Relative Liveness: From Intuition to Automated Verification *

R. Negulescu J. A. Brzozowski

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

{radu,brzozo}@maveric∅.uwaterloo.ca
ftp://cs-archive.uwaterloo.ca/cs-archive/CS-95-32/CS-95-32.ps.Z

Abstract. We point out deficiencies of previous treatments of liveness. We define a new liveness condition in two forms: one based on finite trace theory, and the other on automata. We prove the equivalence of these two definitions. We also introduce a safety condition and provide modular and hierarchical verification theorems for both safety and liveness. Finally, we present a verification algorithm for liveness.

Index terms: Concurrent systems, deadlock, fairness, finite automata, liveness, safety, trace structures, verification.

1 Introduction

Motivation and scope

Formal verification, especially if it can be automated, gains importance as designed systems become more and more complex. Formal verification is particularly important for concurrent systems because non-deterministic interleavings of events can generate considerable complexity.

The subject of this paper is the definition, analysis, and automatic verification of a liveness condition for (possibly asynchronous) digital circuits and other concurrent systems. We view a concurrent system as a set of processes, where a process is a dynamic system with a discrete state-space. Digital circuits, parallel programs, and network protocols are examples of concurrent systems.

According to [LL90], most formal reasoning about concurrent systems has been concerned with two kinds of properties: safety and liveness. Intuitively, safety

*This research was supported by a grant and a scholarship from the Information Technology Research Centre of Ontario and by Grant No. OGP0000871 from the Natural Sciences and Engineering Research Council of Canada. An extended summary of this report was published as [NB95].

properties assert that "something bad does not happen" and liveness properties assert that "something good eventually does happen" [LL90]. Hazards, invalid outputs and invalid inputs are examples of safety faults. Deadlock and unfairness are examples of liveness faults. In our view, another class are progress properties, which assert that 'something good does happen within a bounded time.' Deadlock (again) and livelock are examples of progress faults. In our view, livelock is a progress fault but not a liveness fault: In a livelock situation, something good may take an unbounded time to happen; nevertheless, it will eventually happen. Here we do not consider livelock or other progress faults that do not violate the notion of liveness described informally in the citation above.

State of the topic

Defining a liveness condition has a major obstacle. In our view, a correctness condition should be expressible in a model no more detailed than 'common' representations of concurrent systems, such as Petri nets, concurrent programs in some language, or digital circuit schematics together with, say, relationships between the logic levels of inputs and outputs for representing components. Otherwise, that condition cannot be decided automatically from such common representations, because more information would be needed from the users. The major obstacle is that such representations specify only the finite executions (sequences of events) of concurrent systems, while finite executions are ambiguous for expressing liveness—a fact that follows from the characterization of liveness in [AS85]. More precisely, two systems with different liveness properties can have the same finite executions.

Liveness properties are determined by the *complete executions* of a concurrent system, i.e., by the finite or infinite sequences of actions that represent the *entire* operation of a system, i.e., until it stops or until the 'end of time.' The liveness properties, or, equivalently, the complete executions are not *explicitly* represented in a 'common' model (such as the models we listed above). For example, consider a gate specified by a boolean function. Such a gate is expected to eventually produce an output transition, after the boolean function has changed value in response to input transitions. However, if only the finite executions are specified, a gate which *may* behave as above, or *may* block internally and fail to produce an output transition, respects this specification, because it has exactly the same finite executions as the non-blocking gate. Nevertheless, the blocking gate has strictly more complete executions than the non-blocking gate. The non-blocking gate forbids complete executions that end with input transitions that should be eventually followed by output transitions; the blocking gate permits such complete executions. For another example, consider a mutual exclusion element which ensures that two processes do not access the same resource at the same time. A fair mutual exclusion element eventually grants the resource to each process that demands it. However, an element that may grant the resource to each process, but may also grant it to only one of the processes, would have the same finite executions as the fair mutual exclusion element.

As a result of the lack of modeling power of *finitary* representations (which specify only the finite executions of a concurrent system, like the common representations we listed above), previous treatments of liveness have used more powerful models of concurrent systems. There, the users have to specify the liveness prop-

erties of their systems, or, equivalently, the sets of complete executions of their systems. We call such approaches *user-directed*.

User-directed approaches have a high degree of generality, because they allow many types of liveness properties to be specified; however, they also have important deficiencies. From a practical point of view, such approaches are hard to use. The identification and specification of liveness properties and/or infinite executions is tedious and error-prone, and necessitates familiarity with representations of infinite sequences, such as ω-automata or temporal logics. From a theoretical point of view, user-directed approaches do not decide liveness on the basis of 'common' representations of concurrent systems (i.e., finitary representations like those we listed above). Users have to specify explicitly the liveness properties, or, equivalently, the complete executions, in addition to a 'common' representation of their systems. In effect, the users are required to formalize their own notions of deadlock, starvation, etc., by specifying these liveness properties. Most importantly, from both points of view, a user-directed approach provides no indication of *appropriateness* and *completeness* of a specification. In other words, such approaches do not address the problems whether the liveness requirements, specified by the users, are necessary, and whether they are sufficient to forbid, say, the danger of starvation in a particular implementation. (This stumbling point is also mentioned in [CMP92].)

Our approach

Here we resolve the ambiguity of finite-execution models by taking a different approach. The *constraints* of a concurrent system are the properties known to be satisfied, and the *requirements* are the properties that need to be satisfied. We noticed that, in 'common' models of concurrent systems, the liveness constraints are not explicitly specified. The reason for such omissions may be simply that liveness constraints do not need to be specified, because they are *implicitly assumed*. Practical boolean gates are not supposed to deadlock internally, practical mutual exclusion elements are supposed to be fair, and practical specifications, either global or intermediate, are not supposed to allow deadlock or starvation. We try to model these implicitly assumed liveness constraints by assigning *augmented semantics* to finitary representations; we relate a unique set of liveness properties to a finitary representation.

In other words, we note that, in many practical concurrent systems, the liveness *constraints* are related to the finite executions and to the sets of ports in a unique manner. We formalize this relationship by assigning complete execution semantics to a finitary representation, in addition to the usual, finite-execution semantics of such a representation. In Section 5 we argue this semantics holds at least for a large class of asynchronous circuits.

On the other hand, we note that the liveness *requirements* for a concurrent system may vary considerably. Nevertheless, we also relate the liveness requirements for a given system to finitary *specifications* by our augmented semantics. This way, we obtain a *relative* liveness condition, i.e., a condition that compares an implementation to a specification. This condition is determined by finitary representations of the implementation and the specification. As a result, our condition does not have the deficiencies of a user-directed approach we have mentioned above. From a practical point of view, finite automaton formalisms are more tractable than, say,

ω-automata, and a circuit or a concurrent program can be automatically translated into a network of finite automata, without extra input from the users. From a theoretical point of view, our condition is directly determined by 'common' representation of concurrent systems. Finally, regarding the problem of appropriateness and completeness of specified liveness properties, we cannot guarantee that our default liveness properties are indeed what the users want to specify; nevertheless, by our augmented semantics we at least suggest what the necessary and sufficient liveness properties might be, by analogy with the systems we have considered.

Liveness properties involve complete executions, which can be finite or infinite. The liveness constraints of our augmented semantics admit a unified form for finite and infinite sequences, which we name *strong liveness*.

Apart from the study of examples, we support our liveness condition by proving it satisfies certain desirable algebraic properties. These properties are important as tests of appropriateness of a condition and also constitute a technique for modular and hierarchical verification, as will be discussed later.

We derive a graph-theoretic form for our liveness condition, we show that it is equivalent to the language-theoretic form, and we use it for additional intuitive examination of our liveness condition and for deriving a decision algorithm.

We introduce a new condition for safety. Our safety condition agrees with previous conditions under certain connectivity restrictions, but, for the first time, has no restrictions on how the ports of the involved processes should be connected. We also prove sufficient theorems for modular and hierarchical verification of safety, without any connectivity restrictions.

Previous work

Some prominent treatments of liveness are [AS85], [LT87], [Jo87], and [Di89]. In [Jo87] and [Di89], very general frameworks for reasoning about liveness have been proposed, along with thorough algebraic treatments. However, those approaches are user-directed as described above. In [AS85], an exhaustive characterization of liveness properties has been proposed. However, nothing is said about which of the properties in the class defined by [AS85] can be used for a liveness condition. The liveness condition in [LT87] also provides important insights. However, that condition does not cover some common fairness flaws (see Section 8).

Two elegant models which capture progress properties based on finite traces, and have careful algebraic treatments, have been proposed in [Jos92] and [Ve94]. However, of the liveness faults, [Ve94] treats only global deadlocks, where every process is blocked (no process has to perform an output action) but some process demands an input action. For simplicity, the treatment in [Jos92] does not model processes, such as clocks and ring oscillators, that may never stop in a correct operation, and also does not deal with fairness.

CCS [Mi89] and CSP [Ho85] are two powerful high-level models of concurrent systems. However, [LT87] lists several problems with these formalisms when they are used to define liveness. Also, in CCS, an action between a sender and a receiver can occur only if both processes allow it, which may be inconvenient for modeling low-level communication (where an action can occur even if the receiver is not ready, producing a fault). According to [Di89], CSP has a similar inconvenience.

Contents and form

This paper is structured as follows. In Section 2 we define our basic model, which is closely related to trace theories like those of [Sn83], [Ud86], [Eb86], and [Ve94]. In Section 3, we discuss a general pattern and some desirable properties for modular and hierarchical verification of correctness conditions. In Section 4, we discuss correctness conditions other than liveness that are needed as restrictions for our liveness condition. These conditions extend and simplify other conditions in trace theory. In Section 5 we introduce our liveness condition. In Section 6 we introduce an automaton model for concurrent systems and relate it to the trace structure model by a semantic mapping. We also define a parallel composition on automata and relate it to its trace-structure counterpart. In Section 7 we state a graph-theoretic form for our liveness condition and relate it to the language-theoretic liveness condition of Section 5. In Section 8 we consider and criticize some variations to our liveness condition, and we point out some shortcomings of the liveness condition in [LT87] and of another condition in the literature. In our basic automaton model, we do not model certain cases of non-determinism, for simplicity and because we consider them to be rather marginal; in Section 9, we extend the graph-theoretic form of our condition to capture these cases, too, at the cost of additional complexity. In Section 10, we present and analyze an algorithm for the verification of our liveness condition. Section 11 concludes the paper.

Appendix A contains proofs of the results in Sections 2, 4, and 5 (the results that involve the trace structure model only). Appendix B contains proofs of the results in Sections 6 to 7 (the results that also involve the automaton model). The results regarding the algorithm are given in Section 10.

We use double quotes " " for citations and single quotes ' ' for some informal or undefined terms.

2 Trace Structures

Preliminaries

We let $\mathcal{U}$ be a set, called the *symbol universe.* An *alphabet* is a subset of $\mathcal{U}$. A *word* over an alphabet Σ is a finite sequence of symbols from Σ. Concatenation of two words is denoted by their juxtaposition. The empty word is ε. For two words s and t, we write $s \leq t$ if s is a prefix of t. For example, $aba \leq abaa$. For word t and symbol a, $|t|_a$ denotes the number of occurrences of a in t. For example, $|abccb|_b = 2$.

A *language* is a set of words. We use the following notation for languages: **pref** is prefix-closure (the set of all prefixes of the words in a language), * is Kleene closure, $\cup$ is union, $\cdot$ or juxtaposition is concatenation, symbol x can represent language $\{x\}$, and alphabet Σ can represent the language of single-symbol words with symbols from Σ. A language is *prefix-closed* if it is equal to its prefix-closure.

The trace structure model

A *trace structure* is a triple $P = \langle \mathbf{i}\,P, \mathbf{o}P, \mathbf{lg}\,P \rangle$ of two disjoint alphabets $\mathbf{i}\,P$ and $\mathbf{o}P$ and a prefix-closed, non-empty language $\mathbf{lg}\,P$ over $\mathbf{i}\,P \cup \mathbf{o}P$. The words of $\mathbf{lg}\,P$ are called *traces* of P. The *alphabet* of P, denoted by $\mathbf{a}P$, is $\mathbf{i}\,P \cup \mathbf{o}P$. The symbols

in $\mathbf{a}P$ are called *actions* of P.

If symbol a is in $\mathbf{o}P$, P is a *source* for a; if $a \in \mathbf{i}\,P$, P is a *sink* for a; if a is not in the alphabet of P, P is *unrelated* to a.

A trace structure P can represent a process in the following manner. Symbols in $\mathbf{a}P$ stand for ports. Symbols in $\mathbf{o}P$, called *outputs*, are ports controlled by the process; they include the 'internal' ports and the genuine output ports. Symbols in $\mathbf{i}\,P$, called *inputs*, represent ports controlled by the environment. Traces in $\mathbf{lg}\,P$ stand for finite sequences of events that may have occurred in the modeled process up to a certain time. (This interpretation justifies the restriction that $\mathbf{lg}\,P$ is prefix-closed.)

To illustrate how we represent processes by trace structures, consider a XOR gate with inputs a and b and output c. The actions are the signals on the gate terminals. The traces are all possible sequences of signal transitions, where the signal transitions are denoted by the symbols associated to the terminals on which they occur: A transition on terminal a is denoted by a. The language is derived using the observation that there must be an odd number of input transitions between any two consecutive transitions on c and before the first transition on c, if any.[1] The language is thus $\mathbf{pref}((a \cup b)(a \cup b \cup c))^*$.

Note that, if a process has 'internal ports' (e.g., internal signals, in the case of a circuit), we treat those ports as outputs, since they are controlled by the process, just like the genuine, external, outputs. Several authors (e.g., [Eb91]) allow the input and output alphabets of a trace structure to overlap; the disjointness condition that we use is intended to be consistent with the particular intuitive meaning that we assign to input and output alphabets. For example, if the XOR gate above is connected to a second XOR gate with inputs c and d and output e, the resulting circuit has inputs a, b and d and outputs c and e. In [Eb91], c would be considered both an input and an output, but here we consider it just an output because it is controlled by the circuit.

We do not require processes to accept any input at any time. For example, consider the asynchronous MERGE element (a 'hazard-intolerant' version of a XOR). The environment must wait for a transition on c to occur between any two input transitions. The trace structure of MERGE is $\langle \{a, b\}, \{c\}, \mathbf{pref}((a \cup b)\,c)^* \rangle$. Word $acab$, which is not in the language, causes a hazard because, after trace aca, the environment should wait for another transition on c and is not allowed to produce a b immediately.

Parallel composition

A *network* is a set of trace structures. Note that there are no restrictions on the alphabets of the trace structures in a network.

The *projection* of a word t on an alphabet Σ is a word $t{\downarrow}\Sigma$ obtained by deleting from t all symbols which are not in Σ. For word t, trace structure P, and network N, we denote by t_P the projection of t on the alphabet of P, i.e., $t_P = t{\downarrow}\mathbf{a}P$, and we denote by t_N the projection of t on the union of the alphabets of the trace structures in N, i.e., $t_N = t{\downarrow}(\cup_{Q \in N} \mathbf{a}Q)$. Note that $t_P = t_{\{P\}}$.

