Sigma; I, N \rangle$ of a state type Σ, an initial set of states I, and a reflexive (stuttering-closed) *next-state* relation N that defines the possible *atomic* actions of the system. Seen as a *closed system*, i.e., one with no interaction with an outside environment,[2] a valid *computation* of such a system consists of an infinite sequence of states σ whose initial state $\sigma(0)$ is in I, i.e., $I(\sigma(0))$ holds, and N holds of each pair of adjacent states, i.e., for all i, $N(\sigma(i), \sigma(i+1))$. A property is a set of infinite state sequences. If P is an asynchronous transition system, the set of its computations in the closed interpretation is represented as $[\![P]\!]$. The transition system P *has a property* A, in symbols, $[\![P]\!] \models A$, iff the set of computations $[\![P]\!]$ is a subset of the set of sequences corresponding to the property A. We write $\models A$ when the property A is valid, i.e., contains all the infinite sequences. Properties (sets of infinite sequences) can be combined with connectives $\neg A$ (complement), $A \vee B$ (union), $A \wedge B$ (intersection), and $A \supset B$ which is defined as $\neg A \vee B$. One transition system P *refines* another transition system Q when $\models [\![P]\!] \supset [\![Q]\!]$. In typical usage below, a transition system will be given as $\langle I, N\rangle$ leaving the state type Σ implicit.

Safety Properties. A safety property informally asserts that nothing bad happens during a computation. Let $\sigma[i]$ represent the finite prefix consisting of the first i states $\sigma(0)\ldots\sigma(i-1)$ of σ. A safety property [AS85] is one that excludes an infinite sequence σ exactly when it excludes all extensions $\sigma[i] \circ \rho$ of some finite prefix $\sigma[i]$ of σ. This means that safety properties are falsified by some finite prefix of a sequence. For any property A, there is a property A^S (the *safety closure* of A) which is the strongest safety property containing A defined as $\{\sigma \mid \forall i : \exists \rho : \sigma[i] \circ \rho \in A\}$. The property (set) A^S is clearly a safety property. If A is a safety property, we say that $\sigma[n] \in A$ when $\sigma[n] \circ \rho \in A$ for some ρ.

Liveness Properties. Liveness properties assert that something good eventually happens during the computation. Such properties hold of some infinite extension of any finite sequence α, i.e, they can always be satisfied by an appropriately chosen sequence of states. A liveness property can exclude an infinite sequence σ but must contain some extension of $\sigma[i]$ for each i. Given a property A, let A^L (the *liveness closure* of A) be $A \vee \neg A^S$, where $\neg A^S$ represents the complement of A^S. Then A^L is a liveness property because if for some α there is no ρ such that $\alpha \circ \rho \in A^L$, then since $A \subseteq A^L$, $\forall \rho : \alpha \circ \rho \notin A$, but then $\forall \rho : \alpha \circ \rho \notin A^S$. This is a contradiction since every infinite sequence must be in A^L or A^S. Thus every property A can be expressed as the conjunction of a safety property A^S and a liveness property A^L [Sch87].

Stuttering Invariance. A set of sequences A is *stuttering invariant* if whenever $\sigma[i+1] \circ \rho \in A$ then $\sigma[i+1] \circ \sigma(i) \circ \rho \in A$. In words, if A contains a sequence, then it contains all variants of this sequence obtained by stuttering individual states in the sequence finitely often. Stuttering arises naturally when there is a

[2] The closed interpretation here means that each transition of a valid computation satisfies the next-state relation N leaving no room for any environment transitions other than those already specified by N.

notion of an observation of a transition system so that some of the transitions have no observable effect. Stuttering invariance is often imposed as a constraint on the allowable properties so that the resulting transition system can always be implemented using internal unobservable state components.

Published explanations of assume-guarantee proof techniques often implicitly rely on stuttering invariance without explicitly mentioning it. Stuttering invariance is needed to argue that if we are given safety properties A and B such that $\sigma[i] \in A$ and $\sigma[i] \in B$, then $\sigma[i] \in A \wedge B$. Such a result is valid if A and B are stuttering invariant properties. To see how the result can fail to hold, let A consist of the strictly increasing sequences of even numbers and B consist of the strictly increasing sequences of prime numbers. Both A and B are safety properties that are not stuttering invariant. The singleton prefix $\langle 2\rangle$ is in both A and B but $A \wedge B$ is empty.[3]

Expressing Properties. The above notions of computation and property are typical of the use of linear-time temporal logics for stating and proving properties of closed systems. Examples of such logics include

- Manna and Pnueli's LTL [MP92] with the temporal operators $\bigcirc$ (next-time), $\Box$ (always), and $\Diamond$ (eventually). Properties expressed in LTL that use the $\bigcirc$ operator are not necessarily stuttering invariant.
- Chandy and Misra's Unity [CM88] with operators **invariant**, **stable**, **unless**, **until**, and **leadsto** which are applied to state predicates so that temporal formulas are not nested. Unity properties are stuttering invariant.
- Lamport's temporal logic of actions [Lam94] which drops the next-time operator from linear-time temporal logic but allows temporal operators to range over *actions*, i.e., binary relations over states. TLA is designed to admit only stuttering invariant properties.

In the examples below, we restrict ourselves to some simple operators for defining properties. If p is a predicate on states, then

1. **invariant** p holds of σ iff $\forall i : p(\sigma(i))$. This is a safety property.
2. **eventually** p holds of σ iff $\exists i : p(\sigma(i))$. This is a liveness property for any satisfiable predicate p since any finite sequence can be extended to one in which p eventually holds.

For a given transition system $\langle I, N\rangle$, the invariance of p can be proved using induction by showing that for all states s in Σ, $\vdash I(s) \supset p(s)$, and for all states s and s', and $\vdash p(s) \wedge N(s, s') \supset p(s')$.

[3] A weaker requirement than stuttering invariance suffices for the soundness of the assume-guarantee proof reasoning methods. A safety property A must include the infinite sequence $\sigma[i+1] \circ \sigma(i)^{\omega}$ obtained by infinitely stuttering the last state of any nonempty finite prefix $\sigma[i+1]$ in A.

