Local Liveness
for Compositional Modeling of Fair Reactive Systems*

Rajeev Alur[1] and Thomas A. Henzinger[2]**

[1] AT&T Bell Laboratories, Murray Hill, NJ
[2] Department of Computer Science, Cornell University, Ithaca, NY

Abstract. We argue that the standard constraints on liveness conditions in nonblocking trace models—machine closure for closed systems, and receptiveness for open systems—are unnecessarily weak and complex, and that liveness should, instead, be specified by augmenting transition systems with acceptance conditions that satisfy a *locality* constraint. First, locality implies machine closure and receptiveness, and thus permits the composition and modular verification of live transition systems. Second, while machine closure and receptiveness are based on infinite games, locality is based on repeated finite games, and thus easier to check. Third, no expressive power is lost by the restriction to local liveness conditions. We illustrate the appeal of local liveness using the model of *Fair Reactive Systems*, a nonblocking trace model of communicating processes.

1 Introduction

In the hierarchical development of systems, the designer models a system at different levels of abstraction. The verification task, then, is to prove an appropriate implementation relation $\preceq$ between successive levels. At each level, the system is described by combining communicating processes using operations such as parallel composition $\|$. In order to manage the complexity of proving that one level implements another level, algorithmic methods need to be supported with inference rules that circumvent the construction of product state spaces. The soundness of such verification rules depends on the underlying semantics.

We consider a linear semantics. The behavior of a process is a sequence of observations (partial states or events), called a *trace*, and the semantics of a process is a set of traces. Parallel composition corresponds to the intersection of trace sets (or conjunction), and implementation corresponds to the inclusion of trace sets (or logical implication). These definitions, which are adopted by a variety of formalisms (including I/O automata [13], TLA [11], trace theory [6], and Cospan [10]), support a *compositional* verification rule. Compositionality

---

* A preliminary version of this paper appeared in the *Proceedings of the Seventh International Conference on Computer-aided Verification* (CAV), *Lecture Notes in Computer Science* **939**, Springer-Verlag, 1995, pp. 166–179.

** Supported in part by the NSF grant CCR-9200794, by the AFOSR contract F49620-93-1-0056, and by the DARPA grant NAG2-892.

asserts that the implementation relation $\preceq$ is a congruence with respect to parallel composition. Then, to prove that the product system $S_1 \| S_2$ implements the specification $T_1 \| T_2$, where the T_i-processes are abstractions of the S_i-processes, it suffices to prove both $S_1 \preceq T_1$ and $S_2 \preceq T_2$ independently.

The compositional reasoning principle, while useful, may not always be applicable. In particular, S_1 may not implement T_1 for all environments, but only for environments that behave like S_2, and vice versa. In this case, *modular* verification rules—also called *assumption-guarantee* rules—are needed. A modular verification rule may assert that in order to prove $S_1 \| S_2 \preceq T_1 \| T_2$, it suffices to prove $S_1 \| T_2 \preceq T_1$ and $T_1 \| S_2 \preceq T_2$. Since the state spaces of the abstractions T_i are typically smaller than the state spaces of the implementations S_i, modular reasoning still avoids the bottleneck of constructing the product of S_1 and S_2. However, due to the apparent circularity inherent in the modular verification rule, the rule is sound only under additional semantic assumptions.

Consider, for example, the *inconsistent* process with the empty trace set. If the inconsistent process is a legal process, then the modular verification rule is trivially unsound. For *safe* processes, it can be ensured that the parallel composition of two consistent processes is again consistent by requiring that processes are *nonblocking*; that is, a process cannot be prevented, by any possible environment, from making a move.[3] The nonblocking requirement for processes has the added benefits that safety specifications can be executed in a stepwise fashion, and that safety specifications cannot be trivially implemented by the inconsistent process. For *live* processes, it can be ensured that the parallel composition of two consistent processes is again consistent by requiring that processes are *receptive* [1, 6, 8]; that is, a process cannot be prevented, by any possible environment, from achieving its liveness condition.[4] The receptiveness requirement for processes has the added benefits that live specifications can be executed in a stepwise fashion, and that the liveness condition can be ignored for proving safety properties of a live process.

The receptiveness requirement, while essential for modular reasoning in the trace model, has the drawback of being computationally hard to check. This is because, technically, receptiveness is a property of infinite games ("process versus environment"), and because receptive liveness conditions are not closed under intersection. In practice, therefore, receptiveness is ensured syntactically, by specifying liveness conditions using fairness constraints [11, 14]. The types and forms of fairness constraints, however, depend on the syntax of the specific process description language that is used. We propose a semantic restriction of receptiveness—called *local receptiveness*[5]—that, while abstract and more general, retains the benefits of fairness constraints. Indeed, our definitions may be

[3] For closed processes, which do not interact with any environment, the nonblocking requirement says that every reachable state has a successor state (*deadlock-freedom*).

[4] For closed live processes, the receptiveness requirement says that every finite execution can be extended to an infinite execution that satisfies the liveness condition (*machine-closure* [5]).

[5] For closed processes, the local restriction of machine closure is called *local liveness*.

viewed as game-theoretic version of strong fairness.

The justification of the locality restriction on liveness conditions can thus be summarized as follows. First, commonly used fairness constraints, such as weak and strong transition fairness, already are locally receptive. Second, local receptiveness implies receptiveness, and therefore ensures the stepwise executability and the modular verifiability of live specifications. Third, unlike receptiveness, local receptiveness is a property of finite games, and locally receptive liveness conditions are closed under intersection. The locality of liveness constraints can therefore be checked individually. Suppose, for instance, that a liveness condition on a finite-state structure is specified by k Streett constraints. While checking machine closure is quadratic in k and checking receptiveness requires time exponential in k, local liveness and local receptiveness can be checked in time linear in k.

We develop the locality concept using the model of *Fair Reactive Systems* [4], where each process communicates with its environment through interface variables that can be observed by other processes. A process cannot constrain the interface variables of other processes, and changes in the interface components of two different processes cannot causally depend on each other. Processes are combined using the two operations of parallel composition and variable hiding. The locality concept applies equally to other nonblocking trace models, such as I/O automata, which can be embedded into our model.

Section 2 studies local liveness for closed systems that are modeled as Kripke structures. Section 3 defines the model of Fair Reactive Systems, and studies local receptiveness for open systems that are modeled as fair reactive systems. Section 4 develops compositional and modular reasoning principles for fair reactive systems.