[1] This is only one of many possible behaviors one can associate with a XOR gate. It is the unrestricted behavior [BS95] in a 'single-winner' model (GSW), assuming inertial delays.

The *parallel composition* of trace structures is a binary operation $\|$ such that:

$$\begin{array}{lcl} \mathbf{i}\,(P\|Q) & = & (\mathbf{i}\,P \cup \mathbf{i}\,Q) \;-\; (\mathbf{o}P \cup \mathbf{o}Q), \\ \mathbf{o}(P\|Q) & = & \mathbf{o}P \cup \mathbf{o}Q, \text{ and} \\ \mathbf{lg}\,(P\|Q) & = & \{t \in (\mathbf{a}P \cup \mathbf{a}Q)^* \mid t_P \in \mathbf{lg}\,P \wedge t_Q \in \mathbf{lg}\,Q\}. \end{array}$$

The result of parallel composition is called a *composite*. Note that there are no restrictions on the composed processes. Similar operators have been used before in trace theory (e.g., in [Eb86]).

Parallel composition is naturally extended to arbitrary networks. The *composite* of a network N is a trace structure $\|N$ such that:

$$\begin{array}{lcl} \mathbf{i}\,\|N & = & \bigcup_{P\in N} \mathbf{i}\,P \;-\; \bigcup_{P\in N} \mathbf{o}P, \\ \mathbf{o}\|N & = & \bigcup_{P\in N} \mathbf{o}P, \text{ and} \\ \mathbf{lg}\,\|N & = & \{t \in (\bigcup_{P\in N} \mathbf{a}P)^* \mid \forall\, P \in N,\; t_P \in \mathbf{lg}\,P\}. \end{array}$$

Informally, the composite represents a process whose behavior is compatible with all composed processes. For example, consider again a MERGE $\langle\{a,b\}, \{c\}, \mathbf{pref}((a \cup b)\,c)^*\rangle$ and a WIRE $\langle\{c\}, \{d\}, \mathbf{pref}(c\,d)^*\rangle$, connected at the output of the MERGE. Their composite is $\langle\{a,b\}, \{c,d\}, \mathbf{pref}((a \cup b)\,(c\,(da \cup db \cup ad \cup bd))^*)\rangle$. Symbol c is an output for the composite because it is driven by the device (for us, it does not matter that c is also an input to the WIRE component). Trace $t = acdbcdac$ appears in the language of the composite because $t{\downarrow}\{a,b,c\} = acbcac$ is in the language of the MERGE and $t{\downarrow}\{c,d\} = cdcdc$ is in the language of the WIRE. Trace $acadcd$ appears in the language of the composite because it does not violate the specification of either element. However, if the second c occurred before the first d, a hazard would occur, violating the specification of the WIRE. Thus $acacdd$ is not in the language. The network of concurrent processes instantiated by this circuit is not 'safe' (see Section 4); still, its composite is defined.

Parallel composition is well-defined: The input and output alphabets of the composite are disjoint, and the language of the composite is prefix-closed. Also, this operation has the following algebraic properties:

Proposition 1 *Parallel composition of trace structures is idempotent, commutative, and associative.*

All the proofs are given in the appendices.

Reflection

Another operation of interest on trace structures is reflection. The *reflection* of a trace structure P is a trace structure $\overline{P}$ such that $\mathbf{i}\,\overline{P} = \mathbf{o}P$, $\mathbf{o}\overline{P} = \mathbf{i}\,P$, and $\mathbf{lg}\,\overline{P} = \mathbf{lg}\,P$. Informally, $\overline{P}$ is intended to model the 'worst' environment where P can function correctly.

The *reflection* of network N is a network $\{\overline{\|N}\}$.

3 Common Characteristics of Correctness Conditions

A pattern for correctness conditions
The correctness conditions in this paper have the format $S \sqsubseteq_\xi I$, where S and I are networks and ξ is a correctness criterion. Such a condition is read I *realizes* S *with* ξ; S is called the *specification* and I the *implementation*. We sometimes write $S \sqsubseteq_{\xi\eta} I$ instead of $S \sqsubseteq_\xi I \wedge S \sqsubseteq_\eta I$. Some of the conditions in this paper are also defined as predicates over networks. For such a predicate ξ, we define $S \sqsubseteq_\xi I \Leftrightarrow \xi(I \cup \overline{S})$. Informally, this definition means that I realizes S with ξ if I satisfies the correctness concern ξ when operating in the worst environment of S.

Structured verification
The modular and hierarchical structure of concurrent systems can be used to reduce the computational costs of verification. To allow for modular and hierarchical verification, a realization relationship $\sqsubseteq_\xi$ needs to satisfy only the following two properties:

$\cup$-Compatibility *For networks M, N, and O such that $M \sqsubseteq_\xi N$, we have $M \cup O \sqsubseteq_\xi N \cup O$.*

Note that O is arbitrary. Informally, this property says that, if N is at least as good as M, then N performs at least as well as M even in the context of O. For a system that breaks up into modules, each module having its own implementation, $\cup$-compatibility permits one to verify the modules independently, one at a time.

Transitivity *For networks M, N, and O such that $M \sqsubseteq_\xi N$ and $N \sqsubseteq_\xi O$, we have $M \sqsubseteq_\xi O$.*

For a system that admits different levels of abstraction, transitivity permits one to verify pairs of consecutive levels independently.

For example, suppose we need to verify that $\{S\} \sqsubseteq_\xi \{U, R, V, W\}$. Furthermore, suppose the system $\{U, R, V, W\}$ breaks up into modules such that it is convenient to check that $\{S\} \sqsubseteq_\xi \{P, Q\}$, $\{P\} \sqsubseteq_\xi \{U, R\}$, and $\{Q\} \sqsubseteq_\xi \{V, W\}$, where trace structures P and Q represent some intermediate specifications. By $\cup$-compatibility, we have $\{P\} \sqsubseteq_\xi \{U, R\} \Rightarrow \{P, Q\} \sqsubseteq_\xi \{U, R, Q\}$ and $\{Q\} \sqsubseteq_\xi \{V, W\} \Rightarrow \{U, R, Q\} \sqsubseteq_\xi \{U, R, V, W\}$. By transitivity, it follows that $\{P, Q\} \sqsubseteq_\xi \{U, R, V, W\}$. By transitivity again, since $\{S\} \sqsubseteq_\xi \{P, Q\}$, we obtain the desired result $\{S\} \sqsubseteq_\xi \{U, R, V, W\}$.

Note that, a priori, we impose no restrictions on the alphabets of S, P, Q, U, R, V, and W. As will be discussed later, our condition for liveness still has some connectivity restrictions, but our condition for safety has no such restrictions. For both our conditions, it is possible that specifications and implementations have different alphabets. For example, the intermediate specification P from the example above could have fewer symbols than the implementation part $\{U, R\}$, in order to abstract that part for the next level of verification (S compared to $\{P, Q\}$).

As a result, we avoid the need for projection or hiding operators on processes for performing hierarchical and modular verification: Since we do not restrict the alphabets of the specification and the implementation, we do not need to get these alphabets to match by a projection. (For comparison, in [Di89] a specification and an implementation have to have the same inputs and the same outputs.) We do not care how the intermediate specification P is constructed or guessed; it might be the result of a hiding or projection operator. We do not define such an operator, because it does not preserve the liveness properties of our processes; still, such an operator can be used in our verification method as described above.

For another example, suppose the system $\{U, R, V, W\}$ above breaks up into modules such that it is convenient to check that $\{S\} \sqsubseteq_\xi \{P, Q, R, T\}$, $\{P, Q\} \sqsubseteq_\xi \{U\}$, and $\{T\} \sqsubseteq_\xi \{V, W\}$, where trace structures P, Q, and T represent some intermediate specifications. By $\cup$-compatibility, we have $\{P, Q\} \sqsubseteq_\xi \{U\} \Rightarrow \{P, Q, R, T\} \sqsubseteq_\xi \{U, R, T\}$ and $\{T\} \sqsubseteq_\xi \{V, W\} \Rightarrow \{U, R, T\} \sqsubseteq_\xi \{U, R, V, W\}$. By transitivity, it follows that $\{P, Q, R, T\} \sqsubseteq_\xi \{U, R, V, W\}$. By transitivity again, since $\{S\} \sqsubseteq_\xi \{P, Q, R, T\}$, we obtain the desired result $\{S\} \sqsubseteq_\xi \{U, R, V, W\}$.

These properties refine the "separation" and "substitution" theorems in [Eb91].

4 Connectivity and Safety Conditions

Motivation

We have imposed no restrictions on the operands of our parallel composition, but we need to introduce explicit restrictions on the networks on which we define a concept of liveness. Fortunately, however, these restrictions are themselves necessary correctness conditions: a connectivity condition and a safety condition. These conditions are presented next.

The condition for safety is also interesting by itself because it is intended to cover all safety concerns. On the other hand, we do not do a thorough study of connectivity concerns.

Connectivity

Previous trace models contain several connectivity conditions ("alphabet conditions"). We do not adopt all of them because we are mainly interested in 'minimal' connectivity conditions that ensure the applicability of our liveness condition.

Definition 1 *Network N is* output-consistent, *written $\omega(N)$, if*

$$\forall\, P, Q \in N,\ \mathbf{o}P \cap \mathbf{o}Q = \emptyset.$$

This requirement is necessary for digital circuits: If the outputs of two circuit parts were driving the same circuit node with different voltages, a short would occur. Exceptions (such as wired-logic circuits or tri-state outputs) can be modeled by introducing separate processes for complex connectors (such as buses); this way, the element outputs that could be tied together become tied only to the inputs of the complex connector and to no outputs, thus respecting output consistency.

The output consistency condition is not compatible with hierarchical and modular verification. Nevertheless, it can be checked easily in a direct manner.

Another connectivity condition is that no inputs may be left dangling, i.e., all inputs of all processes in a concurrent system are either outputs of other processes in the system or 'external' inputs of the system (outputs of the 'environment process'). We call this condition *input control.* We do not treat input control formally because we do not need it as a restriction for our liveness condition. Nevertheless, we mention input control because we refer to it in later examples.

Safety

Safety has been extensively studied in trace theory. Conditions covering safety concerns have been proposed, for example, in [Sn83, Ud86, Eb86, Di89, Eb91, Jos92, GBMN94, Ve94]. Our condition for safety agrees with some of these previous conditions under appropriate connectivity restrictions, and we discuss this issue in more detail later in this section. However, all these previous conditions have restrictions (either explicit or hidden in the model) on the ports of the processes they can compare or connect, and on the theorems for structured verification. We have eliminated all such restrictions from the condition itself and its structured verification theorems. The fact that connectivity restrictions are not needed for the treatment of safety was surprising, particularly in Theorem 1, which refers to modular verification (see below).

Definition 2 *Network N is* safe, *written $\sigma(N)$, if, for all words t in $\mathcal{U}^*$ such that*

$$\forall\ P \in N,\ t_P \in \mathbf{lg}\, P \cdot (\mathbf{i}\, P \cup \{\varepsilon\})$$

we have

$$t_N \in \mathbf{lg}\, \|N.$$

For networks S and I we say that I realizes *S* with safety, *written $S \sqsubseteq_\sigma I$, if $\sigma(\overline{S} \cup I)$.*

For an intuitive explanation, we refer to the 'such that' part in the definition of safety as the *precondition* and to the 'we have' part as the *postcondition.* Informally, our safety condition demands that, whenever an event is allowed to happen by all its sources in N, that event must be allowed to happen by all its sinks in N. To see that, consider a situation where the safety condition may be violated. Let $t = ua$ be such that u_N is in $\mathbf{lg}\,\|N$ and symbol a is in $\mathcal{U}$. For every source P of a in N, the precondition says that $(ua)_P$ is in $\mathbf{lg}\, P$, because a cannot be in $\mathbf{lg}\, P \cdot \mathbf{i}\, P$, since $\mathbf{i}\, P$ and $\mathbf{o}P$ are disjoint. For any sink P of a, the precondition is empty because $u_P \in \mathbf{lg}\, P$. For any P unrelated to a, the precondition is also empty because $(ua)_P = u_P \in \mathbf{lg}\, P$. In words, the precondition only says that a is allowed to happen after u by all its sources P. Our safety condition demands that, if the precondition is satisfied, the postcondition must also be satisfied. The postcondition requires that, for every $P \in N$, $(ua)_P \in \mathbf{lg}\, P$. If P is a source for a, the postcondition is a trivial consequence of the precondition. If P is unrelated to a, the postcondition is empty, because $(ua)_P = u_P \in \mathbf{lg}\, P$. Thus, the postcondition only requires that a is allowed to happen after u by all its sinks P.

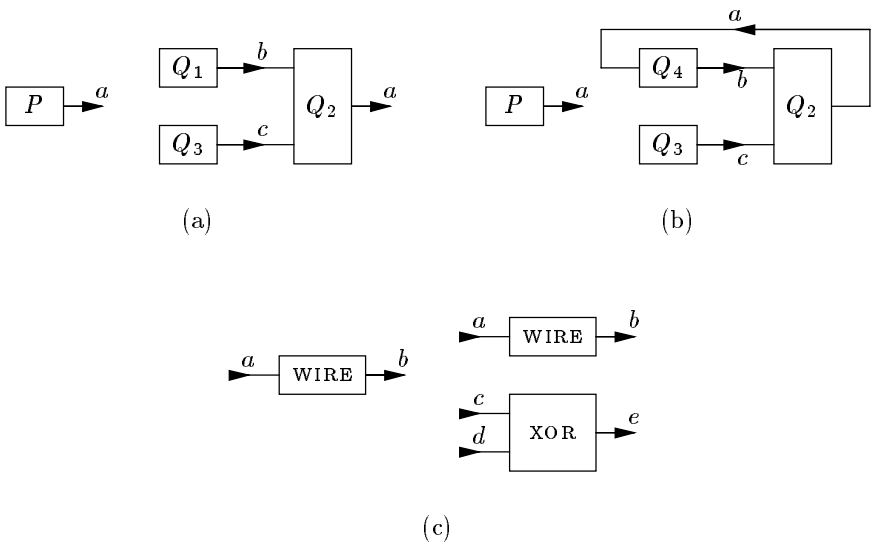


Figure 1: Examples for the safety condition.

Safety in systems with 'normal' connectivity conditions

For a first example, consider a specification containing just a CLOCK $P = \langle \emptyset, \{a\}, a^* \rangle$ and an implementation containing three elements: a CLOCK $Q_1 = \langle \emptyset, \{b\}, b^* \rangle$, a MERGE $Q_2 = \langle \{b, c\}, \{a\}, \mathbf{pref}((b \cup c)a)^* \rangle$, and a link $Q_3 = \langle \emptyset, \{c\}, \{\varepsilon\} \rangle$ from ground to c. (The link to ground ensures there can be no transition on c.) See Figure 1 (a). (Boxes represent processes, incoming arrows represent inputs and outgoing arrows represent outputs.) We show that the network $\{Q_1, Q_2, Q_3\}$ does not realize $\{P\}$ with safety. For consider trace $t = bb$. We check that t satisfies the safety precondition: $t_{Q_1} = bb \in \mathbf{lg}\, Q_1$, $t_{Q_2} = bb \in \mathbf{lg}\, Q_2 \cdot \mathbf{i}\, Q_2$, $t_{Q_3} = \varepsilon \in \mathbf{lg}\, Q_3$, and $t_P = \varepsilon \in \mathbf{lg}\, \overline{P}$. We check that t does not satisfy the safety postcondition: $t_{Q_2} = bb \notin \mathbf{lg}\, Q_2 \Rightarrow t_{\{\overline{P}, Q_1, Q_2, Q_3\}} \notin \mathbf{lg} \,\|\{\overline{P}, Q_1, Q_2, Q_3\}$. Consequently, t causes a safety violation (a 'hazard' for the MERGE) and $\{P\} \not\sqsubseteq_\sigma \{Q_1, Q_2, Q_3\}$.