Components as Open Systems. The next step is to extend the model to open systems so that components can be independently specified and composed to form larger systems. If Σ is the set of global states of the large system, then a component i can be given as a triple $\langle \Sigma; I_i, N_i \rangle$. However, we can no longer take the closed interpretation since a computation must include the actions taken by other components. In the *open system* interpretation, a computation is an infinite sequence of states whose initial state is in I_i and each pair of adjacent states is either related by N_i or is an arbitrary environment transition. The open system interpretation is much too liberal and does not admit any interesting properties since there are no constraints on the environment actions. This can be partially overcome by placing weak constraints on the environment actions, e.g., the values of the local variables of a component must be left unchanged by its environment. With some constraint on the environment actions, one can actually verify reasonably interesting local properties of a component. For example, in TLA [Lam94], the next-state relation of a component is written as $[N]_f$ which holds of a pair of states s, s' when $N(s, s') \vee f(s') = f(s)$. The state function f typically projects out the local variables of the component so that the environment transitions must not affect the values of these variables. In Lynch and Tuttle's I/O automata [LT87], a component is an input-enabled automaton with its own local state so that any component properties established with respect to this interpretation remain globally valid even in composition with other components.

Even with such restrictions on the environment behavior, the open system interpretation is somewhat weak since many properties of a component can only be proved by assuming a stronger degree of cooperation from the environment. We have already seen the example of the adder component of Figure 1 which can be shown to always output even numbers when given odd number inputs by its environment.

The Owicki–Gries Method. The Owicki–Gries method [OG76] is the first attempt at a component-wise decomposition of the verification problem. In this method, one proves a global invariant of the composition $P_1 \| P_2$ by showing it to be a local invariant of one of the components, say P_1, and a stable predicate, i.e., one that is never falsified, of the other component P_2. In other words, one component establishes the invariant and the other component does not falsify it. This method is not really compositional since it requires global reasoning on all the actions of each component in order to establish an invariant. The Owicki–Gries method was originally proposed in the framework of a proof-outline logic where program components are annotated with assertions. Such program-based proof methods can be quite restrictive when compared to the use of high-level behavioral specifications as given by asynchronous transition systems.

Compositional Verification Using the Assume-Guarantee Approach. The assume-guarantee approach originally proposed by Jones [Jon83] and Misra and Chandy [MC81] is perhaps the most widely studied compositional verification technique for concurrent systems. The presentation of this approach given below is adapted from Abadi, Lamport, and Plotkin [AL93, AL95, AP93] and Collette [Col94]. An assume-guarantee specification of a a component property is given as a pair (A, C) consisting of an assumption property A and a guarantee property C. To capture (A, C) is defined as $A \overset{+}{\longrightarrow\!\triangleright} C$ (A *secures* C) which is the subset of $A \supset C$ defined as $\{\sigma \in A \supset C \mid \forall i : \sigma[i] \in A^S \supset \sigma[i+1] \in C^S\}$. Thus $A \overset{+}{\longrightarrow\!\triangleright} C$ rules out unrealizable implementations of $A \supset C$ that exhibit computations where C^S fails before the failure of A^S can be detected by the component. Similarly, $A \longrightarrow\!\triangleright C$ (A *maintains* C) is the set of σ in $A \supset C$ such that for all i, $\sigma[i] \in A \supset \sigma[i] \in C$. Note that $A \overset{+}{\longrightarrow\!\triangleright} C \equiv (A \supset C) \wedge (A^S \overset{+}{\longrightarrow\!\triangleright} C^S)$, and $A \longrightarrow\!\triangleright C \equiv (A \supset C) \wedge (A^S \longrightarrow\!\triangleright C^S)$.

Composition of components $P_1 \| P_2$ is defined so that $[\![P_1 \| P_2]\!]$ is the intersection of $[\![P_1]\!]$ and $[\![P_2]\!]$. Since P_1 and P_2 are specified to allow environment transitions, the composition of P_1 and P_2 includes all the interleavings of P_1 and P_2 actions, but also contains computations with simultaneous P_1 and P_2 actions.[4]

The main compositionality rule in the assume-guarantee method [AL95] is stated in Theorem 1.

Theorem 1.

$$\frac{\begin{array}{c} P_i \models A_i \overset{+}{\longrightarrow\!\triangleright} C_i,\ \mathit{for}\ i = 1,2 \\ \models A^S \wedge C_1^S \wedge C_2^S \supset A_1 \wedge A_2 \\ \models A \overset{+}{\longrightarrow\!\triangleright} (C_1 \wedge C_2 \longrightarrow\!\triangleright C) \end{array}}{P_1 \| P_2 \models A \overset{+}{\longrightarrow\!\triangleright} C.}$$

In words, in order to show that the composition $P_1 \| P_2$ has property $A \overset{+}{\longrightarrow\!\triangleright} C$, it suffices to establish the following premises of the compositionality rule:

1. Each P_i has property $A_i \overset{+}{\longrightarrow\!\triangleright} C_i$.
2. The individual environment constraints A_1 and A_2 must be satisfied by the conjunction of the safety parts of the joint environment constraint A and the guarantee properties C_1 and C_2.
3. The joint commitment C must be maintained by the individual commitments C_1 and C_2 when secured by the environment assumption A.

The formal details justifying the assume-guarantee rule are fairly elaborate, but we can briefly convey some of the intuition by sketching the soundness

[4] To obtain a strict interleaving of P_1 and P_2 actions, such joint actions can be excluded by asserting that the variables written by P_1 and P_2 must be disjoint and never simultaneously updated. Another approach is to label each transition with the agent associated with it, and to have a disjoint set of agents associated with components P_1 and P_2.

argument. It is sufficient to focus our attention on infinite sequences σ such that $\sigma \in (A_1 \xrightarrow{+} C_1) \wedge (A_2 \xrightarrow{+} C_2)$. To show $\sigma \in A \xrightarrow{+} C$, we need to prove both $\sigma \in A \supset C$ and $\sigma \in A^S \xrightarrow{+} C^S$. The argument proceeds in three steps:

- $\sigma \in (A^S \xrightarrow{+} C_1^S \wedge C_2^S)$.
  That is, for any n, $\sigma[n] \in A^S$ implies $\sigma[n+1] \in C_1^S \wedge C_2^S$. This can be proved by induction on n using premises 1 and 2 while noting that the stuttering invariance of A^S, C_1^S, and C_2^S is used in this argument.
- $\sigma \in A^S \xrightarrow{+} C^S$.
  By premise 3, for any n, $\sigma[n] \in A^S$ implies $\sigma[n+1] \in C_1^S \wedge C_2^S \longrightarrow C^S$. From step 1, we therefore have $\sigma[n+1] \in C^S$.
- $\sigma \in A \supset C$. For $\sigma \in A$, since $A \supset A^S$, we have by $A^S \xrightarrow{+} C_1^S \wedge C_2^S$ that $\sigma \in C_1^S \wedge C_2^S$. By premise 2, this yields $\sigma \in A_1 \wedge A_2$. By premise 1 and the definition of $\xrightarrow{+}$, we have that $\sigma \in C_1 \wedge C_2$. We can then apply premise 3 with the definitions of the connectives $\xrightarrow{+}$ and $\longrightarrow$ to obtain $\sigma \in C$.