2 Locally Live Structures

Let Σ be a countable alphabet. A *Σ-language* $\Lambda \subseteq \Sigma^\omega$ is a set of infinite words with letters from Σ. By $pref(\Lambda) \subseteq \Sigma^*$ we denote the set of all finite prefixes of words in Λ. The Σ-language Λ is *safe* iff for all infinite words $\underline{s} \in \Sigma^\omega$, if all finite prefixes of $\underline{s}$ belong to $pref(\Lambda)$, then $\underline{s}$ belongs to Λ.

Structures Safe languages are generated by state-transition structures. A (*Kripke*) *structure* K consists of

(1) [*state space*] a countable set Σ of states,
(2) [*initial region*] a set $\sigma^I \subseteq \Sigma$ of initial states,
(3) [*transition relation*] a serial binary relation $\rightarrow \subseteq \Sigma^2$ on the states,[6]
(4) [*observation alphabet*] a countable set A of observations, and
(5) [*observation function*] a function $\langle\!\langle \cdot \rangle\!\rangle : \Sigma \rightarrow A$ that maps each state to an observation.

[6] Given a state $s \in \Sigma$, we write $post(s) = \{t \in \Sigma \mid s \rightarrow t\}$ for the set of successor states of s, $post^*(s) = (\cup i \geq 0.\, post^i(s))$ for the set of states that are reachable from s, and $Reach(K) = post^*(\sigma^I)$ for the set of states that are reachable from initial states. The relation $\rightarrow \subseteq \Sigma^2$ is *serial* iff for all states $s \in \Sigma$, $post(s)$ is nonempty.

A K-*execution* is a finite or infinite sequence $\underline{s} = s_0 s_1 s_2 \ldots$ of states $s_i \in \Sigma$ such that $s_0 \in \sigma^I$ and for all $i \geq 0$, $s_i \rightarrow s_{i+1}$. The K-execution $\underline{s}$ induces the K-*trace* $\langle\!\langle \underline{s} \rangle\!\rangle = \langle\!\langle s_0 \rangle\!\rangle \langle\!\langle s_1 \rangle\!\rangle \langle\!\langle s_2 \rangle\!\rangle \ldots$ The structure K defines the Σ-language $[K] \subseteq \Sigma^\omega$ of infinite K-executions and the A-language $[\![K]\!] \subseteq A^\omega$ of infinite K-traces. The Σ-languages $[K]$ is safe.[7]

Liveness conditions Unsafe languages are specified by structures with liveness conditions. A K-*liveness condition* $\Lambda \subseteq [K]$ is a set of infinite K-executions.[8] A *live structure* (K, Λ) consists of a structure K and a K-liveness condition Λ. The live structure (K, Λ) defines the A-language $[\![K, \Lambda]\!] = \langle\!\langle \Lambda \rangle\!\rangle$—the *trace set* of (K, Λ).

A K-*region* $\sigma \subseteq \Sigma$ is a set of states, a K-*action* $\alpha \subseteq \rightarrow$ is a set of transitions, and a K-*property* $L \subseteq \mathit{pref}([K])$ is a set of finite K-executions. The infinite K-execution $\underline{s}$ is σ-*recurrent* iff infinitely many states of $\underline{s}$ belong to σ; α-*recurrent* iff infinitely many transitions of $\underline{s}$ belong to α; and L-*recurrent* iff infinitely many prefixes of $\underline{s}$ belong to L. Liveness conditions are specified using boolean combinations of recurrence requirements. In this paper, we restrict ourselves to recurrence requirements that are defined by strong-fairness constraints and by Streett constraints.

A *strong-fairness* K-*constraint* is a K-action. The infinite K-execution $\underline{s}$ is α-*fair*, for the strong-fairness K-constraint α, iff either $\underline{s}$ is not $\mathit{dom}(\alpha)$-recurrent or $\underline{s}$ is α-recurrent, where $\mathit{dom}(\alpha) = \{s \in \Sigma \mid \exists t.\, (s,t) \in \alpha\}$ is the set of source states of α. Thus the strong-fairness K-constraint α requires that if the action α is enabled infinitely often, then the action α is taken infinitely often. A *Streett* K-*constraint* is a pair of K-regions. The infinite K-execution $\underline{s}$ is (σ, τ)-*fair*, for the Streett K-constraint (σ, τ), iff either $\underline{s}$ is not σ-recurrent or $\underline{s}$ is τ-recurrent.

The infinite K-execution $\underline{s}$ is F-*fair*, for a set F of K-constraints, iff $\underline{s}$ is fair for all constraints in F. The set F of K-constraints specifies the K-liveness condition Λ_F that contains all F-fair K-executions. We denote the live structure (K, Λ_F) by (K, F). The live structure (K, F) is a *strong-fairness structure* iff F is a finite set of strong-fairness K-constraints, and (K, F) is a *Streett structure* iff F is a finite set of Streett K-constraints.

Machine-closed liveness conditions For executable specifications, the liveness conditions are required to be machine-closed. The live structure (K, Λ) is *machine-closed* iff every finite K-execution can be extended to an (infinite) K-execution in Λ (i.e., $\mathit{pref}([K]) \subseteq \mathit{pref}(\Lambda)$). Machine closure rules out, for instance, the empty set as a liveness condition, and ensures that by executing the structure K, the liveness condition Λ cannot be violated in a finite number of transitions. It follows that that the liveness condition can be ignored for proving safety properties of machine-closed live structures.

[7] The A-language $[\![K]\!]$ is safe if the initial region σ^I is finite, and the transition relation $\rightarrow$ is *finitely branching*; that is, if for all states $s \in \Sigma$, $\mathit{post}(s)$ is finite.

[8] Sometimes a K-liveness condition is defined to be a set of infinite K-traces. The concepts of this paper apply also to this alternative definition of liveness.