In the following example, we modify the previous example to achieve safety. An I-WIRE (a 'hazard-intolerant' inverter) can be represented by $\langle \{a\}, \{b\}, \mathbf{pref}(ba)^* \rangle$. In the implementation in the previous example, we replace the CLOCK Q_1 by an element Q_4 representing the I-WIRE above. See Figure 1 (b). We show that $\{Q_4, Q_2, Q_3\}$ realizes $\{P\}$ with safety. First, we characterize the languages of the elements in terms of numbers of occurrences of certain actions in traces:

$$\mathbf{lg}\, Q_2 = \{t \in \{a, b, c\}^* \mid \forall\, u \leq t,\ |u|_a \leq |u|_b + |u|_c \leq |u|_a + 1\}$$
$$\mathbf{lg}\, Q_3 = \{t \in \{c\}^* \mid \forall\, u \leq t,\ |u|_c = 0\} = \{\varepsilon\}$$
$$\mathbf{lg}\, Q_4 = \{t \in \{a, b\}^* \mid \forall\, u \leq t,\ |u|_b - 1 \leq |u|_a \leq |u|_b\}$$

Second, we use the safety precondition to deduce relationships on the numbers of

occurrences of certain actions in traces:

$$t_{Q_2} \in \mathbf{lg}\, Q_2 \cdot (\mathbf{i}\, Q_2 \cup \{\varepsilon\}) \quad \Rightarrow \quad \forall\, u \leq t,\ |u|_a \leq |u|_b + |u|_c \tag{1}$$

$$t_{Q_3} \in \mathbf{lg}\, Q_3 \cdot (\mathbf{i}\, Q_3 \cup \{\varepsilon\}) \quad \Rightarrow \quad \forall\, u \leq t,\ |u|_c = 0 \tag{2}$$

$$t_{Q_4} \in \mathbf{lg}\, Q_4 \cdot (\mathbf{i}\, Q_4 \cup \{\varepsilon\}) \quad \Rightarrow \quad \forall\, u \leq t,\ |u|_b \leq |u|_a + 1 \tag{3}$$

(Since $\mathbf{i}\,\overline{P} = \mathbf{a}P = \{a\}$ and $\mathbf{lg}\,\overline{P} = \{a\}^*$, in this example $t_P \in \mathbf{lg}\,\overline{P} \cdot (\mathbf{i}\,\overline{P} \cup \{\varepsilon\})$ is an empty condition.) Finally, we deduce the safety postcondition:

$$(2) \wedge (3) \ \Rightarrow\ \forall\, u \leq t,\ |u|_b + |u|_c = |u|_b \leq |u|_a + 1 \ \Rightarrow\ t_{Q_2} \in \mathbf{lg}\, Q_2$$
$$(1) \wedge (2) \ \Rightarrow\ \forall\, u \leq t,\ |u|_a \leq |u|_b + |u|_c = |u|_b \ \Rightarrow\ t_{Q_4} \in \mathbf{lg}\, Q_4$$
$$(2) \ \Rightarrow\ t_{Q_3} \in \mathbf{lg}\, Q_3$$

Consequently, $\{P\} \sqsubseteq_\sigma \{Q_4, Q_2, Q_3\}$.

For the case with no dangling inputs and no connected outputs, our safety condition agrees with "absence of computation interference." (We refer the reader to the version in [Eb91] for comparison purposes, but similar conditions have been defined at least in [Sn83, Ud86, Eb86, Ve94].) To see that, consider the following informal reasoning. Our condition says that, whenever an event is allowed to happen by *all its sources* in network N, that event must be allowed to happen by *all its sinks* in N. Intuitively, "absence of computation interference" demands that, whenever an event is allowed to happen by *some of its sources* in N, that event must be allowed to happen by *all other sinks or sources of that event* in N (otherwise, "computation interference" would occur). If there are no dangling inputs and no connected outputs, every action has *exactly one source*. In this case, "some of its sources" = "all its sources" and "all its sinks" = "all *other* sinks or sources of that event" (there are *no* other sources in this case, because there is only one source of that event).

Safety in systems with dangling inputs

Systems with dangling inputs may be regarded as 'incorrect,' but they may be 'safe.' Some examples are certain systems where the implementation has redundant elements that do not affect the specified outputs. For instance, consider the WIRE represented by $\langle \{a\},\ \{b\},\ \mathbf{pref}(ab)^* \rangle$ and a XOR represented by $\langle \{c, d\},\ \{e\},\ \mathbf{pref}((c \cup d)(c \cup d \cup e))^* \rangle$. Consider $S = \{\text{WIRE}\}$ and $I = \{\text{WIRE}, \text{XOR}\}$. See Figure 1 (c). Intuitively, note that the XOR is completely disconnected from the WIRE in the implementation, and thus the implementation behaves irreproachably with respect to the specification. (Since the actions c, d, e are not in the alphabet of the specification, their events are unspecified.) Formally, let t be a trace such that $t_{\text{WIRE}} \in \mathbf{lg}\,\text{WIRE} \cdot (\mathbf{i}\,\text{WIRE} \cup \{\varepsilon\})$, $t_{\overline{\text{WIRE}}} \in \mathbf{lg}\,\overline{\text{WIRE}} \cdot (\mathbf{i}\,\overline{\text{WIRE}} \cup \{\varepsilon\})$ and $t_{\text{XOR}} \in \mathbf{lg}\,\text{XOR} \cdot (\mathbf{i}\,\text{XOR} \cup \{\varepsilon\})$. One verifies that, since $\mathbf{i}\,\text{WIRE} \cap \mathbf{o}\text{WIRE} = \emptyset$, we have $t_{\text{WIRE}} \in \mathbf{lg}\,\text{WIRE}$ and $t_{\overline{\text{WIRE}}} \in \mathbf{lg}\,\overline{\text{WIRE}}$, and that $\mathbf{lg}\,\text{XOR} \cdot (\mathbf{i}\,\text{XOR} \cup \{\varepsilon\}) \subseteq \mathbf{lg}\,\text{XOR}$; therefore, $t_{\{\overline{\text{WIRE}}, \text{WIRE}, \text{XOR}\}} \in \mathbf{lg}\,\{\overline{\text{WIRE}}, \text{WIRE}, \text{XOR}\}$. This proves that $S \sqsubseteq_\sigma I$, in agreement with our intuition.

In the previous example, we have chosen XOR rather than MERGE because it can accept arbitrary input transitions (although it may not respond to all of them). If MERGE were used instead, safety violations could occur on the inputs of MERGE. (Recall that MERGE is the 'hazard-intolerant' version of a XOR.)

In the case with dangling inputs, our safety condition imposes a "receptiveness" requirement on the set of traces of a network with respect to the set of dangling inputs: In a safe network, an event on a dangling input port should be acceptable at any time by the sink processes of that port. (The events on ports which are not dangling inputs are treated just like the events from the case with normal connectivity conditions.)

Receptiveness has been used previously in [Di89] and [Jos92]. However, in both [Di89] and [Jos92], receptiveness is used as a model restriction on processes rather than a correctness condition on networks. Moreover, in [Di89] receptiveness is a constraint on the "set of possible traces," rather than the "set of successful traces," and thus it has a different meaning than here. Nevertheless, the receptiveness requirement imposed by our safety condition upon the dangling inputs is similar in meaning to the receptiveness constraint in [Jos92], if the whole network is viewed as a single process.

To point out the difference between our condition and absence of computation interference in the case with dangling inputs, consider the following. First, absence of computation interference is not defined for dangling inputs. More importantly, if that condition were extended by removing the restriction that no inputs should be dangling, absence of computation interference would be trivially satisfied on dangling inputs: No "computation interference" can occur on a dangling input port, since there is no source process in the network to generate an event on that port.

Safety in systems with connected outputs

It was interesting to note what the safety condition says about the situations where output ports are shared. Such situations are normally disallowed and we will not illustrate them by an example. Nevertheless, if inputs are connected, our safety condition can be understood as follows: If an event is not allowed by a source process, that event does not happen and does not cause a safety fault, even if that event is allowed by another source process. Note the disagreement with absence of computation interference in this case.

Structured verification of safety

We now state the $\cup$-compatibility and transitivity theorems for safety.

Theorem 1 *For networks M, N, and O such that $M \sqsubseteq_\sigma N$, we have $M \cup O \sqsubseteq_\sigma N \cup O$.*

Theorem 2 *For networks M, N, and O such that $M \sqsubseteq_\sigma N$ and $N \sqsubseteq_\sigma O$, we have $M \sqsubseteq_\sigma O$.*

Proofs are given in Appendix A.

Note that Theorems 1 and 2 assume *no* connectivity restrictions. This absence of restrictions was surprising, especially for Theorem 1. For example, O may have common symbols with M and N, even common output symbols, and these common symbols do not need to be the same for M and N. For example, O could share output port a with N and input port b with M.

As discussed in Section 3, the absence of connectivity restrictions in these theorems permit one to perform hierarchical and modular verification without using a hiding or projection operator. Nevertheless, such an operator can still be used as a constructor for intermediate specifications to be verified.

5 Liveness

Preliminaries
For alphabet $\Sigma \subseteq \mathcal{U}$, let Σ^ω be the set of all infinite sequences of symbols from Σ, and Σ^∞ the set of all finite or infinite sequences of symbols from Σ. We have $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. Since we do not use other sequences, we refer to (finite or infinite) sequences of symbols from $\mathcal{U}$ as just sequences. Concatenation of a (finite) word and a (possibly infinite) sequence of symbols is denoted by their juxtaposition. For word u, we denote by u^ω the infinite sequence $uuu\ldots$. For language L, we denote by L^ω the set of sequences obtained by concatenating infinitely many words from L. For example, $\{ab, ac\}^\omega = \{e : \mathbb{N} \to \{a, b, c\} \mid \forall\, i \in \mathbb{N},\, (e_{2i} = a \ \wedge\ e_{2i+1} \in \{b, c\})\}$, where $\mathbb{N}$ is the set of natural numbers $\{0, 1, 2, \ldots\}$. For sequences t and e, we write $t \leq e$ if t is a finite prefix of e, that is, any prefix of e except e itself, if e is infinite. For example, if $e = abbb\ldots$, then $e = ab^\omega$ and $abb \leq e$; also, $\varepsilon \leq e$ for every sequence e. Since we do not use infinite prefixes at all, we refer to finite prefixes as just prefixes. We extend the projection operation from words in the obvious way. For sequence e and alphabet Σ, we denote by $e{\downarrow}\Sigma$ the *projection* of e on Σ. For sequence e, trace structure P, and network N, we use the notation $e_P = e{\downarrow}\mathbf{a}P$ and $e_N = e{\downarrow}(\cup_{Q \in N} \mathbf{a}Q)$. We have $e_P = e_{\{P\}}$.

Limits
A *limit* of a language L is a sequence e such that every prefix of e is in L. The *limit set* of L is $\mathbf{lim}\, L = \{e \in \Sigma^\infty \mid \forall\, t \leq e,\, t \in L\}$. A *limit* of trace structure P is a limit of $\mathbf{lg}\, P$; the *limit set* of P is $\mathbf{lim}\, P = \mathbf{lim}\, \mathbf{lg}\, P$. For example, consider a WIRE $\langle \{a\}, \{b\}, \mathbf{pref}(ab)^* \rangle$; the limit set is $\mathbf{pref}\ (ab)^* \cup (ab)^\omega$. Note that limits can be finite, and that the *finite* limits of a trace structure are precisely its traces. (Any prefix of a trace in a trace structure P is itself a trace of P; thus any trace of P is a finite limit of P. Also, any finite limit of P is a finite prefix of itself and thus must be in $\mathbf{lg}\, P$, by the definition of limits.) Thus, we have $\mathbf{lg}\, P \subseteq \mathbf{lim}\, P$.

The following proposition computes the limits of composites and reflections.

Proposition 2 *For trace structures P and Q,*
(a) $\mathbf{lim}\,(P \| Q) = \{e \in (\mathbf{a}P \cup \mathbf{a}Q)^\infty \mid e_P \in \mathbf{lim}\, P \ \wedge\ e_Q \in \mathbf{lim}\, Q\}$
(b) $\mathbf{lim}\, \overline{P} = \mathbf{lim}\, P$.

Strong liveness
Informally speaking, the *complete* executions of a concurrent system are (finite or infinite) sequences of events that can occur until the 'end of time.' In contrast to that, the *partial* executions are (finite) sequences that can occur within a bounded time.

Trying to formalize a notion of complete executions of a concurrent system, we have obtained a generic property that unifies a "strong fairness" property of *infinite*

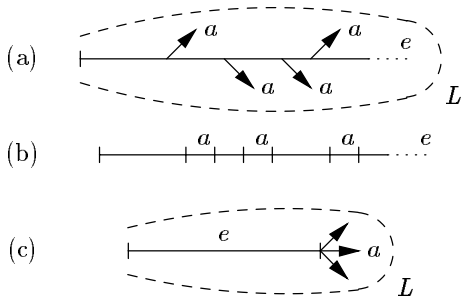


Figure 2: Recurrently enabled and fired symbols.

sequences (e.g., see [Fr86]) with a "quiescence" property of *finite* sequences (e.g., see [Jo87]). We call this property *strong liveness*. The property is formally the same for infinite and finite sequences, but, for clarity, the intuitive explanations are given separately for the two cases.

Symbol a is *recurrently enabled* by sequence e with respect to language L if $\forall\, t \leq e,\ \exists\, u : (tu \leq e \wedge tua \in L)$. The set of recurrently enabled symbols of e with respect to L is denoted by $\mathbf{ren}_L e$. Finite sequence t *immediately enables* a symbol a in language L if $ta \in L$. Note that, if e is infinite, the recurrently enabled symbols of e with respect to L are those symbols that are immediately enabled in L by infinitely many prefixes of e. See Figure 2 (a). If e is finite, the recurrently enabled symbols of e with respect to L are the symbols immediately enabled by e in L. See Figure 2 (c). For example, $\mathbf{ren}_{\mathbf{pref}((ab)^*c)}(ab)^\omega = \{a, b, c\}$ and $\mathbf{ren}_{\mathbf{pref}((ab)^*c)}ab = \{a, c\}$.

Symbol a is *fired* by sequence e if a appears in e at least once. Symbol a is *recurrently fired* by e if $\forall\ t \leq e,\ \exists\ u : tua \leq e$. The set of recurrently fired symbols of e is denoted by $\mathbf{rfi}\, e$. Note that the recurrently fired symbols are exactly the symbols fired infinitely often. See Figure 2 (b). Thus, a finite sequence has no recurrently fired symbols, i.e., for finite sequence e, $\mathbf{rfi}\, e = \emptyset$. For example, $\mathbf{rfi}\, ac(ab)^\omega = \{a, b\}$ and $\mathbf{rfi}\, aba = \emptyset$.