There are some approaches to modular verification based on model checking that employ a weak form of assume-guarantee reasoning. In the work of Grumberg and Long [GL94], assume-guarantee properties (A, C) are treated as implications $A \supset C$ and not $A \xrightarrow{+} C$. Note that the use of implication for assume-guarantee reasoning is not valid in general, and is sound only for a restricted form of Theorem 1 where the cycle of dependencies between A_1, C_2, A_2, and C_1 has been broken. If A and C are just linear-time temporal logic (LTL) formulas, then LTL model checking can be used to verify $A \supset C$ of component P since this implication is also in LTL. If C is a CTL or CTL* formula, then the situation is more complicated since the implication $A \supset C$ is not a well-formed CTL or CTL* state formula, and furthermore, it does not capture the intended meaning of A as an assumption [Jos90] which is that C must hold on the computation tree whose paths have been pruned according to A. Then, A can be chosen as a ∀CTL formula that characterizes the subtree of the computation tree that meets the assumption. For the case of ∀CTL assumptions and synchronous Moore machine composition, Grumberg and Long give a way of compiling the assumption A into a tableau automaton A^T so that $P \| A^T \models C$ iff $P \models A \supset C$. Kupferman and Vardi [KV96] analyze the complexity of various linear and branching-time variants of modular model checking. Alur and Henzinger [AH96] give an assume-guarantee rule for proving language containment in the context of the synchronous composition of a form of Mealy machines called *reactive modules*.

3 Lazy Composition

Lazy composition differs from the assume-guarantee approach in several respects.

1. *Components are not treated as blackboxes.* Compositional verification merely requires that properties be proved locally at the component level. It does not require that components be treated as blackboxes for this purpose. The assume-guarantee approach requires the assumptions to be discharged solely by means of the guarantee properties of a component. The actual implementation of the component is never used for discharging proof obligations. This means that the guarantee properties must either somehow anticipate the possible constraints imposed by other components, or they must contain implementation details. Lazy composition on the other hand does not take a blackbox view of components and allows the behavioral specification to be used for discharging the constraints imposed by other components. Since a typical high-level behavioral specification might not contain enough detail to discharge such external constraints, lazy composition allows the constraints to be discharged lazily as the specification is being refined.
   Blackbox assume-guarantee specifications can be independently refined to yield implementations in terms of smaller blackbox components. Abadi and Lamport [AL95] give a *decomposition* rule for showing that $P' \| Q'$ refines $P \| Q$ when P' refines P and Q' refines Q. This rule has a premise similar to premise 2 of the compositionality rule of Theorem 1 which has the same drawback of requiring the environment constraints to be anticipated in the blackbox specification.
2. *Composition is not necessarily conjunction.* Conjunction can be used to define the interleaving composition of two asynchronous transition systems by a suitably chosen global constraint (see [AL95] and Footnote 4). Instead of encoding composition using conjunction, we regard the definition of the precise notion of composition as something that is fixed by the model of computation and not by the inference rule for composition. For asynchronous composition, one takes the interleaving of the atomic actions of each component, whereas for synchronous composition, i.e., globally clocked systems, one takes the conjunction of the atomic actions. Other formalisms that have asynchronously operating components with synchronous communication, e.g., CSP [Hoa85], can be modelled by means of a suitable definition of composition.
3. *Environment assumptions are specified as abstract components not properties.* One difficulty with environment assumptions as properties is that they apply to both the environment and the component. Typically, these constraints should apply only to environment actions and not the component actions. If we take the example of a bank account component, the environment might be required to only deposit and not withdraw money from the component but such a constraint should not apply to the component. There is no elegant way of stating this distinction between the component and its environment when the environment constraints are stated as properties rather than abstract components.
4. *No assume-guarantee proof obligations are generated.* With lazy composition, properties of a component P_i are proved in the context of an abstract

environment E_i. The composition rule ensures that all local properties are global properties of the composition. It does this by adding (conjoining, as explained below) the environment constraints of one component to the specification of the other component so that the resulting system has the form $(P_1 \wedge E_2) \| (P_2 \wedge E_1)$.

This form of composition appears dishonest (and lazy) since it sidesteps the question of whether the original specification of one component satisfies the environment assumptions of the other. However, specifications are meant to be partial and are therefore not always strong enough to anticipate the environment constraints that can be placed on a component. The best that one can do therefore is assert that if the specifications of each component is strengthened with the environment constraints required by the other component, then the resulting system satisfies the local properties of both components. If a component specification is strong enough to discharge any constraints placed on it, then the strengthening is redundant and can be eliminated by simplification. Otherwise, an implementation of the component is required to satisfy the stronger specification including the environment constraint.

The flexibility in postponing the assume-guarantee proof obligations is needed since some proofs might require the additional information that is provided when the specification is refined. If these proof obligations have to be proved as in the assume-guarantee proof method, then the component specifications must be quite detailed and strong. Since no proof obligations are discharged and environment assumptions impose additional, possibly unanticipated, constraints on a component specification, a component cannot be independently refined in the lazy composition approach. A component can only be independently refined when the component specification already implies all the environment constraints that might be required of it. There is, however, an advantage to refining in the global context where these assumptions are known since global properties can be exploited in the refinement (see Section 5).

5. *Composition can yield inconsistent specifications.* This is also the case when composition is defined as conjunction. In the case of lazy composition, this can arise because there is no computation that is compatible with the collection of constraints given in the specification.

Summarizing the discussion so far, lazy composition takes the middle ground between global verification as used in the Owicki–Gries approach and the strictly modular, property-based verification used in the assume-guarantee approach. Lazy composition is a proof style that uses a suitably weak characterization of a cooperative environment in composition with which a component can exhibit a given property. Once such an environment has been identified, the familiar verification techniques for proving safety, liveness, and refinement properties can be used. In the presentation of lazy composition, it will be assumed for convenience that there is a fixed environment specification for each component, but in practice, the environment can be varied according to the desired property of the component.