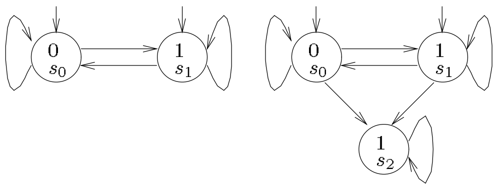

Fig. 1. The structures K_1 (left) and K_2 (right)

Consider a fixed structure K, and two K-liveness conditions Λ_1 and Λ_2. If the live structure (K, Λ_1) is machine-closed and $\Lambda_1 \subseteq \Lambda_2$, then (K, Λ_2) is also machine-closed. This implies that if $\Lambda_1 = \Lambda_F$ is specified by a set F of K-constraints, then machine closure is not destroyed by removing K-constraints from F. The machine-closed K-liveness conditions, however, are not closed under intersection. As an example, consider the structure K_1 of Figure 1, let $\Diamond\Box\, s_0 = (s_0 + s_1)^* s_0^\omega$ and $\Diamond\Box\, s_1 = (s_0 + s_1)^* s_1^\omega$. While the live structures $(K_1, \Diamond\Box\, s_0)$ and $(K_1, \Diamond\Box\, s_1)$ are machine-closed, the K_1-liveness condition $\Diamond\Box\, s_0 \cap \Diamond\Box\, s_1$ is empty.

Every strong-fairness structure (K, F) is machine-closed. For the structure K_1 of Figure 1, the K_1-liveness condition $\Diamond\Box\, s_1$ cannot be specified by strong-fairness K_1-constraints. Streett structures, on the other hand, may not be machine-closed. For instance, for every structure with state space Σ, the Streett constraint $(\Sigma, \emptyset)$ specifies the empty set. For the structure K_1 of Figure 1, the liveness condition $\Diamond\Box\, s_1$ can be specified by the Streett K_1-constraint $(\{s_0\}, \emptyset)$. If K is a structure with n states and m transitions, and F is a set of k Streett K-constraints, then the problem of checking if the Streett structure (K, F) is machine-closed requires time $O(k^2(m+n))$ [7].

Local liveness conditions Machine closure can be defined as a two-player game. Given a live structure (K, Λ), the adversary provides a finite K-execution $\underline{s}$, and the protagonist attempts to extend $\underline{s}$ to an infinite K-execution in Λ. The live structure (K, Λ) is machine-closed iff there exists a winning strategy for the protagonist. Now let us consider the game where the two players take turns to add *finite* extensions to the current K-execution. As before, the protagonist wins the game if the resulting infinite K-execution belongs to the liveness condition Λ. We call the live structure (K, Λ) locally live if the protagonist has a winning strategy in this new game, which is harder on the protagonist than the machine-closure game.

Formally, the live structure (K, Λ) is *locally live* iff there exists a K-property L_W—the *win property*—such that (1) every finite K-execution can be extended to a (finite) K-execution in L_W (i.e., $pref([K]) \subseteq pref(L_W)$), and (2) all L_W-recurrent infinite K-executions belong to Λ.

It follows immediately that every locally live structure is machine-closed. However, not every machine-closed live structure is locally live. As an example, recall the structure K_1 from Figure 1, and let $\Diamond\Box\, 1$ be the $\{0,1\}$-language $(0+1)^*1^\omega$. The live structure $(K_1, \Diamond\Box\, s_1)$ is not locally live. Indeed, there is no K_1-liveness condition Λ such that the live structure (K_1, Λ) is locally live and $[\![K_1, \Lambda]\!] = \Diamond\Box\, 1$. Now consider the three-state structure K_2 from Figure 1. The live structure $(K_2, \Diamond\Box\, s_2)$ is locally live and $[\![K_2, \Lambda]\!] = \Diamond\Box\, 1$. It is our thesis that the three-state structure K_2 more closely resembles executable specifications of the trace set $\Diamond\Box\, 1$ than does the two-state structure K_1. Also, local liveness conditions abstract and generalize the spirit of fairness constraints. For instance, the local K_2-liveness condition $\Diamond\Box\, s_2$ can be specified by the strong-fairness K_2-constraint $\{(s_0, s_2), (s_1, s_2)\}$.[9] For the two-state structure K_1, of the two local K_1-liveness conditions $\Box\Diamond\, s_1$ and $\Box\Diamond\, s_1 s_1 s_1$,[10] only the latter can be specified by strong-fairness K_1-constraints.

Consider a fixed structure K, and two K-liveness conditions Λ_1 and Λ_2. If the live structure (K, Λ_1) is locally live and $\Lambda_1 \subseteq \Lambda_2$, then (K, Λ_2) is also locally live. This implies that if $\Lambda_1 = \Lambda_F$ is specified by a set F of K-constraints, then locality is not destroyed by removing K-constraints from F. Moreover, the local K-liveness conditions are closed under intersection.

Proposition 1. *If (K, Λ_1) and (K, Λ_2) are locally live structures, then $(K, \Lambda_1 \cap \Lambda_2)$ is a locally live structure.*

This implies that the locality of a set F of K-constraints can be checked by checking the locality of each K-constraint in F separately, and that the locality of F is not destroyed by adding locally live K-constraints to F.

Every strong-fairness structure (K, F) is locally live. Since Streett structures may not be machine-closed, there are Streett structures that are not locally live. The following proposition gives a sufficient and necessary criterion for checking if a Streett constraint is locally live.

Proposition 2. *The Streett structure $(K, \{(\sigma, \tau)\})$ is locally live iff for all states $s \in \mathit{Reach}(K)$, there exists a state $t \in \mathit{post}^*(s)$ such that either $t \in \tau$ or $\mathit{post}^*(t) \cap \sigma$ is empty.*

From Proposition 1 it follows that the Streett structure (K, F) is locally live iff every Streett constraint in F satisfies the criterion of Proposition 2.

Finite-state structures A *Streett automaton* is a Streett structure with a finite state space. The trace sets of Streett automata are precisely the ω-regular languages. The following proposition shows that locality does not constrain the expressiveness of Streett automata.

Proposition 3. *Every ω-regular language is the trace set of a locally live Streett automaton.*

[9] Indeed, weak fairness suffices in this case.
[10] For $\Box\Diamond\, s_1 = (s_0^* s_1)^\omega$ and $\Box\Diamond\, s_1 s_1 s_1 = ((s_0 + s_1)^* s_1 s_1 s_1)^\omega$.