For alphabet Σ and language L, limit e of L is *strongly live* with respect to Σ and L if e recurrently fires all symbols from Σ that e recurrently enables in L, i.e., if $\mathbf{ren}_L e \cap \Sigma \subseteq \mathbf{rfi}\, e$.

Limit e of trace structure P is an *output trap* of P if e is strongly live with respect to $\mathbf{o}P$ and $\mathbf{lg}\, P$. The *set of output traps* of P is denoted by $\mathbf{otp}\, P$. (Note that $\mathbf{otp}\, P \subseteq \mathbf{lim}\, P$, and that the set of output traps of a trace structure is uniquely determined by its language and alphabets.) Output traps formalize our idea of 'reasonable' or 'live' complete executions of a process. For an intuitive picture, consider that the execution point of a system follows limit e. The recurrently enabled output actions can be viewed as exerting a pressure to be fired by the process; that pressure is relieved for recurrently fired actions only. If an output action a is recurrently enabled but is not recurrently fired by e, the pressure builds up and e is not complete because an a event is due to be fired by the process.

For example, consider a SELECTOR $\langle \{a\}, \{b,c\}, \mathbf{pref}(a(b \cup c))^* \rangle$ (upon request a, it responds with either b or c, taking a choice). The set of output traps of this SELECTOR is $(a(b \cup c))^* \cup \{\ e \in \{ab, ac\}^\omega \mid e$ fires b infinitely many times and e fires c infinitely many times $\}$. The finite limits from $(a(b \cup c))^*$ are output traps because they do not immediately enable any output action. The infinite limits that fire both b and c infinitely many times are output traps because b and c are the only outputs and are recurrently fired. The remaining finite limits, those in $(a(b \cup c))^* a$, are not output traps because they immediately enable b and c. The remaining infinite limits are not output traps because they cease to fire one of the outputs after some finite prefix, but they recurrently enable both outputs. Intuitively, the remaining infinite limits 'owe' an output event and the remaining infinite limits are 'unfair' to either b or c.

Our liveness condition

Definition 3 *For networks S and I, we write $S \sqsubseteq_\lambda I$ if:*

$$\forall\, e \in \mathbf{lim}\,\|(S \cup I),\ ((\forall\, P \in I,\ e_P \in \mathbf{otp}\, P)\ \rightarrow\ (e_S \in \mathbf{otp}\,\|S)).$$

For networks S and I such that output consistency and safety are satisfied, i.e., such that $S \sqsubseteq_{\omega\sigma} I$, we say that I realizes *S* with liveness *if $S \sqsubseteq_\lambda I$.*

Informally speaking, we consider that liveness violations are caused by limits that are 'not live' for the specification, but are 'live' (!) for the implementation. The fact that sequences causing liveness faults need to be live for the implementation may seem counterintuitive, and is an important insight: liveness faults can be caused only by executions that can be generated by the implementation.

Examples of common liveness faults

To illustrate our liveness condition, we look at some of the possible liveness faults. We try to keep our examples very simple, so that we can study more of them. In addition to the examples in this section, there are several examples for the graph-theoretic form of our condition (see Sections 7 and 8).

Unfairness is basically a type of fault where one or more options of a specified choice is blocked forever. For an example of unfairness, consider a specification containing just a SELECTOR $\langle \{a\}, \{b,c\}, \mathbf{pref}(a(b \cup c))^* \rangle$ and an implementation containing just $P = \langle \{a\}, \{b,c\}, \mathbf{pref}(ab)^* \rangle$. Since the implementation element never issues a c, it is unfair for this specification. Our liveness condition detects this flaw, because the sequence $e = (ab)^\omega$ is in $\mathbf{lim}\,\|(S \cup I)$, e_P is in $\mathbf{otp}\, P$, but e_{SELECTOR} is not in $\mathbf{otp}$ SELECTOR.

A CLOCK can 'flood' the limits of a system with its output events, but that does not necessarily change the liveness properties of the system. Consider a slight modification of the example above, where the specification contains just a SELECTOR and the implementation contains two elements: $P = \langle \{a\}, \{b,c\}, \mathbf{pref}(ab)^* \rangle$ again and a CLOCK $\langle \emptyset, \{f\}, f^* \rangle$. See Figure 3 (a). Since this implementation never issues a c, it is unfair for this specification. Our liveness condition detects this flaw as follows. Let $e = (afb)^\omega$. We have that e is in $\mathbf{lim}\,\|(S \cup I)$, $e_P = (ab)^\omega$ is in $\mathbf{otp}\, P$, and $e_{\text{CLOCK}} = f^\omega$ is in $\mathbf{otp}$ CLOCK, but $e_{\text{SELECTOR}} = (ab)^\omega$ is not in $\mathbf{otp}$ SELECTOR. Hence, e violates our liveness condition, and, by that, the flaw is detected.

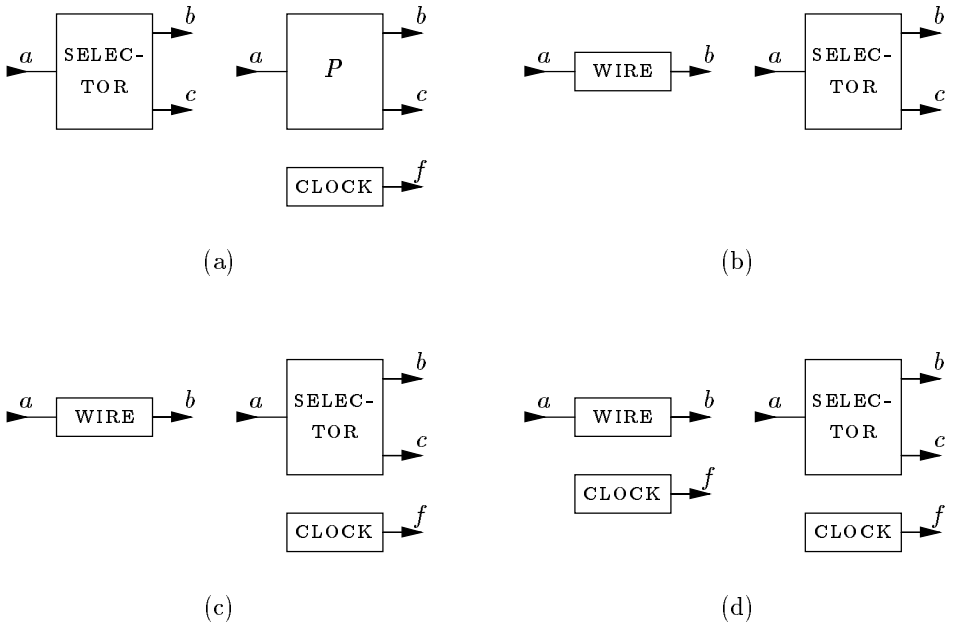


Figure 3: Examples of common liveness faults.

To forbid unfairness, is it sufficient to demand that the implementation be *capable* of producing every trace of the specification? The answer is *no*, and we produce a 'counterexample' by a slight modification of the example above. Consider a specification containing just a SELECTOR, and an implementation containing just $Q = \langle \{a\}, \{b, c, g, h\}, \mathbf{pref}(ag((b \cup c)a)^* \cup ah(ba)^*) \rangle$. After its first input event, Q decides whether to behave exactly like a SELECTOR or to 'be unfair,' like P in the example above. The choices of that decision are represented by g and h. (Recall that, for us, internal actions are the same as output actions because they are all driven by the device. Hence, g and h are in the output set. In formalisms with internal actions, g and h should be internal. With this modification, the present example can be used as a 'counterexample' in a model with internal symbols, too.) Hence, intuitively, this implementation has a danger of unfairness for this specification. This flaw is detected by our liveness condition. The composite of the specification and implementation elements is precisely Q. The sequence $ahb(ab)^\omega$ is a limit of Q, and is an output trap of the implementation, but its projection on the specification alphabet is not an output trap of the specification. However, this implementation satisfies the capability condition described above. In general, one can dodge the capability condition above by exhibiting implementations that can be fair but can also be unfair. Such flaws would pass the test of capability, but violate our liveness condition.

To illustrate deadlock-detection by our condition, consider the following example. Similar examples have been indicated previously in trace theory as limitations

of models that address safety concerns only (see for instance Example 5.5.1 in [Ve94]). Consider a specification containing just a WIRE $\langle\{a\}, \{b\}, \mathbf{pref}(ab)^*\rangle$ and an implementation containing just a SELECTOR $\langle\{a\}, \{b, c\}, \mathbf{pref}(a(b \cup c))^*\rangle$. See Figure 3 (b). After an a, the SELECTOR may choose c and block, while, at the 'interface' of the specification (actions a and b, no c) it seems that the device has received an a and then has blocked. Thus, the implementation has a danger of deadlock. Our liveness condition detects this fault: Sequence $e = ac$ is in $\mathbf{lim}\,\|(S \cup I)$, $e_{\text{SELECTOR}} = ac$ is in $\mathbf{otp}\,\text{SELECTOR}$, but $e_{\text{WIRE}} = a$ is not in $\mathbf{otp}\,\text{WIRE}$ because word a immediately enables output b.

A point of view which we reject is that deadlock can occur only where *all* processes in a system are blocked, i.e., none of them has to produce an output action. (See for instance [Ve94].) Consider a specification containing just a WIRE $\langle\{a\}, \{b\}, \mathbf{pref}(ab)^*\rangle$ and an implementation containing two elements: a SELECTOR $\langle\{a\}, \{b, c\}, \mathbf{pref}(a(b \cup c))^*\rangle$, and a CLOCK $\langle\emptyset, \{f\}, f^*\rangle$. See Figure 3 (c). Without the CLOCK, this realization has deadlock (see the previous example). Intuitively, introducing a CLOCK which does not interfere in any way with the rest of the system can neither repair nor change the nature of the fault: Although the CLOCK cannot be blocked, the system deadlocks. Formally, our liveness condition also detects this flaw and declares it a violation of liveness: Sequence $e = acf^\omega$ is in $\mathbf{lim}\,\|(S \cup I)$, $e_{\text{CLOCK}} = f^\omega$ is in $\mathbf{otp}\,\text{CLOCK}$, and $e_{\text{SELECTOR}} = ac$ is in $\mathbf{otp}\,\text{SELECTOR}$, but $e_{\text{WIRE}} = a$ is not in $\mathbf{otp}\,\text{WIRE}$.

In the example above, any number of f events can occur consecutively, while f is an output of the implementation but not an action of the specification; thus it can be viewed as an 'internal' action. One could object that the problem above ('local deadlock') can only occur where a string with unboundedly many internal actions and with no 'external' action is part of a complete execution (i.e., in a 'divergence' situation). However, we can adjust the example above to dismiss this objection: It suffices to make the CLOCK visible, i.e., modify the specification to be $\{\text{WIRE}\|\text{CLOCK}\}$ while the implementation remains $\{\text{SELECTOR}, \text{CLOCK}\}$. See Figure 3 (d). On the intuitive side, it seems that the introduction of a CLOCK that does not interfere with the rest of the system should have no effect on the correctness or on the type of flaw of that system. One verifies that the point of view does not permit to detect the flaw, but our liveness condition is violated.

Liveness in systems that are incorrect for other reasons

As mentioned in the definition, we restrict our liveness condition for specification-implementation pairs that satisfy safety and output consistency. Nevertheless, our liveness condition does not have an input control restriction, i.e., it also applies to systems that have dangling inputs.

To illustrate the problems with liveness for unsafe systems, consider the following example:

$$\begin{aligned} S &= \{\langle\{a\}, \{b\}, \{\varepsilon, a, ab\}\rangle\} \\ I &= \{\langle\{a\}, \{b\}, \{\varepsilon\}\rangle\} \end{aligned}$$

Here I does *not* realize S with safety because the trace a causes a safety violation on the implementation element. Therefore, this implementation is incorrect for this specification, and the pair is outside the domain of applicability of our liveness

condition. Let us see, however, what would happen if the safety restriction were not introduced. Formally, we have $S \sqsubseteq_\lambda I$ because the single output trap of the implementation is ε, which is an output trap of the specification. Intuitively, however, it can be argued that I is less live than S, because S can produce b's whereas S cannot. There are also objective difficulties caused by having $S \sqsubseteq_\lambda I$ in this case. For instance, let $N = \{\langle\{a\}, \{b\}, \{\varepsilon, a\}\rangle\}$. We have $S \sqsubseteq_\lambda I \sqsubseteq_\lambda N$, but $S \not\sqsubseteq_\lambda N$ and transitivity does not hold.

To illustrate the problems with liveness for systems without output consistency, consider the following example. Let $M = \{Q_1\}$, $N = \{Q_1, Q_2\}$, and $O = \{Q_2\}$, where:

$$\begin{aligned} Q_1 &= \langle \emptyset,\ \{a, b\},\ (a \cup b)^* \rangle \\ Q_2 &= \langle \emptyset,\ \{a, b\},\ b^* \rangle \end{aligned}$$

Since $\mathbf{o}Q_1 \cap \mathbf{o}Q_2 \neq \emptyset$, our liveness condition does not apply in this case. Let us see, however, what would happen if the output consistency restriction were not introduced. We have that $Q_1 \| Q_2 = Q_2$; thus $\mathbf{otp}\, Q_2 = \mathbf{otp}\, \| N$, and $N \sqsubseteq_\lambda O$. We have that $\mathbf{otp}\, Q_2 = \{\ e \in \{a, b\}^\omega \mid e \text{ fires } a \text{ infinitely many times and } e \text{ fires } b \text{ infinitely many times } \}$ and that $\mathbf{otp}\, Q_1 = \{b^\omega\}$. Thus, there is no sequence e such that $e_{Q_2} \in \mathbf{otp}\, Q_2$ and $e_{Q_1} \in \mathbf{otp}\, Q_1$. Hence, trivially, $M \sqsubseteq_\lambda N$. However, $M \not\sqsubseteq_\lambda O$ (unfairness) and transitivity does not hold.

In conclusion, our liveness condition needs safety and output consistency restrictions for transitivity. These restrictions are sufficient (see Theorem 4) and not severe (see the explanation accompanying Theorem 4). Informally speaking, the problem lies in the fact that $\|$ can introduce new output traps. In the example above, b^ω is an output trap in $\| N$ but not in all elements of N. The same problem would occur if a were an input in Q_2, but then safety would be violated. However, as shown in Theorem 4, this problem cannot occur if safety and output consistency are satisfied; hence the restriction.