We now move on to the details of lazy composition for asynchronous transition systems while noting that the techniques can easily be adapted to other models and notions of composition. As already stated, an asynchronous transition system is given by a triple $\langle \Sigma; I, N\rangle$ consisting of the state Σ, an initialization predicate I on the state, and a binary next-state relation N. Given such a triple P of the form $\langle \Sigma; I, N\rangle$, the closed interpretation of P is written as $[\![P]\!]$ and defined as the set sequences $\{\sigma | I(\sigma(0)) \wedge \forall i : N(\sigma(i), \sigma(i+1))\}$. We focus mainly on closed interpretations since one cannot prove interesting properties of computations that admit arbitrary environment actions. When we are talking about components, we will assume that Σ is the global state type and omit it from the transition system.

Given two transition systems, P_1 of the form $\langle I_1, N_1\rangle$, and P_2 of the form $\langle I_2, N_2\rangle$, the composition $P_1 \| P_2$ is the transition system $\langle I_1 \wedge I_2, N_1 \vee N_2\rangle$. Note that composition essentially yields the interleaving of the component transitions.

The environment E is also given as a transition system $\langle I^e, N^e\rangle$. A component together with its environment is given as a pair $P//E$. The set of computations corresponding to $P//E$, i.e., $[\![P//E]\!]$, is defined as $[\![P\|E]\!]$, i.e., the closed interpretation of $P\|E$. Note that though $P//E$ and $P\|E$ have the same computations, the notation $P//E$ is chosen to emphasize the syntactic asymmetry between component P and environment E.

Given two transition systems P_1 and P_2, the conjunction of these, $P_1 \wedge P_2$, is $\langle I_1 \wedge I_2, N_1 \wedge N_2\rangle$. Let P_i^e denote the component-environment specification $P_i//E_i$. Given two component–environment specifications P_1^e and P_2^e, the *closed co-imposition* of these two specifications $P_1^e \otimes P_2^e$ is defined as the transition system $(P_1 \wedge E_2) \| (P_2 \wedge E_1)$. The *open co-imposition* of P_1^e and P_2^e, written as $P_1^e \times P_2^e$, is defined as $(P_1^e \otimes P_2^e)//(E_1 \wedge E_2)$ and its computations contain actions corresponding to

1. P_1 but respecting E_2,
2. P_2 but respecting E_1, and
3. Environment actions respecting E_1 and E_2.

The closed co-imposition $P_1^e \otimes P_2^e$ yields a system with only the actions of P_1 and P_2, whereas the open co-imposition $P_1^e \times P_2^e$ yields a system with environment actions that are constrained to conform to both E_1 and E_2. Both operators are associative and commutative. It is easy to see that the property preservation result given in Theorem 2 holds so that $[\![P_1^e \otimes P_2^e]\!]$ and $[\![P_1^e \times P_2^e]\!]$ are both subsets of $[\![P_1^e]\!]$, and hence any properties of P_1^e are also properties of $P_1^e \otimes P_2^e$ and $P_1^e \times P_2^e$.

Theorem 2. *1.* $\models [\![P_1^e \otimes P_2^e]\!] \supset [\![P_1^e]\!]$
2. $\models [\![P_1^e \times P_2^e]\!] \supset [\![P_1^e]\!]$

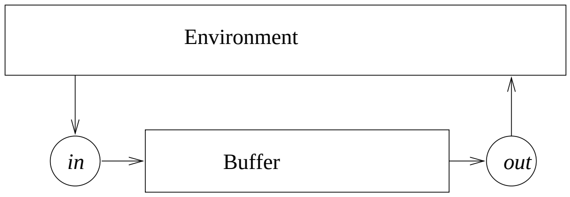

Fig. 3. A FIFO buffer with environment

We will henceforth ignore the closed co-imposition operator since its properties are similar to those of open co-imposition. The use of the co-imposition operation in lazy composition will be illustrated in Section 4. The obvious problem with lazy composition is that it asserts the property preservation of $P_1^e \times P_2^e$ and says nothing about $P_1 \| P_2$. By discharging proof obligations similar to those in Theorem 1, we can show that the transition system specification $P_1^e \times P_2^e$ is equivalent to the specification $(P_1 \| P_2) // (E_1 \wedge E_2)$, where the latter system contains actions corresponding to P_1 and P_2 without any restrictions, and the environment action $E_1 \wedge E_2$. In Section 5, we show that the environment constraints can be discharged in this manner by showing that P_2 refines E_1, and P_1 refines E_2. These refinement proofs can actually be carried out in the context of global invariants, i.e., invariants of $P_1^e \times P_2^e$. The resulting refinement proof obligations are similar to the assume-guarantee proof rule where $A^S \wedge C_1^S \wedge C_2^S$ must entail $A_1 \wedge A_2$.

4 Using Lazy Composition

Lazy composition will be illustrated by means of the example of a FIFO buffer component that is composed from two smaller FIFO buffer components. This examples has been frequently used with minor variations in the compositionality literature [Col93, AL95].

A single (bounded or unbounded) FIFO buffer component shown in Figure 3 consists of a buffer variable b that contains a queue of values, and the input and output variables *in* and *out* which contain values or are empty, i.e., contain a distinguished value $\perp$. Two history variables are used to specify the correct behavior of the buffer. The variable *inh* is a stack of all the non-$\perp$ values placed by the environment into in, and the variable *outh* is the stack of non-$\perp$ values read by the environment from out. The non-stuttering actions of the buffer are:

- Read a non-$\perp$ value from the variable in and enqueue it at the back of b while setting in to $\perp$. Formally, this is captured by the relation between the pre-state $\langle in, b, out, inh, outh\rangle$ and the post-state $\langle in', b', out', inh', outh'\rangle$ as

$$read \triangleq \begin{cases} \quad in \neq \bot \\ \wedge\ b' = enqueue(in, b) \\ \wedge\ in' = \bot \\ \wedge\ out' = out \\ \wedge\ outh' = outh \\ \wedge\ inh' = inh \end{cases}$$

– Dequeue a value from the front of queue b and place this value in the variable out when out is empty. Formally,

$$write \triangleq \begin{cases} \quad nonempty?(b) \\ \wedge\ out = \bot \\ \wedge\ b' = dequeue(b) \\ \wedge\ out' = front(b) \\ \wedge\ outh' = push(front(b), outh) \\ \wedge\ in' = in \\ \wedge\ inh' = inh \end{cases}$$

In the initial state, all the variables associated with the buffer are empty. Formally,

$$init_b \triangleq (out = \bot \wedge b = outh = null).$$

The buffer component P is then given by the pair $\langle init_b, read \vee write\rangle$.