The structure $K = (\Sigma, \sigma^I, \rightarrow, A, \langle\!\langle\cdot\rangle\!\rangle)$ is *deterministic* if for all states $s, t, r \in \Sigma$, if $s \rightarrow t$ and $s \rightarrow r$ and $t \neq r$, then $\langle\!\langle t\rangle\!\rangle \neq \langle\!\langle r\rangle\!\rangle$. While deterministic Streett automata can define all ω-regular languages, the ω-regular $\{0,1\}$-language $\Diamond\Box\, 1$ is not definable by a deterministic Streett automaton that is locally live. Indeed, if K is a deterministic structure, not necessarily finite-state, and (K, Λ) is locally live, then $[\![K, \Lambda]\!] \neq \Diamond\Box\, 1$. Let **DLStreett** be the class of languages that are defined by deterministic and Streett automata that are locally live, and let **DBüchi** be the class of languages that are defined by deterministic Büchi automata. The class **DBüchi** is a proper subset of the ω-regular languages.

Proposition 4. *The class* **DLStreett** *is a proper superset of* **DBüchi***, and a proper subset of the ω-regular languages. The class* **DLStreett** *is closed under union and intersection, but not under complement.*

For example, the $\{0,1,2\}$-language "if $\Box\Diamond\, 2$ then $\Box\Diamond\, 1$" is in **DLStreett** but not in **DBüchi**.

Propositions 1 and 2 suggest an algorithm for checking if a Streett automaton is locally live. While checking the machine closure of a Streett automaton requires quadratic time in the number of Streett constraints, the local liveness of a Streett automaton can be checked in linear time, because for local liveness all constraints can be checked individually.

Proposition 5. *Let (K, F) be a Streett automaton with n states, m transitions, and k Streett constraints. It can be checked in time $O(k(m+n))$ if (K, F) is locally live.*

3 Fair Reactive Systems

This section introduces the model of *Fair Reactive Systems* [4], which equips Kripke structures with a communication interface.

3.1 Safe Reactive Systems

A reactive system is situated within an environment. Some changes of the environment are visible to the system and cause reactions by the system. Other system actions are independent of the environment.

Variables A reactive system operates on a finite set X of typed *system variables*. The *ownership record* $\overset{---}{X}$ partitions the set X into three disjoint subsets:

- $\overset{-\cdot\cdot}{X}$ —the set of *private variables*. Each private variable can be read and modified by the system, and neither read nor modified by the environment.
- $\overset{\cdot-\cdot}{X}$ —the set of *interface variables*. Each interface variable can be read by both the system and the environment, and modified by the system only.
- $\overset{\cdot\cdot-}{X}$ —the set of *external variables*. Each external variable can be read by both the system and the environment, and modified by the environment only.

The variables in $\overset{--\cdot}{X} = \overset{-\cdot\cdot}{X} \cup \overset{\cdot-\cdot}{X}$, which are owned by the system, are called *local*; the variables in $\overset{\cdot--}{X} = \overset{\cdot-\cdot}{X} \cup \overset{\cdot\cdot-}{X}$, which are visible to the environment, are called *observable*.

States A *state* is a valuation for the set X of system variables. The set Σ_X of states is called the *state space* of the system (the subscript is suppressed when the set of system variables is understood). Each state s consists of a private state $\overset{-\cdot\cdot}{s}$, an interface state $\overset{\cdot-\cdot}{s}$, and an external state $\overset{\cdot\cdot-}{s}$. We write $\overset{-\cdot\cdot}{\Sigma}$ for the set private states, etc. A *local state* consists of a private state and an interface state; an *observation* consists of an interface state and an external state. Finitely many local states in $\overset{--\cdot}{\Sigma}$ are declared to be *initial*; the system behaves correctly only if started in an initial local state. The set of initial states is denoted by *Init*.

Moves A reactive system proceeds in discrete moves. Each system move consists of an active phase followed by a reactive phase. During the *active phase*, the system updates the interface state and, independently, the environment updates the external state. During the *reactive phase*, the system updates the private state depending on the result of the previous active phase. Thus, the system cannot constrain the update of the external state, and the update of the interface state does not depend on the update of the external state. The system moves are formally specified by two functions. The *action function* $Act: \Sigma \rightarrow \mathbf{2}^{\overset{\cdot-\cdot}{\Sigma}}$ defines for each state s, a finite and nonempty set $Act(s)$ of possible interface successor states. The *reaction function* $React: \Sigma \times \overset{\cdot--}{\Sigma} \rightarrow \mathbf{2}^{\overset{\cdot-\cdot}{\Sigma}}$ defines for each state s and each updated observation $\overset{\cdot--}{t}$ such that $\overset{\cdot-\cdot}{t} \in Act(s)$, a finite and nonempty set of possible private successor states $\overset{-\cdot\cdot}{t} \in React(s, \overset{\cdot--}{t})$.

Systems A *reactive system* S consists of (1) an ownership record $\overset{---}{X}$ for a finite set X of system variables, (2) a finite set $Init \subseteq \overset{--\cdot}{\Sigma}$ of initial local states, (3) an action function Act, and (4) a reaction function $React$. The reactive system S defines the structure $K_S = (\Sigma_X, \sigma^I, \rightarrow, \overset{\cdot--}{\Sigma}, \overset{\cdot--}{\cdot})$, where $s \in \sigma^I$ iff $\overset{--\cdot}{s} \in Init$, and $s \rightarrow t$ iff $\overset{\cdot-\cdot}{t} \in Act(s)$ and $\overset{-\cdot\cdot}{t} \in React(s, \overset{\cdot--}{t})$.

Reactive systems satisfy two nonblocking properties. First, the system does not constrain the update of the external state (i.e., for every state s, $\overset{\cdot\cdot-}{post}(s) = \overset{\cdot\cdot-}{\Sigma}$). Second, the updated interface state does not depend on the updated external state (i.e., for every state s, $\overset{\cdot--}{post}(s) = \overset{\cdot-\cdot}{post}(s) \times \overset{\cdot\cdot-}{post}(s)$). The former property is called *environment enableness*, the latter *interface independence*. To motivate interface independence, consider a system with the boolean interface variable y and the boolean external variable z. Without interface independence, the system could vow to always keep the value of y equal to the value of z. Symmetrically, the environment could vow to always keep the value of z different from the value of y. This would result in an inconsistent specification.

It is a routine exercise to model different programming-language constructs as reactive systems. For example, *asynchronous* systems, where the updates of the interface state and the external state proceed at independent speeds, are a special case of reactive systems (those with $\overset{\cdot-\cdot}{s} \in Act(s)$ for all states s). Consequently, asynchronous models such I/O automata have a natural embedding into reactive systems. We refer the reader to [4] for a discussion of modeling issues and examples.