Our liveness condition does not have an input control restriction. Some systems may be regarded as 'incorrect,' but they may be 'live.' Obvious examples are systems where the implementation has redundant elements which do not affect the outputs of the specification, either directly or indirectly. For instance, recall from Section 4 the WIRE represented by $\langle\{a\}, \{b\}, \mathbf{pref}(ab)^*\rangle$ and the XOR represented by $\langle\{c, d\}, \{e\}, \mathbf{pref}((c \cup d)(c \cup d \cup e))^*\rangle$. Consider $S = \{\text{WIRE}\}$ and $I = \{\text{WIRE}, \text{XOR}\}$. See Figure 1 (c). Intuitively, the XOR is completely disconnected from the WIRE in the implementation, and thus the implementation behaves irreproachably with respect to this specification. (Since actions c, d, e are not in the alphabet of the specification, their transitions are unspecified.) Formally, if the projection on $\{a, b\}$ of a sequence is an output trap of the implementation WIRE, then that projection is also an output trap of the (identical) specification WIRE. Thus, $S \sqsubseteq_\lambda I$, in agreement with our intuition.

Modeling power

Now we address the following modeling power problem: For which concurrent systems are the complete executions exactly the output traps? The key is Proposition 3 below, but unfortunately this point needs some informal considerations regarding the notion of 'complete execution.'

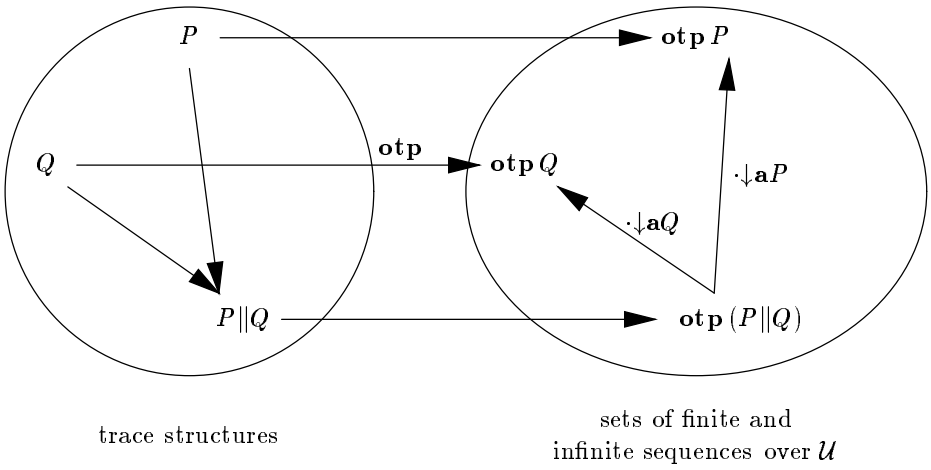


Figure 4: Output traps, parallel composition, and projection.

Proposition 3 *For networks S and I such that $S \sqsubseteq_{\omega\sigma} I$, and sequence $e \in \mathcal{U}^\infty$ such that $e_S \in \mathbf{lim}\,\|S$,*

$$(e_I \in \mathbf{otp}\,\|I) \;\leftrightarrow\; (\forall\, P \in I,\; e_P \in \mathbf{otp}\,P).$$

Proposition 3 is illustrated in Figure 4.

Admitting (informally) that the 'complete executions' of a concurrent system are those executions that are 'complete' for every element of that concurrent system, i.e., that project as complete executions on the alphabet of each element of the system, Proposition 3 has the following (informal) interpretation. Suppose network I realizes specification S with safety and output consistency. If, for every element P of a network I, the complete executions of P are exactly the output traps of P, then the complete executions of $\|I$ that are 'legal' for S are exactly the output traps of $\|I$. By 'legal for the specification' we mean 'can be generated in the specified environment.' Accordingly, the complete executions of $\|I$ correspond to the output traps of $\|I$, but only if they are limits of a specification which is realized by I with safety and output consistency. Since complete executions of $\|I$ that are not limits of the specification do not occur anyway, the only restriction is that of existence of a specification S such that $S \sqsubseteq_{\omega\sigma} I$.

Therefore, unfortunately, just like the liveness condition, this relationship between complete executions and output traps has safety and output consistency restrictions. Nevertheless, the restrictions are not severe, because safety and output consistency need to be satisfied anyway, for different reasons.

Now, to show that the relationship between complete executions and output traps occurs for a class of circuits, under the restrictions above, it suffices to check the basic components. For example, one verifies that the basic asynchronous components (for instance, the version in [Ve94]), satisfy this relationship (since the complete executions of these components were not precisely defined, we take as reference the intuitive descriptions in [Ve94]—notably that of a TOGGLE: "without ill effect, the [...] selector can be replaced, by, for instance, a toggle [...]." (p. 63)). Consequently, for any circuits formed with these components and that are safe and

output consistent for a specification, the 'legal' complete executions are exactly the output traps.

Theorems facilitating verification of liveness
We now state the $\cup$-compatibility and transitivity theorems for liveness.

Theorem 3 *For networks M, N, and O such that $M \sqsubseteq_\lambda N$, we have $M \cup O \sqsubseteq_\lambda N \cup O$.*

Theorem 4 *For networks M, N and O such that $M \sqsubseteq_{\omega\sigma\lambda} N$ and $N \sqsubseteq_{\omega\sigma\lambda} O$, we have $M \sqsubseteq_\lambda O$.*

Note that there are no restrictions for $\cup$-compatibility; this fact is surprising, just as it was in Theorem 1. Unfortunately, however, we had to introduce safety and connectivity restrictions for transitivity. Nevertheless, these restrictions are not severe because they are necessary correctness conditions themselves. Moreover, the safety restriction/condition has unrestricted $\cup$-compatibility and transitivity properties (see Section 4) and output consistency is easy to verify directly.

The following proposition provides another (simpler ?) form for our liveness condition, using the safety and connectivity restrictions. We use the initial form for proving the structure theorems and for discussion of liveness outside the restrictions, and we use this second form for automatic verification.

Proposition 4 *For networks M and N, if $M \sqsubseteq_{\omega\sigma} N$ then*

$$M \sqsubseteq_\lambda N \;\Leftrightarrow\; \{\|M\} \sqsubseteq_\lambda \{\|N\}.$$

Equivalently, for networks M and N such that $M \sqsubseteq_{\omega\sigma} N$, we have that:

$$M \sqsubseteq_\lambda N \;\Leftrightarrow\; \forall\, e \in \mathcal{U}^\infty,\; ((e_M \in \mathbf{lim}\,\|M \;\wedge\; e_N \in \mathbf{otp}\,\|N \;\rightarrow\; e_M \in \mathbf{otp}\,\|M)).$$

In words, every sequence that is 'live' for the implementation and 'legal' for the specification must be 'live' for the specification, too.

6 Modeling Non-deterministic Processes by Automata

The language-theoretic model we have used so far is convenient for the algebraic treatment and for handwritten proofs of correctness. For automatic verification, an automaton model seems more suitable. Another motivation for using an automaton model is that we define a graph-theoretic form for our condition and prove it equivalent to the language-theoretic form. This provides another test for the appropriateness of our liveness condition.

We define a model of *non*-deterministic concurrent systems using so-called behavior automata, which are formally incomplete deterministic finite automata. With an input/output distinction, these automata represent non-deterministic systems because, for example, they can have choices between edges marked with output symbols.

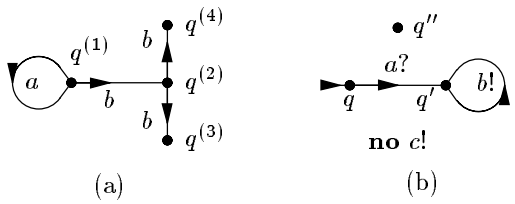


Figure 5: (a) A state graph; (b) a behavior automaton.

Our automaton model was inspired by the "state graphs" used previously in trace theory to represent trace structures having regular languages. To some extent, this model turned out to be similar to the I/O-automata in [LT87]. The main differences between I/O automata and our automata are that I/O automata can have infinitely many states, I/O automata require each input action to be enabled in each state, and I/O automata use "partitions of the locally-controlled actions" (which would correspond to partitions of the output alphabets in our model) to represent fairness properties.

Basic definitions

We define a *state graph* over a finite alphabet Σ as a pair $G = \langle \mathbf{st}\, G, \mathbf{ed}\, G \rangle$, where $\mathbf{st}\, G$ is a finite set of *states* and $\mathbf{ed}\, G \subseteq V \times \Sigma \times V$ is a (finite) set of labeled *edges*. If (q, b, q') is an edge, then b is its *label*. Note that some symbols of Σ might not appear as labels. An example of a state graph is given in Figure 5 (a), where $\Sigma \supseteq \{a, b\}$, $\mathbf{st}\, G = \{q^{(1)}, q^{(2)}, q^{(3)}, q^{(4)}\}$, and $\mathbf{ed}\, G = \{(q^{(1)}, a, q^{(1)}),\ (q^{(1)}, b, q^{(2)}),\ (q^{(2)}, b, q^{(3)}),\ (q^{(2)}, b, q^{(4)})\}$.

A state graph is *ambiguous* if two edges leaving a state have the same label. For example, the state graph in Figure 5 (a) is ambiguous. A state graph is *unambiguous* if it is not ambiguous.

A behavior automaton consists of an unambiguous state graph whose alphabet is partitioned into inputs and outputs, together with an initial state. Formally, a *behavior automaton* is a tuple $A = \langle \mathbf{i}\, A, \mathbf{o}A, \mathbf{st}\, A, \mathbf{ed}\, A, \mathbf{init}A \rangle$ such that $\mathbf{i}\, A$ and $\mathbf{o}A$ are finite and disjoint subsets of $\mathcal{U}$ and $\langle \mathbf{st}\, A, \mathbf{ed}\, A \rangle$ is an unambiguous state graph over $\mathbf{i}\, A \cup \mathbf{o}A$. We call $\mathbf{i}\, A$ the *input alphabet*, $\mathbf{o}A$ the *output alphabet*, $\mathbf{st}\, A$ the set of *states*, $\mathbf{ed}\, A$ the set of *edges*, and $\mathbf{init}A \in \mathbf{st}\, A$ the *initial state*. We use the same representation as for trace structures. For us, internal symbols are outputs, because they are driven by the device rather than the environment.

The unambiguity restriction means that behavior automata cannot directly represent systems where two options of a choice have the same label. (Still, one can use modeling tricks to represent such systems, as discussed in Section 9 and illustrated in Figure 10.) However, we consider such systems to be rather marginal, since actions of interest are normally denoted by different symbols. In particular, the options of a choice should be represented in our model by different output symbols. If the choice is internal to the implementation, the option symbols should be from the complement of the specification alphabet. Note the dissimilarity from CCS, which has only one internal symbol and thus distinguishes the options of an internal choice only by their external effects. In Section 9 we also state a version of our liveness condition in automata with ambiguous choice and with a CCS-

style silent action. The condition we have obtained is quite complex compared to traplock-freedom. We settled for the unambiguity restriction, for simplicity without a significant loss of modeling power.

A behavior automaton is rendered like its graph, except that: (a) the initial state is distinguished by an incoming arrow; (b) symbols have punctuation, ? for inputs and ! for outputs; and (c) unused alphabet symbols are listed below the graph. Figure 5 (b) shows a behavior automaton.

For a behavior automaton A we use the following notation. The *alphabet* of A, written $\mathbf{a}A$, is $\mathbf{i}\,A \cup \mathbf{o}A$; the *graph* of A, written $\mathbf{gr}\,A$, is $\langle \mathbf{st}\,A, \mathbf{ed}\,A \rangle$; the *language* of A, written $\mathbf{lg}\,A$, is the set of all traces spelled by finite paths in $\mathbf{gr}\,A$ that start in the initial state. Note that the language of a behavior automaton is always prefix-closed and contains ε. For example, let A denote the behavior automaton in Figure 5 (b); then $\mathbf{a}A = \{a, b, c\}$ and $\mathbf{lg}\,A = \mathbf{pref}(ab^*)$.

Trace structures of behavior automata

The semantics of behavior automata is given by their languages and alphabets. For behavior automaton A, we define the *trace structure* of A as $\mathbf{tr}\,A = \langle \mathbf{i}\,A, \mathbf{o}A, \mathbf{lg}\,A \rangle$. Note that $\mathbf{tr}\,A$ is a well-formed trace structure, i.e., $\mathbf{i}\,A \cap \mathbf{o}A = \emptyset$, $\mathbf{lg}\,A \subseteq (\mathbf{a}A)^*$ and $\mathbf{lg}\,A$ is non-empty (contains ε) and prefix-closed.

Subgraphs and knots

A *subgraph* of a behavior automaton A is a state graph G over $\mathbf{a}A$ such that $\mathbf{st}\,G \subseteq \mathbf{st}\,A$ and $\mathbf{ed}\,G \subseteq \mathbf{ed}\,A$. Note that the edges of G must be consistent with its states, because G is a state graph; however, not all edges of A between states of G must appear in G. Note that G is unambiguous, since $\mathbf{gr}\,A$ is unambiguous.

A subgraph G of a behavior automaton is *non-void* if G has at least one state. Note that a subgraph with one state and no edges is non-void, and that for behavior automaton A, $\mathbf{gr}\,A$ is non-void (contains at least the initial state). A subgraph G of a behavior automaton is *strongly connected* if, for every two states q and q' of G, there exists a path in G from q to q'. A subgraph G of a behavior automaton A is *reachable* if, for every state q of G, there exists a path in $\mathbf{gr}\,A$ from $\mathbf{init}A$ to q.

A *knot* in a behavior automaton is a non-void, reachable and strongly connected subgraph. For example, the behavior automaton in Figure 5 (b) has the following knots: $\langle \{q\}, \emptyset \rangle$, $\langle \{q'\}, \emptyset \rangle$, and $\langle \{q'\}, \{(q', b, q')\} \rangle$. The subgraph with only the state q'' is non-void and strongly connected but not a knot, because it is not reachable.

The leads-to operation

For behavior automaton A and trace t in $\mathbf{lg}\,A$, we define $A \diamond t$ to be the state of A at the end of the unique path starting in the initial state and spelling t; $\diamond$ is called the *leads-to* function of A. For example, if A is the behavior automaton in Figure 5 (b), then $A \diamond \varepsilon = q$ and $A \diamond abb = q'$. For arbitrary behavior automaton A, we have $A \diamond \varepsilon = \mathbf{init}A$.

The leads-to operation is extended to infinite sequences. For behavior automaton A and sequence e in $\mathbf{lim}\,\mathbf{tr}\,A$, we define $A \diamond e$ to be a subgraph of $\mathbf{gr}\,A$ such

that

$$\mathbf{st}\,(A \diamond e) = \{q \in \mathbf{st}\,A \mid \forall\, t \leq e,\ \exists\, u \text{ such that } tu \leq e \ \wedge\ A \diamond tu = q\}$$
$$\mathbf{ed}\,(A \diamond e) = \{(q, a, q') \in \mathbf{ed}\,A \mid \forall\, t \leq e,\ \exists\, u \text{ such that } tua \leq e \ \wedge\ A \diamond tu = q\}$$

If e is finite, $A \diamond e$ contains just one state and no edge, where the state is the same as that produced by the leads-to operation for traces. If e is infinite, $\mathbf{st}\,(A \diamond e)$ contains all states that are reached infinitely often by e, and $\mathbf{ed}\,(A \diamond e)$ contains all edges that are passed infinitely often by e, informally speaking.

The following lemmas link the knots in A to sequences in $\mathbf{lim\,tr}\,A$ by means of the leads-to operation. These lemmas are the basis of the connection between our language-theoretic and graph-theoretic treatments of liveness.