The environment component initializes the variables in and inh so that they are both empty:

$$init_e \triangleq (in = \bot \wedge inh = null).$$

In each non-stuttering action, the environment leaves b unchanged and may change the value of in when empty and may set the value of out to $\bot$. Formally,

$$load \triangleq (in = \bot \wedge in' \neq \bot \wedge inh' = push(in', inh))$$
$$unload \triangleq (out \neq \bot \wedge out' = \bot \wedge outh' = outh)$$
$$env \triangleq \begin{cases} \quad (load \vee (in' = in \wedge inh' = inh)) \\ \wedge\ (unload \vee (out' = out \wedge outh' = outh)) \\ \wedge\ b' = b \end{cases}$$

The environment component E is given by the pair $\langle init_e, env\rangle$.

It is easy to prove by induction that

$$[\![P//E]\!] \models \textbf{invariant}\ inh = \overline{in} \circ q2s(b) \circ outh,$$

where $\circ$ is stack concatenation, $q2s(b)$ converts the queue b into a stack by repeatedly pushing elements from the front of queue b, and $\overline{in}$ is $push(in, empty)$ when $in \neq \bot$, and $empty$, otherwise. We have thus proved an invariant of a buffer component P by assuming that the environment behavior is as specified by E.

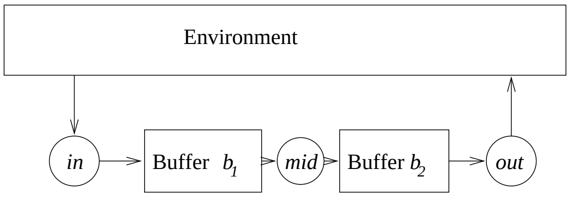

Fig. 4. FIFO buffer composed from smaller buffers

Compositional reasoning is used when two such buffers are composed as shown in Figure 4 to implement a single buffer. We do this by taking one instance $P_1//E_1$ of the buffer as specified above but renaming the variables b to b_1, out to mid, and $outh$ to $midh$, and a second instance $P_2//E_2$ with b renamed to b_2, and where in and inh are just mid and $midh$, respectively. In other words, buffer P_1 communicates values to buffer P_2 via mid.

Having already proved the invariant above for a FIFO buffer P, the goal now is to prove a similar invariant $inh = \overline{in} \circ q2s(b) \circ outh$, for some b, for the composition $(P_1 \| P_2)//(E_1 \wedge E_2)$ of the two buffers. However, we cannot use the invariant proved of P for composite buffers with P_1 and P_2 since those invariants are proved for the systems $P_1//E_1$ and $P_2//E_2$.

The claim $\models [\![(P_1 \| P_2)//(E_1 \wedge E_2)]\!] \supset [\![P_1//E_1]\!]$ is not provable since the definitions of P_1 and P_2 are not strong enough to imply the constraints E_2 and E_1, respectively. This is because E_1 specifies that each environment action must leave the buffer variable b_1 unchanged and that the variable in must be written only by the environment. The actions of P_2 place no constraints on the update of the values of b_1 or in. Since we cannot demonstrate $\models [\![(P_1 \| P_2)//(E_1 \wedge E_2)]\!] \supset [\![P_1//E_1]\!]$, the invariant for P_1^e, namely, $inh = in \circ b_1 \circ midh$, cannot be used as a global invariant of $(P_1 \| P_2)//(E_1 \wedge E_2)$.

The best that we can do therefore is to conclude $\models [\![P_1^e \times P_2^e]\!] \supset [\![P_1^e]\!] \wedge [\![P_2^e]\!]$, so that the conjunction of the individual invariants holds for $P_1^e \times P_2^e$. From the conjunction of the two invariants:

1. $[\![P_1^e \times P_2^e]\!] \models$ **invariant** $inh = \overline{in} \circ q2s(b_1) \circ midh$
2. $[\![P_1^e \times P_2^e]\!] \models$ **invariant** $midh = \overline{mid} \circ q2s(b_2) \circ outh$

we can conclude

$$[\![P_1^e \times P_2^e]\!] \models \textbf{invariant } inh = \overline{in} \circ q2s(b_1) \circ q2s(mid) \circ q2s(b_2) \circ outh.$$

So if we take b to be $s2q(q2s(b_1) \circ \overline{mid} \circ q2s(b_2))$ where $s2q$ is the inverse of $q2s$ and converts a stack back into the corresponding queue, we have the desired invariant $inh = \overline{in} \circ q2s(b) \circ outh$ for $P_1^e \times P_2^e$.

In proving this invariant, we have used only the corresponding invariants of the component buffers and some elementary lemmas about the concatenation operation. We have not directly used the specification of individual buffers themselves. We have worked at the level of the *specification* of the behavior of the individual buffers rather than the corresponding *program* which would be a complete specification of each transition. Since specifications can be partial, it makes sense to conjoin the environment constraints to the component specification rather than discharge them as proof obligations. Thus a more detailed implementation will have to satisfy the higher-level specification of the component as well as the constraints on the component imposed by the other components in the combined system.

When refining P_1 to a more refined specification or a program in the context $P_1^e \times P_2^e$, it is valid to use all the global invariants that have been proved of $P_1^e \times P_2^e$. The introductory example involving odd and even numbers can be used to illustrate the use of such invariants in refinement. The system P there is of the form $\langle I_P, N_P \rangle$ where

$$\begin{aligned} I_P &\stackrel{\triangle}{=} \mathit{even?}(z) \\ N_P &\stackrel{\triangle}{=} (z' = x + y) \wedge (x' = x) \wedge (y' = y) \end{aligned}$$

If P's environment constraint D is of the form $\langle I_D, N_D \rangle$ where

$$\begin{aligned} I_D &\stackrel{\triangle}{=} \mathit{odd?}(x) \wedge \mathit{odd?}(y) \\ N_D &\stackrel{\triangle}{=} \mathit{odd?}(x') \wedge \mathit{odd?}(y') \wedge z' = z \end{aligned}$$

then we can prove the invariant $\mathit{even?}(z) \wedge \mathit{odd?}(x) \wedge \mathit{odd?}(y)$ for the system $P//D$. Let Q be defined to be $\langle I_Q, N_Q \rangle$ where

$$\begin{aligned} I_Q &\stackrel{\triangle}{=} \mathit{odd?}(x) \wedge \mathit{odd?}(y) \\ N_Q &\stackrel{\triangle}{=} (x' = x + z) \wedge (y' = y + z) \wedge (z' = z) \end{aligned}$$

Let E be the unconstrained system consisting of the everywhere-true initialization predicate and next-state relation. We would now like to show that the constraint D is satisfied by Q, but this is not true in general. It does however hold in the context of the invariant $\mathit{even?}(z) \wedge \mathit{odd?}(x) \wedge \mathit{odd?}(y)$. The use of invariants allows $[\![(P//D) \times (Q//E)]\!]$ to be simplified to $[\![P \| Q \| D]\!]$ since $P \wedge E$ simplifies to P, $D \wedge E$ simplifies to D, and $Q \wedge D$ can be simplified to Q given

$$\vdash \mathit{even?}(z) \wedge \mathit{odd?}(x) \wedge \mathit{odd?}(y) \wedge N_Q \supset N_D.$$

We show how invariants can be used in proving a refinement relation between two transition systems using stepwise simulation in the next section.