An *S-execution* is a K_S-execution, and we write $[S] = [K_S]$ for the set of infinite S-executions. The definitions of regions, actions, properties, liveness conditions, and Streett constraints for structures apply to reactive systems also. For instance, an S-*property* $L \subseteq \mathit{pref}([S])$ is a set of finite S-executions, and an S-*liveness condition* $\Lambda \subseteq [S]$ is a set of infinite S-executions.

3.2 Live Reactive Systems

A *live reactive system* (S, Λ) consists of a reactive system S and an S-liveness condition Λ.

Receptive liveness conditions For open systems, the counterpart of machine closure is receptiveness. Informally, the live reactive system (S, Λ) is receptive if, no matter how the environment behaves, the system can guarantee to generate an infinite S-execution that meets the liveness condition Λ. The receptiveness of (S, Λ) is best understood as an infinite two-player game. The protagonist attempts to produce an infinite S-execution in Λ, while the adversary tries to prevent this. Initially, the adversary chooses a finite S-execution. At each step, the protagonist chooses a new interface state, the adversary chooses a new external state, and then the protagonist extends the current execution by choosing a new private state. The protagonist wins the game iff the resulting infinite S-execution belongs to Λ. The live reactive system (S, Λ) is receptive iff the protagonist has a winning strategy.

Consider, for example, the reactive system S_0 with a boolean private variable x, a boolean interface variable y, and a boolean external variable z. The action function is defined by $\mathit{Act}(x_0 y_0 z_0) = \{y_0, z_0\}$, and the reaction function is defined by $\mathit{React}(x_0 y_0 z_0, y_1 z_1) = \{x_0, z_1\}$; that is, during each active phase, the interface variable y either stays unchanged or receives the old value of z, and during each reactive phase, the private variable x either stays unchanged or receives the new value of z. Some nonreceptive S_0-liveness conditions are $\Box\Diamond\, z = 0$ ("infinitely often $z = 0$"), $\Box\Diamond\, y = 0$, and $\Box\Diamond\, y = z$. Some receptive S_0-liveness conditions are $\Box\Diamond\, x = z$, if $\Box\Diamond\, z = 0$ then $\Box\Diamond\, y = 0$, and $\Diamond\Box\, y = 0$ ("eventually always $y = 0$").

The receptiveness game can be formalized as follows.[11] Let S be a reactive system. An S-*strategy* is a partial function h that maps a finite S-execution $\underline{s}_{0..i}$ and an observation $\overline{t}$ to a pair $(\overline{u}, \overline{v})$ consisting of a private state $\overline{u} \in \mathit{React}(s_i, \overline{t})$ and an interface state $\overline{v} \in \mathit{Act}(\overline{t} \cup \overline{u})$. Note that the strategy need not always be defined. Furthermore, we have merged the update of the private state during a reactive phase with the update of the interface state during the subsequent active phase. The S-strategy h is *enabled* at position i of the infinite S-execution $\underline{s} = s_0 s_1 s_2 \ldots$ iff $h(\underline{s}_{0..i}, \overline{s}_{i+1})$ is defined, and h is *played* at the $\underline{s}$-position i iff $(\overline{s}_{i+1}, \overline{s}_{i+2}) = h(\underline{s}_{0..i}, \overline{s}_{i+1})$. If a strategy is enabled infinitely often, then starting at infinitely many of these positions, the strategy is played continuously unless it becomes disabled. The infinite S-execution $\underline{s}$ is an *outcome* of the S-strategy h iff either h is enabled at only finitely many $\underline{s}$-positions or for infinitely many $\underline{s}$-positions i, either h is played at all $\underline{s}$-positions $j \geq i$ or there is a $\underline{s}$-position $k > i$ such that h is disabled at the $\underline{s}$-position k

[11] A simpler formalization is possible for defining receptiveness. Our definitions are chosen because they lead to a natural locality restriction.

and played at all $\underline{s}$-positions $i \leq j < k$. The S-strategy h is *winning* for the S-liveness condition Λ iff all outcomes of h belong to Λ. The live reactive system (S, Λ) is *receptive* iff there exists a winning S-strategy for Λ.

Observe that if the live reactive system (S, Λ) is receptive, then the underlying live structure (K_S, Λ) is machine-closed. As with machine closure, the receptive S-liveness conditions of a reactive system S are not closed under intersection; that is, the receptiveness of the live reactive systems (S, Λ_1) and (S, Λ_2) does not imply the receptiveness of $(S, \Lambda_1 \cap \Lambda_2)$.

Local receptiveness We now define the open analog of locally live structures. In the local-receptiveness game, the protagonist is allowed to apply its strategy only for infinitely many finite stretches. For a reactive system S, the *S-strategy* h is *local* iff there is no infinite S-execution $\underline{s}$ such that for some $\underline{s}$-position i, the strategy h is played at all $\underline{s}$-positions $j \geq i$. Thus, if a local S-strategy is continuously played, then it must eventually disable itself. It follows that the infinite S-execution $\underline{s}$ is an outcome of the local S-strategy h iff either h is enabled at only finitely many $\underline{s}$-positions or for infinitely many $\underline{s}$-positions i, there is a $\underline{s}$-position $k > i$ such that h is disabled at the $\underline{s}$-position k and played at all $\underline{s}$-positions $i \leq j < k$. The live reactive system (S, Λ) is *locally receptive* iff there exist finitely many S-liveness conditions $\Lambda_1, \ldots, \Lambda_k$ such that $(\cap 1 \leq i \leq k.\, \Lambda_i) \subseteq \Lambda$ and for each $1 \leq i \leq k$, there exists a winning local S-strategy for Λ_i.

For the reactive system S_0, the liveness condition $\Box\Diamond\, x = z$ is locally receptive. The S_0-liveness condition $\Diamond\Box\, x = z$ is receptive, but not locally receptive. Let Λ_0 be the requirement "if $\Box\Diamond\, z = 0$ then $\Box\Diamond\, x = z = 0$," and let Λ_1 be the requirement "if $\Box\Diamond\, z = 1$ then $\Box\Diamond\, x = z = 1$." There is a winning local S_0-strategy for Λ_0, and there is a winning local S_0-strategy for Λ_1. However, there is no single local S_0-strategy that wins for $\Lambda_0 \cap \Lambda_1$.