Proposition 5 *For behavior automaton A and sequence e in $\mathbf{lim\,tr}\,A$, $A \diamond e$ is a knot.*

Proposition 6 *For behavior automaton A and knot G in A, there exists a sequence e in $\mathbf{lim\,tr}\,A$ such that $A \diamond e = G$.*

Parallel composition

In the following we define a parallel composition operation on behavior automata and we link it to the parallel composition of trace structures.

A triple $e = (q, a, q')$ is *compatible* with a behavior automaton A if either $e \in \mathbf{ed}\,A$ (i.e., the transition in question actually occurs in A), or we have both $a \notin \mathbf{a}A$ and $q' = q$ (i.e., the symbol a is not in the alphabet of A, in which case A 'does not mind' a occurring, and the state of A cannot be affected by this occurrence).

The *parallel composition* of two behavior automata A and B is a behavior automaton $A\|B$ such that:

$$\begin{array}{lcl}
\mathbf{i}\,(A\|B) & = & (\mathbf{i}\,A \cup \mathbf{i}\,B) - (\mathbf{o}A \cup \mathbf{o}B); \\
\mathbf{o}(A\|B) & = & \mathbf{o}A \cup \mathbf{o}B; \\
\mathbf{st}\,(A\|B) & = & \mathbf{st}\,A \times \mathbf{st}\,B; \\
\mathbf{ed}\,(A\|B) & = & \{\ ((p, q), a, (p', q')) \in \mathbf{st}\,(A\|B) \times \mathbf{a}(A\|B) \times \mathbf{st}\,(A\|B) \mid \\
 & & \qquad (p, a, p') \text{ is compatible with } A, \text{ and} \\
 & & \qquad (q, a, q') \text{ is compatible with } B\ \}; \\
\mathbf{init}(A\|B) & = & (\mathbf{init}A, \mathbf{init}B).
\end{array}$$

Informally speaking, the parallel composition describes behaviors consistent with both operands. As we did for trace structures, we call the result of parallel composition a *composite*.

Note that the case where $a \notin \mathbf{a}A$ and $a \notin \mathbf{a}B$ cannot occur in the definition of $\mathbf{ed}\,(A\|B)$, because $a \in \mathbf{a}\ (A\|B)$. One verifies that the other properties of a behavior automaton are satisfied by $A\|B$; thus, $A\|B$ is well-formed. Note also that an input of a process connected to an output of another process is not an input of the composite, but all process outputs are outputs of the composite.

For behavior automata A and B and state $o = (p, q) \in \mathbf{st}\,(A\|B)$, we use the notations $o_A = p$ and $o_B = q$. For trace t and sequence e, we use the notation $t_A = t{\downarrow}\mathbf{a}A$ and $e_A = e{\downarrow}\mathbf{a}A$.

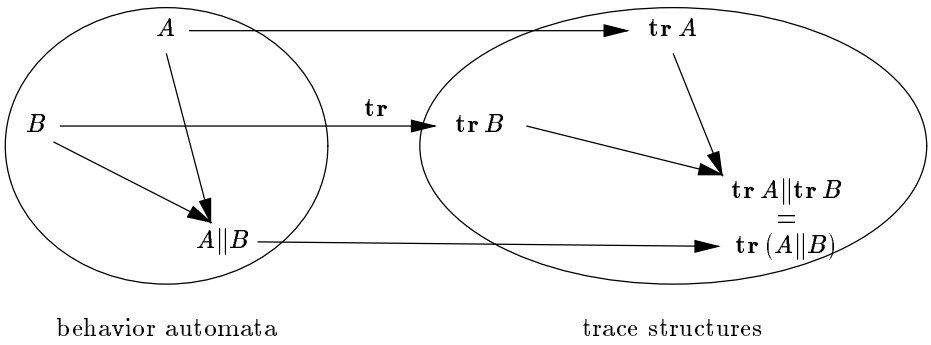


Figure 6: Commutative diagram of parallel compositions.

Lemma 5 *For behavior automata A and B and word t in $\mathbf{lg}\,(A\|B)$, we have*

$$t_A \in \mathbf{lg}\,A \ \wedge\ ((A\|B) \diamond t)_A = A \diamond t_A, \ \textit{and}$$
$$t_B \in \mathbf{lg}\,B \ \wedge\ ((A\|B) \diamond t)_B = B \diamond t_B.$$

The following theorem links the parallel compositions of behavior automata and trace structures by the **tr** semantics, as illustrated by the commutative diagram in Figure 6. To prove it, we use the following lemma:

Lemma 6 *For behavior automata A and B and word t in $(\mathbf{a}(A\|B))^*$, we have*

$$t \in \mathbf{lg}\,A\|B \ \Leftrightarrow\ t_A \in \mathbf{lg}\,A \ \wedge\ t_B \in \mathbf{lg}\,B.$$

Theorem 5 *For behavior automata A and B, we have $\mathbf{tr}\,(A\|B) = \mathbf{tr}\,A \parallel \mathbf{tr}\,B$.*

Theorem 5 is important because it shows that the parallel compositions of behavior automata and trace structures model the same operation.

Knot projections

We now define knot projections and relate them to sequence projections. For behavior automata A and B and knot G in $A\|B$, we define subgraph G_A of A such that:

$$\begin{aligned}\mathbf{st}\,G_A &= \{p \in \mathbf{st}\,A \mid \exists\ q \in \mathbf{st}\,B \text{ such that } (p,q) \in \mathbf{st}\,(A\|B)\}\\ \mathbf{ed}\,G_A &= \{(p,b,p') \in \mathbf{ed}\,A \mid\\ &\qquad \exists\ q,q' \in \mathbf{st}\,B \text{ such that } ((p,q),b,(p',q')) \in \mathbf{ed}\,(A\|B)\}\end{aligned}$$

and we define subgraph G_B of B similarly.

Projections of knots are knots:

Proposition 7 *For behavior automata A and B and knot G in $A\|B$, G_A is a knot in A and G_B is a knot in B.*

The following lemma links knot projections to sequence projections.

Lemma 7 *For behavior automata A and B and sequence e in $\mathbf{lim}\,\mathbf{tr}\,(A\|B)$, $((A\|B)\diamond e)_A = A \diamond e_A$ and $((A\|B) \diamond e)_B = B \diamond e_B$.*

7 Traplock-freedom

In this section we define and discuss *traplock-freedom*, a graph-theoretic form of our liveness condition.

Traps

For behavior automaton A, subgraph G of A, and state p of A, the *set of fired symbols* of G is $\mathbf{fi}\, G = \{a \in \mathbf{a}A \mid \exists\, (p', a, p'') \in \mathbf{ed}\, G\}$, the *set of enabled symbols* of p in A is $\mathbf{en}_A p = \{a \in \mathbf{a}A \mid \exists\, p' \in \mathbf{st}\, A$ such that $(p, a, p') \in \mathbf{ed}\, A\}$, and the *set of enabled symbols* of G in A is $\mathbf{en}_A G = \bigcup_{p' \in \mathbf{st}\, G} \mathbf{en}_A p'$. For example, let A be the behavior automaton represented in Figure 5 (b), let $G = \langle \{q'\}, \{(q', b, q')\}\rangle$, and let $H = \langle \{q'\}, \emptyset\rangle$. We have $\mathbf{fi}\, G = \{b\}$, $\mathbf{fi}\, H = \emptyset$, and $\mathbf{en}_A G = \mathbf{en}_A H = \{b\}$.

For alphabet Σ, behavior automaton A, and knot G in A, G is a *trap* in A with respect to Σ if $\mathbf{en}_A G \cap \Sigma \subseteq \mathbf{fi}\, G$. (Since we always have $\mathbf{fi}\, G \subseteq \mathbf{en}_A G$, the subset relationship in the definitions of traps can be replaced by equality.) G is an *output trap* in A if $\mathbf{en}_A G \cap \mathbf{o}A \subseteq \mathbf{fi}\, G$. For example, let behavior automaton A and knots G and H be as in the example in the paragraph above. Since $b \in \mathbf{o}A$, G is an output trap in A but H is not.

Lemma 9 *For behavior automaton A, knot G in A, and sequence e in* $\mathbf{lim}\, \mathbf{tr}\, A$ *such that $A \diamond e = G$, we have that G is an output trap in A iff e is an output trap in* $\mathbf{tr}\, A$.

Traplock-freedom

Definition 4 *For behavior automata S and I, I is* traplock-free *for S if, for every knot G in $S \| I$ such that G_I is an output trap in I, G_S is an output trap in S.*

Just as we did for $\sqsubseteq_\lambda$, we restrict the applicability of the traplock-freedom condition to specification-implementation pairs of behavior automata that satisfy $\{\mathbf{tr}\, S\} \sqsubseteq_{\omega\sigma} \{\mathbf{tr}\, I\}$, i.e., satisfy the safety and the output-consistency conditions.

Intuitively, traplock-freedom demands that every trap in the implementation correspond to a trap in the specification. If this condition is *not* satisfied, then the implementation allows the execution point to remain forever in a trap, while the specification expects the execution point to eventually leave the set of specification states corresponding to the implementation trap.

In the examples in Figure 7, we compare the traplock-freedom condition to our intuitive notion of liveness. The implementation in Figure 7 (a) appears to be correct with respect to its specification. Accordingly, that implementation is traplock-free for its specification, because the only trap in the implementation is the whole graph, corresponding to the whole graph of the specification, which is also a trap. The implementation in Figure 7 (b) has danger of deadlock, because that implementation can block after the occurrence of an (internal) output event c, while the specification does not indicate the possibility of such blocking. Note that the implementation in Figure 7 (b) has traplock (is not traplock-free) for that specification, as demonstrated by the corresponding knots in the rectangles. The implementation in Figure 7 (c) is unfair, because the implementation cannot issue

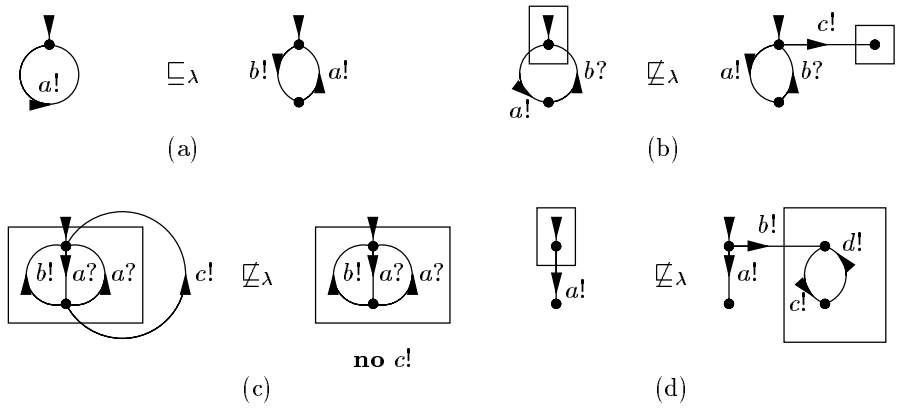


Figure 7: Intuitive liveness compared to traplock-freedom.

the output c, while the specification expects a choice between b and c after each occurrence of a. Again, the implementation has traplock for the specification. The implementation in Figure 7 (d) appears to have a deadlock-like flaw, because the current state can become trapped in the framed cycle, while an a output is expected by the specification. Again, the implementation has traplock for the specification.

The examples in Figure 7 are 'mainstream' situations, relatively easy to model. Several 'extreme' situations and subtle points in modeling liveness are also disussed in Section 8.

In conjunction with Proposition 4, the following theorem shows the equivalence of our traplock-freedom condition to our liveness condition.

Theorem 6 *For behavior automata S and I, I is traplock-free for S iff* $\{\mathbf{tr}\, S\} \sqsubseteq_\lambda \{\mathbf{tr}\, I\}$.

Theorem 6 has important consequences. First, it proves the equivalence of two forms in essentially different models. Second, this equivalence facilitates verification by allowing the application of the (language-theoretic) structured verification theorems in Section 5 to the (graph-theoretic) automatic verification method in Section 10. Finally, since two isomorphic behavior automata have the same language, this equivalence shows that traplock-freedom is invariant under automaton isomorphisms—a necessary property, since isomorphic automata normally represent the same process.

8 Alternative Liveness Conditions

In this section we consider some variations to our condition for liveness, and point out their disadvantages. We also point out deficiencies of the condition for liveness in [LT87] and another condition, which appears in many forms in the literature,

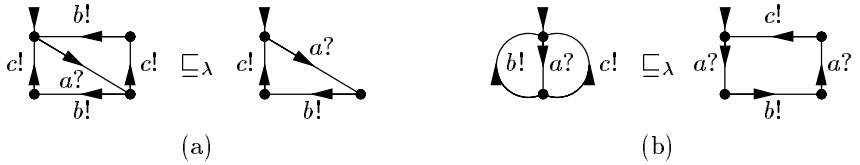


Figure 8: Traplock-freedom vs. wraplock-freedom.

and we call generically *capability*. We also discuss our liveness condition on several 'extreme' cases.

Liveness with respect to words

The first variation is obtained by using liveness with respect to words instead of symbols, as we sketch below. For a subgraph G of a behavior automaton A, the set $\mathbf{enw}_A G$ of *enabled words* of G in A is the set of all words spelled by finite paths in A starting in states of G. The set $\mathbf{fiw}\, G$ of *fired words* of G is the set of all words spelled by finite paths in G. A knot G in a behavior automaton A is an *output wrap* (short for 'output-word trap') if $\mathbf{enw}_A G \cap (\mathbf{o}A)^* \subseteq \mathbf{fiw}\, G$.

Definition 5 *An implementation I is* wraplock-free *for a specification S if, for every knot G in $S||I$, if G_I is an output wrap in I then G_S is an output wrap in S.*

The examples in Figure 8 illustrate the difference between traplock-freedom and wraplock-freedom. In Figure 8 (a), we represent a specification and an implementation of a FORK. The specification allows the two output actions to occur in either order, but only one order is possible in the implementation. We take the position that this is not a flaw. Such a case is quite usual in digital circuits, where a FORK may have unequal delays in the branches, but the relationship between the two branch delays is not known at design time. Formally, our traplock-freedom condition is satisfied, but wraplock-freedom is not. In Figure 8 (b), we represent a SELECTOR implemented by a TOGGLE. The TOGGLE alternates the b and c outputs instead of choosing non-deterministically between them, like the SELECTOR does. In agreement with an opinion expressed in [Ve94] (p. 63), we consider that TOGGLE is a good implementation for the SELECTOR. Our reason is that SELECTORs represent arbiters, and, for arbitration purposes, periodicity is an acceptable substitute for randomness. Formally, our traplock-freedom condition is again satisfied, but wraplock-freedom is not.

Why would traplock-freedom seem more suitable for practical concurrent systems than wraplock-freedom? Why would liveness with respect to *symbols* seem more suitable than liveness with respect to *words*? A possible explanation may have to do with psychology: We presume that designers denote by symbols, rather than interleavings of symbols, the actions of interest. Accordingly, we presume that the implicit liveness properties normally refer to single events rather than bursts of events. Consequently, requirements like 'this event must eventually occur if this state is reached infinitely often' appear to be appropriate liveness conditions, while

requirements like 'all interleavings of these concurrent events must eventually occur' appear to be undesirable as liveness conditions.