5 Discharging Proof Obligations by Refinement

We now examine how the familiar notion of refinement via simulation can be used to simplify away the environment constraints E_1 and E_2 in the lazy composition $P_1^e \times P_2^e$. This is analogous to the assume-guarantee proof obligations (premise 2) except that lazy composition is more flexible about how and when these proof obligations are discharged. Recall that in the assume-guarantee approach, the assumptions of one component had to be discharged using the guarantee properties of all the components along with the global environment constraints. As we noted, this has the disadvantage that the guarantee properties have to be chosen to somehow anticipate the likely environment constraints. By contrast, in lazy composition, these proof obligations are discharged lazily during refinement.

The refinement rule establishes the conclusion $\models [\![P]\!] \supset [\![Q]\!]$ by showing that each transition of P can be simulated by a transition of Q . In particular, this means that P inherits all the properties of Q. The simulation of P transitions by Q transitions can be shown in the presence of invariants of P and Q. The invariants might be needed because the simulation relation between the actions of P and Q might not hold outside their respective reachable states. The invariant that is used for P can be an action invariant, a binary relation r on Σ such that $\forall i : r(\sigma(i), \sigma(i+1))$. In this case, we say that **invariant** r holds of σ. Given a state predicate p, an action r, and two transition systems P and Q of the form $\langle I_P, N_P\rangle$ and $\langle I_Q, N_Q\rangle$, respectively, the refinement rule is stated in Theorem 3.

Theorem 3.

$$\frac{\begin{array}{l}[\![P]\!] \models \textbf{invariant } r\\ [\![Q]\!] \models \textbf{invariant } p\\ \vdash p(s) \wedge r(s,s') \wedge N_P(s,s') \supset N_Q(s,s')\\ \vdash I_P(s) \supset I_Q(s)\end{array}}{\models [\![P]\!] \supset [\![Q]\!]}$$

The proof of the refinement rule is by a straightforward induction on the length of the computations in $[\![P]\!]$. The relevance of the refinement rule for compositional verification is that we can use it to eliminate the constraints imposed on one component by another. When composing specifications using the co-imposition operator, we end up with a specification $P_1^e \times P_2^e$ which is equivalent to $(P_1 \wedge E_2)\|(P_2 \wedge E_1)\|(E_1 \wedge E_2)$. To eliminate, say, E_2 from this specification, we need to show that $(P_1 \wedge E_2)\|(P_2 \wedge E_1)\|(E_1 \wedge E_2)$ can be refined by $P_1\|(P_2 \wedge E_1)\|(E_1 \wedge E_2)$. The constraint E_1 can also be similarly eliminated. This kind of refinement can be carried out with the aid of a simple corollary to the refinement rule that can be used to show that $[\![P\|Q]\!]$ refines $[\![P \wedge E\|Q]\!]$ by showing that each P transitions can be simulated by an E transition.

Corollary 4.

$$\frac{\begin{array}{l}[\![(P \wedge E)\|Q]\!] \models \textbf{invariant } p\\ \vdash p(s) \wedge N_P(s,s') \supset N_E(s,s')\\ \vdash I_P(s) \supset I_E(s)\end{array}}{\models [\![P\|Q]\!] \supset [\![(P \wedge E)\|Q]\!]}$$

Note that any global invariant p can be used in proving the stepwise simulation. This is what justifies the use in Section 4 of the invariant $even?(z) \wedge odd?(x) \wedge odd?(y)$ in showing that the strengthening of the specification Q with D is redundant.

6 Liveness

Compositional liveness reasoning is needed for showing progress properties for a component contingent on similar progress properties of other components. For example, the FIFO buffer can only guarantee that an output will always eventually be written if the environment can guarantee that a value in the *out* variable will always eventually be read.

Liveness or progress assumptions have to be handled with some care in compositional verification. For example, suppose a component P guarantees that output z is eventually 4 assuming the input x is eventually 3, and conversely, component Q guarantees an eventual output 3 on x assuming that the input z is eventually 4. If the guarantee properties are used to discharge assumptions, then the composed system $P \| Q$ guarantees that z will eventually take on the value 4 and that eventually x will take on the value 3. This would be unsound since the system actually need not obey either eventuality for x or z and the individual assume-guarantee properties would still be satisfied. The assume-guarantee proof rule is carefully crafted to rule out this kind of circularity by ensuring in premise 2 that the assumptions have to be satisfied solely from the safety parts of the guarantee properties. Component liveness properties are instead expressed as implications in the property C_1 of a component P_1, where the antecedent of the implication is the fairness constraint on the other component. This antecedent is of course easily discharged in the conjunction $C_1 \wedge C_2$ if C_2 includes the fairness condition of P_2.

To admit proofs of liveness properties in lazy composition, it will be necessary to extend the notion of a transition system to include fairness conditions. An asynchronous transition system with fairness is of the form $\langle \Sigma; I, N, F \rangle$ where F is a *fairness* property that a valid computation must satisfy, i.e., $[\![\langle I, N, F \rangle]\!] \equiv [\![\langle I, N \rangle]\!] \wedge F$. It is desirable that the F component be used only to establish progress properties so that any safety property should follow from the system $\langle \Sigma; I, N \rangle$ without F. For this to be the case, the fairness condition F should be *machine closed*, i.e., any finite prefix $\sigma[n]$ in $\langle I, N \rangle$ should be extendable to a sequence $\sigma[n] \circ \rho$ in $[\![\langle I, N, F \rangle]\!]$. F is machine closed with respect to the transition system $[\![\langle I, N \rangle]\!]$ iff $[\![\langle I, N, F \rangle]\!]^S = [\![\langle I, N \rangle]\!]$. For example, if P is a transition system with only one state component x whose value is initially 0, and a next-state relation $x' = x + 2 \vee x' = x + 3 \vee x' = x$, then the property **eventually** $x = 3$ is not a machine-closed fairness condition since it excludes the computations in which x takes the value 2.