As with local liveness, our definition of local receptiveness ensures that the locally receptive S-liveness conditions of a reactive system S are closed under intersection.

Proposition 6. *The live reactive system $(S, \Lambda_1 \cap \Lambda_2)$ is locally receptive iff both (S, Λ_1) and (S, Λ_2) are locally receptive.*

Fair reactive systems The liveness condition for a reactive system can be specified using strong-fairness constraints for active and reactive phases.[12] Let S be a reactive system with the state space Σ. An *active strong-fairness S-constraint* is a subset of $\Sigma \times \overline{\Sigma}$. For the active strong-fairness S-constraint γ, the set of source states is $dom(\gamma) = \{s \in \Sigma \mid \exists \overline{t}\,.(s, \overline{t}) \in \gamma\}$. The infinite S-execution $\underline{s} = s_0 s_1 s_2 \ldots$ is *γ-recurrent* iff $(s_i, \overline{s}_{i+1}) \in \gamma$ for infinitely many positions i, and $\underline{s}$ is *γ-fair* iff either $\underline{s}$ is not $dom(\gamma)$-recurrent or $\underline{s}$ is γ-recurrent.

A *reactive strong-fairness S-constraint* is an S-action. The reactive strong-fairness constraint $\alpha \subseteq \Sigma^2$ is *enabled* at position i of the infinite S-execution $\underline{s}$ iff

[12] Weak-fairness constraints can be added to reactive systems in a similar fashion.

$(s_i, \overline{s}_{i+1} \cup \overline{t}) \in \alpha$ for some private state $\overline{t}$. The infinite S-execution $\underline{s}$ is α-*fair* iff either α is enabled at only finitely many positions of $\underline{s}$, or $\underline{s}$ is α-recurrent.

A *fair reactive system* consists of a reactive system S and a finite set F of active and reactive strong-fairness S-constraints. The infinite S-execution $\underline{s}$ is F-*fair* iff $\underline{s}$ is fair for all constraints in F. The fair reactive system (S, F) specifies the live reactive system (S, Λ_F), where Λ_F is the set of F-fair S-executions. Every fair reactive system is locally receptive.

Streett reactive systems A *Streett reactive system* (S, F) consists of a reactive system S and a finite set F of Streett S-constraints. The corresponding live reactive system (S, Λ_F) is defined as in the case of structures. The problem of checking if a finite-state Streett reactive system is receptive is computationally hard (exponential in the number of states) [3, 12]. By contrast, it can be checked in polynomial time if a finite-state Streett reactive system is locally receptive.

First, each Streett constraint can be checked individually for local receptiveness. Second, the locality check of a single Streett constraint can be set up as a finite game. Let S be a finite-state reactive system, and consider the Streett S-constraint (σ, τ). The start positions for the game are the states in σ that are reachable from initial states of S. The winning positions are the states in τ and the states from which no state in σ can be reached. At each step, the protagonist chooses an interface state, the adversary chooses an external state, and the protagonist updates the private state. The Streett reactive system $(S, \{(\sigma, \tau)\})$ is locally receptive iff the protagonist has a strategy that leads to a winning position. The complexity of solving such an and-or game is quadratic in the size of the underlying structure.

Proposition 7. *Let (S, F) be a Streett reactive system with n states, m transitions, and k Streett constraints. It can be checked in time $O(k \cdot m \cdot n)$ if (S, F) is locally receptive.*

4 Compositional and Modular Verification

The live reactive system (S, Λ) defines the *trace set* $[\![S, \Lambda]\!] = \overline{\Lambda}$. Let $\mathcal{S}$ be a live reactive system with the ownership record $\overline{X}$, and let $\mathcal{T}$ be a live reactive system with the ownership record $\overline{Y}$. The two systems $\mathcal{S}$ and $\mathcal{T}$ are *comparable* iff $\overline{X} = \overline{Y}$ and $\ddot{X} = \ddot{Y}$. The system $\mathcal{S}$ *implements* the system $\mathcal{T}$, denoted $\mathcal{S} \preceq \mathcal{T}$, iff $\mathcal{S}$ is comparable with $\mathcal{T}$ and $[\![\mathcal{S}]\!] \subseteq [\![\mathcal{T}]\!]$. The implementation relation—trace inclusion—is central to the hierarchical verification of systems.

4.1 Operations on Reactive Systems

We combine reactive systems using parallel composition and variable hiding.

Parallel composition Two reactive systems and are *compatible* iff they have disjoint sets of interface variables. Let $S_1 = (\overline{X}_1, \mathit{Init}_1, \mathit{Act}_1, \mathit{React}_1)$ and $S_2 = (\overline{X}_2, \mathit{Init}_2, \mathit{Act}_2, \mathit{React}_2)$ be two compatible reactive systems. We assume that the private variables of S_1 are disjoint from the system variables of S_2, and vice versa (this can be achieved by renaming private variables). The *parallel composition* $S_1 \| S_2$ is the reactive system $(X, \mathit{Init}, \mathit{Act}, \mathit{React})$:

Private variables: $\overset{-\cdot\cdot}{X} = \overset{-\cdot\cdot}{X}_1 \cup \overset{-\cdot\cdot}{X}_2$;
Interface variables: $\overset{\cdot-\cdot}{X} = \overset{\cdot-\cdot}{X}_1 \cup \overset{\cdot-\cdot}{X}_2$;
External variables: $\overset{\cdot\cdot-}{X} = (\overset{\cdot\cdot-}{X}_1 \cup \overset{\cdot\cdot-}{X}_2) - \overset{\cdot-\cdot}{X}$;
Initial states: $\overset{--\cdot}{s} \in \mathit{Init}$ iff $\overset{--\cdot}{X}_1[s] \in \mathit{Init}_1$ and $\overset{--\cdot}{X}_2[s] \in \mathit{Init}_2$;[13]
Action function: $\overset{\cdot-\cdot}{t} \in \mathit{Act}(s)$ iff both $\overset{\cdot-\cdot}{X}_1[t] \in \mathit{Act}(X_1[s])$ and $\overset{\cdot-\cdot}{X}_2[t] \in \mathit{Act}(X_2[s])$;
Reaction function: $\overset{-\cdot\cdot}{t} \in \mathit{React}(s, \overset{\cdot--}{t})$ iff both $\overset{-\cdot\cdot}{X}_1[t] \in \mathit{React}(X_1[s], \overset{\cdot--}{X}_1[t])$ and $\overset{-\cdot\cdot}{X}_2[t] \in \mathit{React}(X_2[s], \overset{\cdot--}{X}_2[t])$.