Accordingly, in our paradigm, the term "something good" from the informal description of liveness in [LL90] (see the Introduction) means an occurrence of an action.

Weak liveness

Another condition can be obtained by using weak instead of strong liveness. For a subgraph G of a behavior automaton A, the set $\mathbf{cen}_A G$ of *continuously enabled symbols* of G in A is the set of all symbols enabled by all states of G. A knot G in a behavior automaton A is an *weak output trap* if $\mathbf{cen}_A G \cap (\mathbf{o}A)^* \subseteq \mathbf{fiw}\, G$.

Definition 6 *An implementation I is* weakly traplock-free *for a specification S if, for every knot G in $S||I$, if G_I is a weak output trap in I then G_S is a weak output trap in S.*

To point out a deficiency of this definition, we consider the example in Figure 7 (c). Intuitively, that implementation appears to be unfair, and thus it is not live (in agreement with the traplock-freedom condition) for that specification. Our liveness condition detects this flaw, as discussed in Section 7. However, that implementation is weakly traplock-free for that specification: All weak output traps of the implementation contain the non-initial implementation state. All implementation knots that contain the non-initial implementation state correspond to specification knots which contain the non-initial specification state. Finally, one verifies that all such specification knots are weak output traps.

Capability

Another candidate for a liveness condition, which we also ruled out as insufficient in Section 5, is a condition that the implementation be capable of producing every trace of the specification. By the examples in Figure 8, that condition is not even a necessary correctness condition. For Figure 8 (a), trace *acb* can be produced by the specification but not by the implementation. For Figure 8 (b), trace *abab* can be produced by the specification but not by the implementation.

In Figure 9 (a), the implementation cannot produce output b. However, it is impossible to tell it apart from the specification by an external experiment, because the absence of a 'reset' action indicates that the specified process is to be used only *once*, i.e., to produce only one output event. The implementation can only produce a, but that does not contradict this 'one-shot' specification. Correspondingly, our traplock-freedom condition is satisfied. Examples of such processes are very rare. An example may be a surprise box: The figure inside is fixed by fabrication, and is used to surprise only once. If the surprise box can be reused, or there are infinitely (statistically sufficiently) many boxes available, then the appropriate specification is that in Figure 9 (b), because it indicates explicitly the 'reset' action which is, in fact, performed. In Figure 9 (b), the implementation is unfair for the specification (as can be noticed by iterated experiments), and, accordingly, it is not traplock-free for the specification.

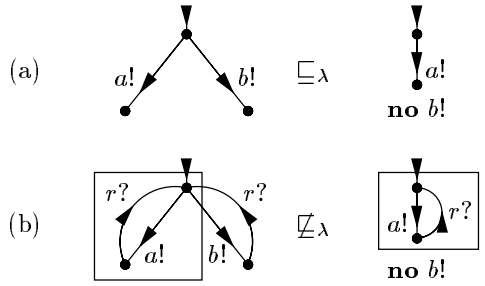


Figure 9: Choice: (a) not repeatable; (b) repeatable.

The definition in [LT87]

We consider again the example in Figure 7 (c). The "locally-controlled actions" from [LT87] correspond to outputs of behavior automata. The model in [LT87] requires the users to specify partitions of these output sets, corresponding to the output sets of the elements of the modeled concurrent system. Consider again the processes represented by the behavior automata in Figure 7 (c), and let us represent them using the formalism in [LT87]. Both the specification and the implementation have a single element, thus their respective "partitions of locally-controlled actions" are trivial: They have one "class" each, consisting of all the outputs ($\{b, c\}$). The formal definition of liveness in [LT87] is satisfied in this example. However, this implementation should be considered not live for this specification according to our intuition (because it is unfair) and even according to intuition described [LT87]: If the (modified) SELECTOR (the specification) is used for arbitration purposes, then the implementation in the (modified) example has the flaw of "lockout," which [LT87] considers to be a violation of liveness. (Our liveness condition does detect this flaw: See the explanations of Figure 7.)

Although we have chosen the partitions of locally controlled actions as indicated in [LT87], that indication is informal. Nevertheless, there is only one other way in which these partitions could be chosen: $\{b\}$ $\{c\}$. Even in that case, the formal condition in [LT87] is satisfied in spite of the flaw, and our objection stands.

In general, classical cases of unfairness which pass undetected by the liveness condition in [LT87] can be constructed using specification-implementation pairs such that a choice in the specification has one option, say c, disabled in the implementation. (Set $\{c\}$ can be a class in the partition of locally-controlled actions.) Often, such a choice is part of a cycle in the specification, and, at some point in the cycle, c in also disabled in the specification. In such cases, the condition in [LT87] is satisfied, and does not detect the flaw because basically every behavior of the specification is considered fair with respect to the option c, according to [LT87].

9 Modeling Power

Syntactic vs. semantic non-determinism
We model non-determinism by output or internal choice, where the options have different labels by output actions, as they are in Figure 7 (b), (c), and (d). Recall that we consider internal actions to be outputs, too, because they are driven by the device rather than by the environment. This way, we associate deterministic automata to non-deterministic processes. (By a non-deterministic process we mean, informally, a process that has some choice over its future behavior, as opposed to a deterministic process, whose behavior is entirely determined by the environment.)

From a semantic point of view, the formal (syntactic) term "deterministic" is misleading when applied to automata which can choose between several outputs. For that, we prefer the term *unambiguous*, reflecting the fact that different options of a choice that the automaton can take have to be labeled distinctly, unambiguously.

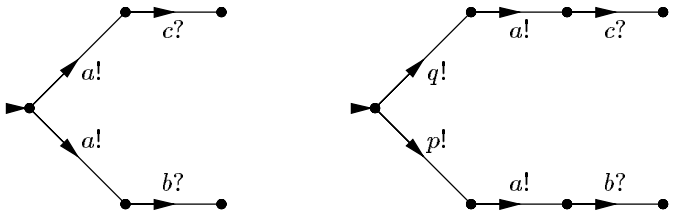
However, one may object that our unambiguous automata cannot express behaviors like that in Figure 10 (a) (which can stand for, say, a vending machine). That objection is superficial: Such a behavior can be simply expressed by introducing two fresh internal actions, $p!$ and $q!$, standing for the options of the internal choice in the initial state. In fact, the least we can ask as part of the representation discipline is that important events, such as the options of this internal choice, be explicitly represented, not omitted from the automaton. This way, one obtains a deterministic automaton, drawn in Figure 10 (b), which can replace the automaton in Figure 10 (a) in the representation of an implementation, provided the symbols p and q are not used anywhere else in the concurrent system to be verified.

In general, one can always transform a non-deterministic automaton into a deterministic automaton by using fresh symbols to label the previously 'invisible' choices, just as we did in the example in Figure 10. Note that this transformation is different from the usual determinization procedures for finite automata and takes only linear time, because we introduce fresh symbols.

Weighting the disadvantage of having to represent explicitly the 'invisible' choices, versus the advantage of having simpler algorithms and a simpler correctness condition, we decided to use unambiguous (deterministic) behavior automata as a model of non-deterministic processes.

Traplock-freedom with syntactically non-deterministic automata
Nevertheless, our traplock-freedom condition can be easily extended to a more complex automaton model, allowing ambiguous choices. In the following, we make this extension, for the following reasons: First, we believe that there is no reason to restrict a correctness condition to just one or two models of concurrency. Second, in our behavior automaton model projection can be used as a constructor of intermediate specifications of concurrent systems, but one has to ensure that projections preserve liveness properties by applying our liveness condition. A model using syntactically non-deterministic automata has the potential to preserve liveness properties under projection.

A *branching automaton* is a tuple $A = \langle \mathbf{i}\,A, \mathbf{o}A, \mathbf{st}\,A, \mathbf{ed}\,A, \mathbf{init}A \rangle$, where $\mathbf{i}\,A$ is the *input alphabet* $\mathbf{o}A$ is the *output alphabet*, $\mathbf{st}\,A$ is the set of *states*, $\mathbf{ed}\,A$ is the



(a) (b)

Figure 10: Modeling invisible choices by fresh device-driven actions.

set of *edges*, and $\mathbf{init}A \in \mathbf{st}\,A$ is the *initial state*, such that $\mathbf{i}\,A$ and $\mathbf{o}A$ are finite and disjoint subsets of $\mathcal{U}$ and $\langle \mathbf{st}\,A, \mathbf{ed}\,A\rangle$ is a state graph over $\mathbf{i}\,A \cup \mathbf{o}A$. Note that branching automata differ from behavior automata only by allowing ambiguous state graphs. Branching automata are similar to the "behavior schemas" in [BS95], if the "choice sets" are taken to be the sets of edges with the same label and the same source state.

Note that behavior automata are a particular case of branching automata. An example of a branching automaton which is not a behavior automaton is shown in Figure 10 (a).

An *option* of a branching automaton A is a behavior automaton B such that

$$\begin{array}{lcl} \mathbf{i}\,B & = & \mathbf{i}\,A \\ \mathbf{o}B & = & \mathbf{o}A \\ \mathbf{st}\,B & = & \mathbf{st}\,A \\ \mathbf{init}B & = & \mathbf{init}A \end{array}$$

and such that

$$\begin{array}{l} \mathbf{ed}\,B \subseteq \mathbf{ed}\,A, \text{ and} \\ \forall\,(q,a,q') \in \mathbf{ed}\,A,\ \exists\,q'' \in \mathbf{st}\,B \text{ such that } (q,a,q'') \in \mathbf{ed}\,B \end{array}$$

The options of a branching automaton are similar to the "options of a behavior schema" in [BS95]. For example, the options of the branching automaton in Figure 10 (a) are the behavior automata in Figure 11.

Note that a behavior automaton has only one option, itself.

We now extend the definition of traplock-freedom to branching automata.

Definition 7 *A branching automaton I is* traplock-free *for a branching automaton S if, for every option B of I, there exists an option A of S such that B is traplock-free for A.*

Note that, since a behavior automaton has only one option, Definition 7 agrees with Definition 4 if S and I are behavior automata.

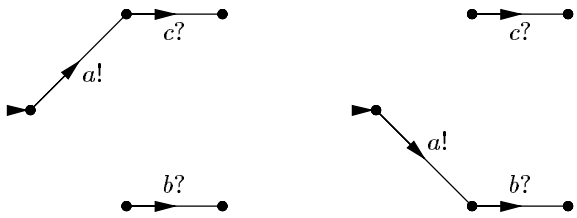


Figure 11: Options of a branching automaton.

Silent actions

We can extend further the branching automata by introducing a CCS-style silent action τ and trivially generalizing traplock-freedom accordingly. An *extended branching automaton* is defined like a branching automaton (see above) except that $\mathbf{ed}\,A \subseteq \mathbf{st}\,A \times (\mathbf{i}\,A \cup \mathbf{o}A \cup \{\tau\}) \times \mathbf{st}\,A$ (compare to $\mathbf{ed}\,A \subseteq \mathbf{st}\,A \times (\mathbf{i}\,A \cup \mathbf{o}A) \times \mathbf{st}\,A$ for behavior and branching automata). An *extended behavior automaton* is an extended branching automaton A such that $\forall\,(q,a,q'),(q,b,q'') \in \mathbf{st}\,A$, if $a = b$ or $a = \tau$ or $b = \tau$ then $q' = q''$. (Note that extended behavior automata are formally deterministic, but, just like behavior automata, they stand for non-deterministic processes.) Extended branching automata are also similar to the "behavior schemas" in [BS95], if the "choice blocks" are taken to be sets of edges with the same source state and with either the same label or the label τ. (The silent action can be viewed as a 'wildcard': Edges bearing the label τ will belong to all choice blocks for a certain source state.)

An *option* of an extended branching automaton A is an extended behavior automaton B such that:

$$\begin{array}{lcl} \mathbf{i}\,B & = & \mathbf{i}\,A \\ \mathbf{o}B & = & \mathbf{o}A \\ \mathbf{st}\,B & = & \mathbf{st}\,A \\ \mathbf{init}B & = & \mathbf{init}A \end{array}$$

and such that:

$$\begin{array}{l} \mathbf{ed}\,B \subseteq \mathbf{ed}\,A, \text{ and} \\ \forall\,(q,a,q') \in \mathbf{ed}\,A, \\ \qquad \exists\,q'' \in \mathbf{st}\,B \text{ such that either } (q,a,q'') \in \mathbf{ed}\,B \text{ or } (q,\tau,q'') \in \mathbf{ed}\,B \end{array}$$

For example, the options of the extended branching automaton $\langle \emptyset, \{a\}, \{q,q',q''\}, \{(q,a,q'),(q,\tau,q'')\}, q\rangle$ (Figure 12 (a)) are $\langle \emptyset, \{a\}, \{q,q',q''\}, \{(q,a,q')\}, q\rangle$ (Figure 12 (b)) and $\langle \emptyset, \{a\}, \{q,q',q''\}, \{(q,\tau,q'')\}, q\rangle$ (Figure 12 (c)). (In Figure 12, the silent action τ is represented without punctuation, because it is neither an input action nor an output action.)

A *compaction* of an extended behavior automaton A is a behavior automaton

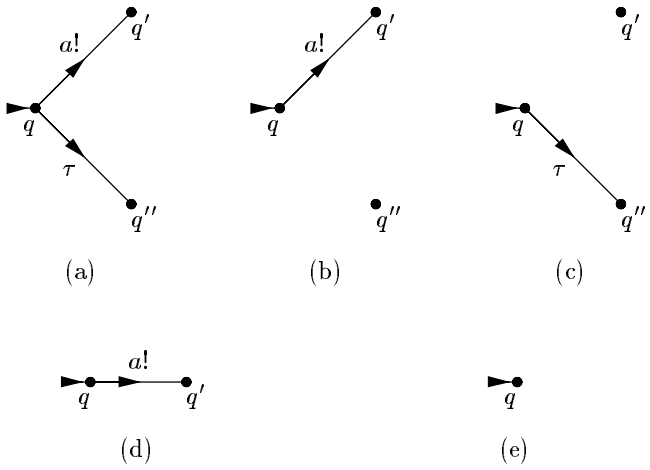


Figure 12: Extended branching automata, options and compactions.

B such that there exists a function $h : \mathbf{st}\, A \to \mathbf{st}\, B$ such that:

$$
\begin{array}{l}
h(\mathbf{init} A) = \mathbf{init} B, \text{ and} \\
\forall\ (p, a, p') \in \mathbf{ed}\ A, \\
\qquad (\quad ((a \neq \tau)\ \rightarrow\ ((h(p), a, h(p')) \in \mathbf{ed}\ B)) \quad \wedge \\
\qquad\quad\ ((a = \tau)\ \rightarrow\ (h(p) = h(p'))) \qquad)
\end{array}
$$

Note that compaction is well-defined: The resulting automaton is unambiguous. For example, possible compactions of the options of the extended branching automaton in the previous example are $\langle \emptyset, \{a\}, \{q, q'\}, \{(q, a, q')\}, q\rangle$ (Figure 12 (d)) and $\langle \emptyset, \{a\}, \{q\}, \emptyset, q\rangle$ (Figure 12 (e)), respectively. In the cases where the extended behavior automaton A corresponds to a "fundamental mode behavior" from [BS95], a compaction of A can be related to a "direct behaviour" obtained by the construction in [BS95], p. 250.