Typical notions of fairness such as weak and strong fairness are machine closed with respect to the closed interpretation of a single transition system.

An action r is said to be enabled in a state s, formally $enabled(r)(s)$, iff there exists a state s' such that $r(s, s')$ holds. A predicate p holds infinitely often on a sequence σ iff $\forall i : \exists j : j > i \wedge p(\sigma(j))$. Similarly, an action r holds infinitely often on σ iff $\forall i : \exists j : j > i \wedge r(\sigma(j), \sigma(j+1))$. A sequence σ is said to be *weakly fair* with respect to an action r iff either $\neg enabled(r)$ holds infinitely often or r holds infinitely often on σ. A sequence σ is said to be *strongly fair* with respect to action r iff r holds infinitely often on σ when $enabled(r)$ does. It can be shown that F is machine closed with respect to transition system $\langle I, N \rangle$ if F is a conjunction of weak and strong fairness assertions on actions $r_1, \ldots, r_n$ such that each r_i is *unblocked* in $\langle I, N \rangle$,[5] i.e.,

$$[\![\langle I, N \rangle]\!] \models \textbf{invariant } enabled(r_i) \supset enabled(r_i \wedge N).$$

When F is machine closed with respect to $\langle I, N \rangle$, we say that the fair transition system $\langle I, N, F \rangle$ is machine closed.

The situation is not so simple for transition systems whose computations include both component and environment transitions. The definition of composition for fair asynchronous transition systems is

$$\langle I_1, N_1, F_1 \rangle \| \langle I_2, N_2, F_2 \rangle \stackrel{\triangle}{=} \langle I_1 \wedge I_2, N_1 \vee N_2, F_1 \wedge F_2 \rangle.$$

The purpose of distributing the fairness conditions among the various components is to allow componentwise properties to be deduced using just the relevant global fairness conditions. In particular, machine closure is defined only with respect to a closed interpretation so that it is only required for specifications such as $P//E$ or $P_1^e \times P_2^e$. Given the above definition of composition for fair asynchronous transition systems, all fairness conditions are global and apply to all components.

In a blackbox style of component specification, implementability considerations require the component fairness condition to be machine closed with respect to the open interpretation, and also *receptive*, i.e., machine closed without relying on cooperation from the environment. The receptiveness constraint on the fairness condition can exclude unconditional strong fairness constraints since a hostile environment can enable and disable a component action r without allowing the component a chance to execute r. Receptiveness is a sensible restriction when specifying an open component operating in an uncontrolled environment, but this is not the situation in compositional verification since the environment includes components whose specifications are an integral part of the design.

Given the definition of composition extended with fairness conditions, the definitions of the operations $\otimes$ and $\times$ remain unchanged from Section 3. The property preservation results claimed in Theorem 2 also holds in the presence of fairness conditions.

[5] Abadi and Lamport [AL95] state this constraint differently by requiring each r_i to be a possible program action. This is equivalent since the fairness constraint r_i can just as well taken to be $N \wedge r_i$.

There is however one serious problem with lazy composition in the presence of fairness. The co-imposition $P_1^e \times P_2^e$ of machine-closed specifications $P_1 // E_1$ and $P_2 // E_2$ is not necessarily machine closed. The co-imposition contains the conjunctions $P_1 \wedge E_2$, $P_2 \wedge E_1$, and $E_1 \wedge E_2$. The conjunction of two transitions systems $\langle I_1, N_1, F_1 \rangle \wedge \langle I_2, N_2, F_2 \rangle$ is defined as $\langle I_1 \wedge I_2, N_1 \wedge N_2, F_1 \wedge F_2 \rangle$. Since the actions of the conjoined transition system are specified by $N_1 \wedge N_2$ which is more constrained than either N_1 or N_2, the fairness condition $F_1 \wedge F_2$ might not be machine closed in the resulting transition system. For example, let $x' = x + 1$ be a possible action of P_1 where P_1 initializes x to 0 and has no actions that decrement or reset x. Then it can be proved of $[\![P_1^e]\!]$ that if the increment action is weakly fair, then **eventually** $x = 3$. However, if E_2 in P_2^e requires that x not be incremented, then the set of computations $[\![P_1^e \times P_2^e]\!]$ is empty since the only possible computations are those where the value of x is never changed and these are ruled out by the weak fairness requirement on the increment action. Of course, the property **eventually** $x = 3$ is vacuously preserved in this case. Note that machine closure is violated in this example even if E_2 contains no fairness conditions simply because E_2 blocks a fair action of P_1.

There is therefore a proof obligation that the system $P_1^e \times P_2^e$ be shown to be machine closed. In the special case of fairness conditions that only contain weak and strong fairness assertions, this proof obligation can be discharged by showing that each fair action is unblocked in the combined system.

The notion of refinement used to eliminate environment constraints has to be extended to fair asynchronous transition systems. The goal is to show that $\models [\![\langle I_P, N_P, F_P \rangle]\!] \supset [\![\langle I_Q, N_Q, F_Q \rangle]\!]$. For this, we need to add one additional premise to the refinement rule in Section 5.

Theorem 5.

$$\frac{\begin{array}{l} [\![P]\!] \models \textbf{invariant } r \\ [\![Q]\!] \models \textbf{invariant } p \\ \vdash p(s) \wedge r(s, s') \wedge N_P(s, s') \supset N_Q(s, s') \\ \vdash I_P(s) \supset I_Q(s) \\ \models [\![\langle I_P, N_P, F_P \rangle]\!] \supset F_Q \end{array}}{\models [\![P]\!] \supset [\![Q]\!]}$$

The discharging of the new premise can require temporal reasoning. For the case of fairness conditions that are conjunctions of weak and strong fairness assertions, one can simply show that to any weakly fair action r_i in Q, there is a weakly or strongly fair action r'_j in P such that

$$[\![\langle I_P, N_P \rangle]\!] \models \textbf{invariant } r'_j \supset r_i$$

and

$$[\![\langle I_P, N_P \rangle]\!] \models \textbf{invariant } \mathit{enabled}(r_i) \supset \mathit{enabled}(r'_j \wedge N_P).$$

Similarly, to each strongly fair action in Q, there must be a corresponding strongly fair action in P.

Returning to the example of the FIFO buffer, if the actions *read* and *write* are weakly fair, and the *unload* action for the buffer environment is weakly fair, then in any fair computation of this transition system it is always the case that a state in which $x = in \neq \perp$ is eventually followed by (i.e., *leads to*, in the terminology of temporal logic) a state in which $out = x$.