Two live reactive systems are *compatible* iff the underlying reactive systems are compatible. Let $\mathcal{S}_1 = (S_1, \Lambda_1)$ and $\mathcal{S}_2 = (S_2, \Lambda_2)$ be two compatible live reactive systems. The *parallel composition* $\mathcal{S}_1 \| \mathcal{S}_2$ is the live reactive system $(S_1 \| S_2, \Lambda_1 \| \Lambda_2)$, where the liveness condition $\Lambda_1 \| \Lambda_2$ consists of all infinite $(S_1 \| S_2)$-executions $\underline{s}$ such that $X_1[\underline{s}] \in \Lambda_1$ and $X_2[\underline{s}] \in \Lambda_2$.

Proposition 8. *For two compatible live reactive systems $\mathcal{S}_1$ and $\mathcal{S}_2$, if $\mathcal{S}_1$ and $\mathcal{S}_2$ are (locally) receptive, then $\mathcal{S}_1 \| \mathcal{S}_2$ is (locally) receptive.*

Notice that parallel composition corresponds to intersection. If the underlying transition relations of the component systems S_1 and S_2 are specified by two transition predicates over unprimed and primed variables (representing current and new values of system variables), then the conjunction of both transition predicates specifies the transition relation of the composed system $S_1 \| S_2$. If the liveness conditions of the component systems are specified by two sets of strong-fairness or Streett constraints, then the union of both constraint sets specifies the liveness condition of the composed system.

Variable hiding After parallel composition, some interface variables of the component systems may no longer be visible to the environment. This is achieved through the hiding of interface variables. Let Y be a set of variables. Given a reactive system $S = (\overset{---}{X}, \mathit{Init}, \mathit{Act}, \mathit{React})$, by *hiding* the variables in Y we obtain the reactive system $S \backslash Y = (\overset{---}{Z}, \mathit{Init}, \mathit{Act}, \mathit{React})$, where $\overset{-\cdot\cdot}{Z} = \overset{-\cdot\cdot}{X} \cup (Y \cap \overset{\cdot-\cdot}{X})$, $\overset{\cdot-\cdot}{Z} = \overset{\cdot-\cdot}{X} - Y$, and $\overset{\cdot\cdot-}{Z} = \overset{\cdot\cdot-}{X}$. The hiding operation leaves the liveness condition of a live reactive system unchanged.

4.2 Verification Rules

Compositional reasoning The compositional verification rule states that the implementation relation for live reactive systems is a congruence with respect to parallel composition and variable hiding.

Theorem 9. *Let $\mathcal{S}$ and $\mathcal{T}$ be two live reactive systems such that $\mathcal{S} \preceq \mathcal{T}$. Then for every live reactive system $\mathcal{U}$ compatible with $\mathcal{S}$, $\mathcal{U}$ is compatible with $\mathcal{T}$ and $\mathcal{S} \| \mathcal{U} \preceq \mathcal{T} \| \mathcal{U}$; and for every set X of variables, $\mathcal{S} \backslash X \preceq \mathcal{T} \backslash X$.*

[13] For a state $s \in \Sigma_X$ and a set $Y \subseteq X$ of variables, by $Y[s] \in \Sigma_Y$ we denote the projection of s to the variables in Y.

The soundness of the compositional verification rule depends on the facts that parallel composition corresponds to trace intersection (or conjunction), variable hiding is trace projection (or existential quantification), and implementation is trace inclusion (or logical implication).

Modular (assumption-guarantee) reasoning Modular verification rules have been presented for a variety of specific notations [2, 9, 15, 16, 17]. We present a simple and powerful rule for receptive live reactive systems. Consider two compatible live reactive systems $\mathcal{S}_1$ and $\mathcal{S}_2$, an abstraction $\mathcal{T}_1$ of $\mathcal{S}_1$, and an abstraction $\mathcal{T}_2$ of $\mathcal{S}_2$. We wish to prove that the parallel composition $\mathcal{S}_1 \| \mathcal{S}_2$ implements the abstraction $\mathcal{T} = \mathcal{T}_1 \| \mathcal{T}_2$. The modular verification rule states that it suffices to prove that $\mathcal{S}_1$ with the environment $\mathcal{T}_2$ implements $\mathcal{T}$, and $\mathcal{S}_2$ with the environment $\mathcal{T}_1$ implements $\mathcal{T}$, provided that all liveness conditions are receptive and there is no circularity in the use of the liveness conditions. The breaking of such circularities leads to an asymmetric rule.

Theorem 10. *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two compatible receptive live reactive systems, and let $\mathcal{T}_1$ and $\mathcal{T}_2 = (T_2, \Lambda)$ be two receptive live reactive systems such that $\mathcal{T}_1$ is comparable with $\mathcal{S}_1$, and $\mathcal{T}_2$ is comparable with $\mathcal{S}_2$. Let $\mathcal{T}_2^* = (T_2, \Lambda^*)$, where Λ^* is the set of infinite T_2-executions. If $\mathcal{S}_1 \| \mathcal{T}_2^* \preceq \mathcal{T}_1 \| \mathcal{T}_2^*$ and $\mathcal{T}_1 \| \mathcal{S}_2 \preceq \mathcal{T}_1 \| \mathcal{T}_2$, then $\mathcal{S}_1 \| \mathcal{S}_2 \preceq \mathcal{T}_1 \| \mathcal{T}_2$.*

The soundness of the modular verification rule depends on the facts that every reactive system is environment-enabling and interface-independent, that every reactive system has only finitely many initial local states, and that in each state, there are only finitely many possible updates of the local state in response to an update of the external state (there may be infinitely many possible updates of the external state). Without any one of these conditions, the rule fails already for safe systems. The restriction, in one hypothesis of the rule, to the safety part of $\mathcal{T}_2$ breaks any potential circularity in the use of the liveness conditions, and without this restriction, the rule fails. The rule also fails without the receptiveness assumptions.

We conclude by illustrating an application of the modular verification rule. Look at Figure 2. The fair reactive system $\mathcal{S}_1$ has two boolean interface variables *msg* and *tick*, a boolean external variable *ack*, and a private location variable that ranges over the nodes of the $\mathcal{S}_1$-graph in Figure 2. The initial location is marked with an incoming arrow; the interface variables may have any initial value. An edge labeled with an interface variable, say *msg*, corresponds to complementing the value of *msg*. An edge labeled with the external variable *ack* corresponds to updating the location variable when the environment complements *ack*. Each location has an implicit self-loop that represents internal computation. For the edges labeled with underlined interface variables, weak fairness is assumed. Thus, in the start location, the system repeatedly performs an internal computation without state change until it moves to the top location when the environment changes *ack*, or to the bottom location complementing *msg*. The weak fairness of the latter move ensures that the system cannot stay in the start location forever. Similar conventions are used in the graphical representations of the other

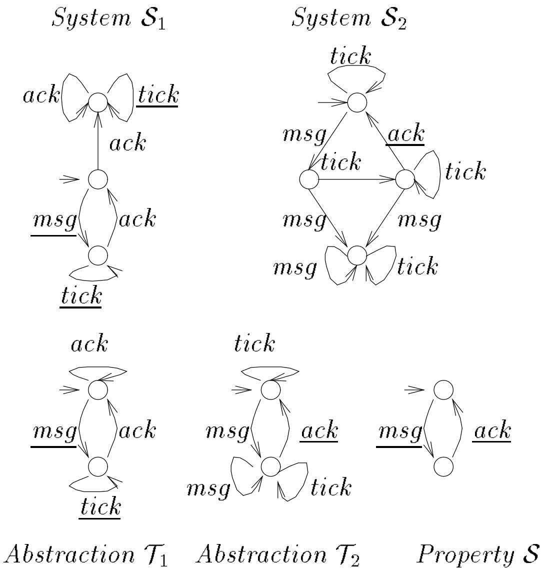

Fig. 2. Example of modular reasoning

systems. For the fair reactive system $\mathcal{S}_2$, *ack* is an interface variable, and *tick* and *msg* are external variables. The fair reactive system $\mathcal{T}_1$ is comparable with $\mathcal{S}_1$, and $\mathcal{T}_2$ is comparable with $\mathcal{S}_2$. The fair reactive system $\mathcal{S}$ has two interface variables, *msg* and *ack*.

We wish to prove that the composed system $\mathcal{S}_1 \| \mathcal{S}_2$ implements the property $\mathcal{S}$ (for type compatibility, we assume that *tick* is also an interface variable of $\mathcal{S}$, and the update of *tick* is unconstrained by $\mathcal{S}$). The systems $\mathcal{T}_1$ and $\mathcal{T}_2$ can be viewed as decompositions of the property $\mathcal{S}$. By transitivity, it suffices to prove $\mathcal{S}_1 \| \mathcal{S}_2 \preceq \mathcal{T}_1 \| \mathcal{T}_2$ and $\mathcal{T}_1 \| \mathcal{T}_2 \preceq \mathcal{S}$. The proof of the first obligation is simplified by applying the modular verification rule. First we need to show that $\mathcal{S}_1$ implements $\mathcal{T}_1$. This cannot be proved without assumptions regarding the environment, but it suffices to use only the safety part T_2 of $\mathcal{T}_2$; that is, $\mathcal{S}_1 \| T_2 \preceq \mathcal{T}_1 \| T_2$ holds. Next, we want to prove that $\mathcal{S}_2$ implements $\mathcal{T}_2$. For this purpose, we need both safety and liveness assumptions regarding the environment (the necessary liveness assumption is that an update to *msg* is eventually followed by an update to *tick*). The implementation $\mathcal{T}_1 \| \mathcal{S}_2 \preceq \mathcal{T}_1 \| \mathcal{T}_2$ holds, and thus the conclusion of the modular verification rule supplies $\mathcal{S}_1 \| \mathcal{S}_2 \ \preceq\ \mathcal{T}_1 \| \mathcal{T}_2$.

Acknowledgements. We thank Robert Kurshan and Mihalis Yannakakis for helpful discussions.

References

[1] M. Abadi, L. Lamport. Composing specifications. *ACM TOPLAS*, 15(1):73–132, 1993.

[2] M. Abadi, L. Lamport. *Conjoining Specifications*. Technical Report 118, DEC-SRC, 1993.

[3] M. Abadi, L. Lamport, P. Wolper. Realizable and unrealizable specifications of reactive systems. *Automata, Languages, and Programming*, LNCS 372, pp. 1–17. Springer, 1989.

[4] R. Alur, T.A. Henzinger. *Fair Reactive Systems*. Technical Report, Computer Science Department, Cornell University, 1995.

[5] K. Apt, N. Francez, S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2(4):226–241, 1988.

[6] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. MIT Press, 1989.

[7] E.A. Emerson, C. Lei. Modalities for model checking: branching time strikes back. *Symp. Principles of Programming Languages*, pp. 84–95. ACM, 1985.

[8] R. Gawlick, R. Segala, J. Sogaard-Andersen, N. Lynch. *Liveness in timed and untimed systems*. Technical Report MIT/LCS/TR-587, MIT, 1993.

[9] O. Grumberg, D. Long. Model checking and modular verification. *ACM TOPLAS*, 16(3):843–871, 1994.

[10] R. Kurshan. *Computer-aided Verification: The Automata-theoretic Approach*. Princeton University Press, 1994.

[11] L. Lamport. *The Temporal Logic of Actions*. Technical Report 79, DEC-SRC, 1991.

[12] H. Lescow. On polynomial-size programs winning finite-state games. *Computer-aided Verification*, LNCS. Springer, 1995.

[13] N. Lynch, M. Tuttle. Hierarchical correctness proofs for distributed algorithms. *Symp. Principles of Distributed Computing*, pp. 137–151. ACM, 1987.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1991.

[15] J. Misra, K. Chandy. Proofs of networks of processes. *IEEE Trans. Software Engineering*, 7(4):417–426, 1981.

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

[17] A. Pnueli. In transition from global to modular temporal reasoning about programs. *Logics and Models of Concurrent Systems*, pp. 123–144. Springer, 1984.