7 Discussion

We have argued thus far that lazy composition is superior to the assume-guarantee method for compositional verification on the grounds that:

1. Lazy composition employs proof methods that are already familiar whereas the assume-guarantee proof rule is quite formidable.
2. Assume-guarantee methods require specifications that can anticipate future environment constraints.
3. The assume-guarantee assumptions apply to both component and environment and it is awkward to restrict these so that they only constrain the environment.
4. Assume-guarantee specifications are more appropriate for writing blackbox characterizations of open components rather than for compositional verification where the point is to achieve a useful decomposition of the verification task.

The advantage of lazy composition with respect to non-compositional, global reasoning as characterized by the Owicki–Gries approach [OG76] is that it combines the simplicity of global reasoning with the economy of using an abstract characterization of the environment rather than the actual components in the environment. This abstract characterization can be used to prove a number of component properties. The actual components can then be shown to conform to this abstract characterization by means of a refinement proof.

The Owicki–Gries approach is subsumed by lazy composition. If P is a component that is required to satisfy an invariant p, then we can take the environment E to be the transition system that merely preserves p, i.e., $\vdash N_E(s, s') \wedge p(s) \supset p(s')$. Then the refinement proof obligation reduces to a global demonstration that each component that is composed with P preserves the invariant. This is obviously the most general assumption one can make of an environment to P given that one wants to establish the invariant p, but it is not the optimal way to use lazy composition. The more appropriate use of lazy composition is by describing the allowed or intended environment actions that are relevant to the state variables that are read or written by component P and that are needed to obtain useful properties of P. Thus lazy composition modularizes the global reasoning by identifying suitable abstractions for the environment of each component.

7.1 Other Applications of Lazy Composition

We have employed lazy composition in the verification of the safety properties of an N-process mutual exclusion algorithm [Sha97] and the alternating-bit communication protocol [BSW69]. These verifications have been carried out using PVS [ORS92]. The mutual exclusion algorithm has been verified using a combination of induction, abstraction, and model checking. The algorithm uses a Boolean *turn* variable for each process to arbitrate access to successive rounds of competition using 2-process mutual exclusion, for eventual access to the critical section. The environment to each process has to be constrained to not affect the value of this *turn* variable in an undesirable way, e.g., when a process has checked the *turn* value and has entered its critical section.

The example of the alternating-bit protocol consists of a sender process, a receiver process, and the message and acknowledgement channels. The sender process constrains its environments merely to drop messages from the message channel, and the receiver process similarly constrains the value of the acknowledgement channel. With these constraints, it is possible to carry out a modular verification of the safety property of the alternating-bit protocol where all the invariants are proved solely by local reasoning in terms of the receiver or the sender process, possibly using previously proved global invariants.

8 Conclusions

We have presented the details of the paradigm of lazy compositional verification. This approach has several advantages over the assume-guarantee paradigm. We have formalized lazy composition verification within PVS [ORS92] and verified several medium-scale examples with this approach. We do not yet have any conclusive evidence that the method scales up to larger systems. Lazy composition can be adapted to models other than asynchronous transition systems by suitably altering the definitions of composition, conjunction, and refinement. Lazy composition does not need any new verification machinery since it builds on existing techniques for proving safety, liveness, and refinement properties.

References

[AH96] Rajeev Alur and Thomas A. Henzinger. Reactive modules. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 207–218, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[AL93] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[AL95] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[AP93] Martín Abadi and Gordon D. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, 1993.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[Bar85] H. Barringer. *A Survey of Verification Techniques for Parallel Programs*, volume 191 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260, 261, May 1969.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[CMP94] Edward Chang, Zohar Manna, and Amir Pnueli. Compositional verification of real-time systems. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 458–465, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[Col93] P. Collette. Application of the composition principle to Unity-like specifications. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings of TAPSOFT '93*, volume 668 of *Lecture Notes in Computer Science*, pages 230–242, Berlin, 1993. Springer-Verlag.

[Col94] Pierre Collette. An explanatory presentation of composition rules for assumption-commitment specifications. *Information Processing Letters*, 50(1):31–35, April 1994.

[dBdRR90] J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

[dBdRR94] J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, Noordewijkerhout, The Netherlands, 1994. Springer Verlag.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 16, pages 995–1072. Elsevier and MIT press, Amsterdam, The Netherlands, and Cambridge, MA, 1990.

[GL94] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1985.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[Hoo91] J. Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[Jos90] B. Josko. Verifying the correctness of AADL modules using model checking. In de Bakker et al. [dBdRR90], pages 386–400.

[Kur93] R.P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes.* Princeton University Press, Princeton, NJ, 1993.

[KV96] O. Kupferman and M. Y. Vardi. Module checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification96*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86. Springer Verlag, 1996.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth Annual Symposium on Principles of Distributed Computing, New York*, pages 137–151. ACM Press, 1987.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification.* Springer-Verlag, New York, NY, 1992.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[PJ91] P. K. Pandya and M. Joseph. P–A logic — a compositional proof system for distributed programs. *Distributed Computing*, 5(1):37–54, 1991.

[Pnu84] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logic and Models of Concurrent Systems*, NATO-ASI, pages 123–144. Springer Verlag, 1984.

[Sch87] Fred B. Schneider. Decomposing properties into safety and liveness using predicate logic. Technical Report 87-874, Department of Computer Science, Cornell University, Ithaca, NY, October 1987.

[Sha93a] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.

[Sha93b] N. Shankar. A lazy approach to compositional verification. Technical Report SRI-CSL-93-8, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[Sha97] N. Shankar. Machine-assisted verification using theorem proving and model checking. In Manfred Broy and Birgit Scheider, editors, *Mathematical Methods in Program Development*, volume 158 of *NATO ASI Series F: Computer and Systems Science*, pages 499–528. Springer, 1997.

[Sta85] E. W. Stark. A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 369–391. Springer Verlag, 1985.

[XCC94] Q.-W. Xu, A. Cau, and P. Collette. On unifying assumption–commitment style proof rules for concurrency. In B. Jonsson and J. Parrow, editors, *CONCUR'94*, volume 836 of *Lecture Notes in Computer Science*, pages 267–282. Springer Verlag, 1994.

[XdRH97] Q.-W. Xu, W.-P. de Roever, and J.-F. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness*, volume 321 of *Lecture Notes in Computer Science*. Springer Verlag, 1989.