Accordingly, we define an extended branching automaton I to be *traplock-free* for an extended branching automaton S if, for every compaction B' of every option B of I, there exists a compaction A' of an option A of S such that B' is traplock-free for A'.

While the last extension is fine for modeling deadlock, it leads to problems with modeling fairness just like CCS does, because τ confuses different kinds of internal actions. Therefore, we encourage the reader to use Definition 7 instead, and simulate silent actions by ambiguous choices. Or, better, use (deterministic) behavior automata to model non-deterministic processes, as we do in the rest of this paper.

10 An Algorithm for Verification of Liveness

In this section we introduce an algorithm, based on our formalization, for verifying liveness.

Parallel composition
A polynomial-time algorithm for parallel compositions can be constructed straightforwardly using the definition of parallel compositions of behavior automata in Section 6.

Traplock-freedom
It remains to derive an algorithm to verify traplock-freedom of two behavior automata S and I. Considering in turn all output traps in I, all knots that are not output traps in S, or all knots in $S||I$ would be very inefficient, because there are exponentially many of them in the worst case. Therefore, we have to use a more elaborate method, as shown in the algorithm below.

```
predicate is_traplock-free?(S, I)
    for each b in oS do
        build C by removing from gr (S||I) all edges firing b
        for each state (q', q'') of C such that b ∈ en_S q' do
            repeat
                let H be the strongly connected component
                        of (q', q'') in C
                if H_I is an output trap in I then
                        return false
                build C removing from H all states (p', p'')
                        such that en_I p'' ∩ oI ⊈ fi H
            while (q', q'') ∈ st C
    return true
```

Correctness
To sketch a proof of partial correctness, we first note that the traplock problem amounts to the existence, for some $b \in \mathbf{o}S$ and $(q', q'') \in \mathbf{st}\,(S||I)$ such that $b \in \mathbf{en}_S\, q'$, of a knot H in $S||I$ with the properties: (1) H passes through (q', q''); (2) H does not fire b (which causes H_S to be not an output trap in S); and (3), H_I is an output trap in I. Suppose there exists such an H; we show that the algorithm will not overlook it. Let $H^0, H^1, \ldots$ be the candidates considered by the algorithm, and $C^0, C^1, \ldots$ the parts of $S||I$ considered. We prove that, for any $i \geq 0$, if $\mathbf{st}\,H \subseteq \mathbf{st}\,H^i$, then $\mathbf{st}\,H \subseteq \mathbf{st}\,H^{i+1}$. We have that $\mathbf{fi}\,H \subseteq \mathbf{fi}\,H^i$, because H^i contains all the edges of $S||I$ whose source states are in $\mathbf{st}\,H^i$ and do not fire b. Thus, if H contained any of the states removed from H^i to obtain C^{i+1}, then H_I would not be an output trap in I, and we have a contradiction. Thus, if $\mathbf{st}\,H \subseteq \mathbf{st}\,H^i$ then $\mathbf{st}\,H \subseteq \mathbf{st}\,C^{i+1}$. Furthermore, since H^{i+1} is the strongly connected component of (q', q'') in C^{i+1}, and H is strongly connected and contains (q', q''), we have that, if $\mathbf{st}\,H \subseteq \mathbf{st}\,C^{i+1}$, then $\mathbf{st}\,H \subseteq \mathbf{st}\,H^{i+1}$. Consequently, the algorithm

can only return false if there exists a knot H with the properties (1), (2) and (3). Conversely, the algorithm cannot return false if there exists no knot H with the properties (1), (2) and (3) (see the condition for the **return** statement to be performed). In conclusion, the algorithm can only return the correct answer if it terminates.

To sketch a proof of termination, we note that, at each iteration of the **repeat** cycle, at least one state of H is removed. If H_I is not an output trap in I, then at least one state p'' of H_I has the property $\mathbf{en}_I p'' \cap \mathbf{o}I \not\subseteq \mathbf{fi}\, H$ and thus is removed. This proves the algorithm terminates within a bounded time.

Time and space analysis

The time complexity can be assessed as follows. For a subgraph P, let the *size* of P be $|P| = |\mathbf{st}\, P| + |\mathbf{ed}\, P|$. Searching for strongly connected components of P is known to take a linear time, i.e., $\mathcal{O}(|P|)$, and with a small constant. The body of **repeat** thus takes $\mathcal{O}(|C|)$ time. Because at least one state must be removed each time, the iterations of **repeat** are $\mathcal{O}(|C|)$. The iterations of **for** (q', q'') are also $\mathcal{O}(|C|)$; the body of **for** b takes thus $\mathcal{O}(|C|^3)$. The iterations of **for** b are $\mathcal{O}(|\mathbf{o}S|)$. Therefore, the total worst-case time complexity of our algorithm is $\mathcal{O}(|\mathbf{gr}\,(S \| I)|^3 \cdot |\mathbf{o}S|)$. Also, the constants hidden in these $\mathcal{O}$ computations are small.

The worst-case space complexity of our algorithm is $\mathcal{O}(|\mathbf{gr}\,(S \| I)|)$, because this is the worst-case size of C, the largest of the (constant number of) data structures in this algorithm.

Practical considerations

The method for verification of liveness sketched as a remote possibility in [Di89] is acknowledged to be impractical, because its worst-case time cost grows exponentially with the square of the size of the specification. To assess the practicality of our algorithm, we compare it to the verification algorithm for *safety* in [Di89]. The (worst-case) time cost of our verification method is no larger than T^3 times a small linear factor, where T is the (worst-case) running time of Dill's safety algorithm. The space cost of our liveness algorithm is the same as that for the safety algorithm in [Di89]. Note, however, that a liveness condition is of a different nature than a safety condition, and seems to be more complex *a priori*. Also, in the average case our algorithm does not visit those states of I that cannot be reached according to the specification; this feature is important because, as pointed out in [Di89], most states of a flawed implementation are typically such spurious states. Still, although the costs of our algorithm for traplock-freedom are polynomial in the sizes of S and I, the costs of the overall verification method should include computing I as a parallel composition. These costs are exponential in the number of components, just like the successful method for safety in [Di89]. This state-explosion problem is partly remedied by modular and hierarchical verification (as we have shown that our safety and liveness conditions have the required algebraic properties).

Using this algorithm, we have implemented a program for the verification of traplock-freedom.

11 Conclusions

Contributions

In this paper we have defined a liveness condition which can be decided from common representations of concurrent systems, such as networks of finite automata. This was previously thought impossible, because such representations are ambiguous with respect to liveness properties ([Bl86] argues that for trace theory). We resolve the ambiguity by assigning augmented semantics to a finitary representation of concurrent systems (see Sections 1 and 5). This semantics represents liveness properties that seem to be implicitly assumed for many digital circuits and other practical concurrent systems. Therefore, this semantics can be enforced as a representation discipline. By defining this semantics, we suggest what may be an appropriate and complete specification of liveness properties of a concurrent system, by analogy with systems that we have studied.

To define this semantics, we introduce strong liveness—a generic property which admits a unified form for finite and infinite sequences of events. The label "strongly live" has been used previously (e.g. in [YLS92]), to denote a completely different concept. Our reason for reusing this label is that strong liveness relates to strong fairness in a manner similar to the way liveness relates to fairness.

Basically, our extended semantics specifies what we consider to be the 'complete' executions of a concurrent system for which only finite executions are given. Apart from studying examples, we show this semantics is preserved by parallel composition, under certain restrictions (Proposition 3). (The restrictions are not severe, because they amount to necessary correctness conditions.) Thus, it can be shown that an entire class of concurrent systems obeys this semantics, provided their basic components or 'building blocks' obey this semantics. We have argued in Section 5 for a large class of asynchronous circuits that they obey this semantics.

Our extended semantics is not closed under projection or hiding operators on processes, because such operators can 'hide' deadlocks and other liveness violations. Still, as discussed in Section 3, such operators are not needed in our method for modular and hierarchical verification, because the processes we compare can have arbitrary, unrelated alphabets. (Note the dissimilarity from previous treatments of liveness, which impose various connectivity constraints on the processes they can couple or compare. For example, [Di89] only compares processes with the same input and output alphabets.) Nevertheless, such operators can be used in our verification method to build intermediate specifications, which are to be compared to their respective implementations, in a modular and hierarchical manner. Intermediate specifications may be built or guessed using any other method, as well.

We also derive a graph-theoretic form for our liveness condition, in addition to the language-theoretic form based on strong liveness. (Actually, we derived the graph-theoretic form first, but we chose to present the language-theoretic form first because we found the treatment of the algebraic properties of the condition to be more comfortable using languages than using graphs.)

In support of our liveness condition, we study several examples in the two essentially different models and we prove that our condition satisfies certain desirable algebraic properties. We prove sufficient theorems for the modular and hierarchical verification of our liveness condition.

We present, prove and analyze a verification algorithm for our liveness condition.

We define a safety condition. Although reasonable conditions for safety have been previously defined in trace theory, our safety condition is mentioned as a contribution because, unlike the previous conditions, it has *no* connectivity (structure) restrictions. We also provide theorems for the modular and hierarchical verification of our safety condition, which have no connectivity restrictions (Section 4). This absence of connectivity restrictions may be surprising, especially in the theorem referring to modularity.

Clarification

Our automata are not automata over infinite objects (e.g. [Th90]). The semantics of our automata are only alphabets and languages of finite words (Section 6, definition of $\mathbf{tr}$). The main objective of our paper is to define and verify liveness in terms of common concurrent system representations, which specify finite executions only; ω-automata are not appropriate for this approach.

Further work

Unfortunately for the theoretician, our liveness condition is not decoupled from other correctness concerns (unlike our safety condition). To achieve this decoupling, our liveness condition needs to be extended for concurrent systems that do not satisfy safety and output consistency. Such an extension should make sense intuitively and should provide transitivity and $\cup$-compatibility theorems for liveness that do not involve other correctness conditions, just like our safety condition. Unfortunately, the extension that uses the same $\sqsubseteq_\lambda$ relation over unsafe systems or over systems without output consistency does not satisfy these criteria.

A generalization of our liveness condition to arbitrary concurrent systems does not seem very promising, because it would boil down to "all liveness requirements have to be satisfied if all liveness constraints are satisfied," where the liveness requirements and constraints would need to be specified by the users, possibly as sets of complete executions. Such user-directed approaches have been taken before and have the disadvantages listed in Section 1. Nevertheless, various intermediate generalizations can be considered, to give some more specification freedom to the users and to achieve closure of the model under interesting operators such as reflection. One easy extension is to augment the trace structures with a set of 'live' actions, to represent inputs or outputs with respect to which traps are defined. Allowing the users this small degree of freedom in specifying liveness properties by alphabet distinctions does not seem to put hard demands on the users and does not trivialize the liveness condition. The bottom line is that alphabet extensions to a partial-execution model seem acceptable.

Acknowledgements

We are grateful for critical evaluations of this work to Joanne Atlee, David Dill, and to the members of the Maveric group at the University of Waterloo, especially Igor Benko, Rob Berks and John Segers. We are indebted to Jo Ebergen and Charles Molnar for very important comments and suggestions.

Appendix A. Trace Structure Proofs

For word t and alphabets Σ and Γ, one verifies, by structural induction on t, that $(t{\downarrow}\Sigma){\downarrow}\Gamma = t{\downarrow}(\Sigma \cap \Gamma)$.

Proposition 1 *Parallel composition of trace structures is idempotent, commutative, and associative.*

Proof For the properties above, the parts referring to alphabets are not difficult to verify, using set theory. We only discuss here the parts referring to the languages.

Let P, Q, and R be trace structures. To show idempotence, one notes that any trace t of P satisfies $t_P = t$; therefore, the traces of $P \| P$ are precisely the traces of P. Commutativity follows from the commutativity of the set and logical operators in the definition of parallel composition. To show associativity, one first notes that the traces in $\mathbf{lg}\,((P\|Q)\|R)$ are exactly the words t in $(\mathbf{a}((P\|Q)\|R))^*$ that satisfy $t{\downarrow}(\mathbf{a}(P\|Q)){\downarrow}\mathbf{a}P \in \mathbf{lg}\,P$, $t{\downarrow}(\mathbf{a}(P\|Q)){\downarrow}\mathbf{a}Q \in \mathbf{lg}\,Q$, and $t_R \in \mathbf{lg}\,R$. Then, one verifies that the alphabet of a composite is the union of the alphabets of the composed elements, thus $\mathbf{a}((P\|Q)\|R) = \mathbf{a}P \cup \mathbf{a}Q \cup \mathbf{a}R$, and also $t{\downarrow}(\mathbf{a}(P\|Q)){\downarrow}\mathbf{a}P = t_P$ and $t{\downarrow}(\mathbf{a}(P\|Q)){\downarrow}\mathbf{a}Q = t_Q$. Consequently, $\mathbf{lg}\,((P\|Q)\|R) = \{t \in (\mathbf{a}P \cup \mathbf{a}Q \cup \mathbf{a}R)^* \mid t_P \in \mathbf{lg}\,P \ \wedge\ t_Q \in \mathbf{lg}\,Q \ \wedge\ t_R \in \mathbf{lg}\,R\}$. Since this form is symmetrical in P, Q, and R, one verifies similarly that $\mathbf{lg}\,(P\|(Q\|R)) = \{t \in (\mathbf{a}P \cup \mathbf{a}Q \cup \mathbf{a}R)^* \mid t_P \in \mathbf{lg}\,P \ \wedge\ t_Q \in \mathbf{lg}\,Q \ \wedge\ t_R \in \mathbf{lg}\,R\}$, and associativity is established. □

Theorem 1 *For networks M, N, and O such that $M \sqsubseteq_\sigma N$, we have $M \cup O \sqsubseteq_\sigma N \cup O$.*

Proof Let word t be such that t satisfies the precondition of $M \cup O \sqsubseteq_\sigma N \cup O$, i.e., such that $\forall\ P \in N \cup O \cup \overline{M \cup O}$, $t_P \in \mathbf{lg}\,P \cdot (\mathbf{i}\,P \cup \{\varepsilon\})$. Taking $P = \overline{\|(M \cup O)}$, we obtain $t_{M\cup O} \in \mathbf{lg}\,\overline{\|(M \cup O)} \cdot (\mathbf{i}\,\overline{\|(M \cup O)} \cup \{\varepsilon\}) = \mathbf{lg}\,\|(M \cup O) \cdot (\mathbf{o}\|(M \cup O) \cup \{\varepsilon\})$. We split this set and obtain two cases for t:

Case 1 If $t_{M\cup O} \in \mathbf{lg}\,\overline{\|(M \cup O)}$, then $t_M \in \mathbf{lg}\,\|M$ and $t_O \in \mathbf{lg}\,\|O$. Thus, $t_M \in \mathbf{lg}\,\overline{\|M} \subseteq \mathbf{lg}\,\overline{\|M} \cdot (\mathbf{i}\,\overline{\|M} \cup \{\varepsilon\})$. Since $\forall\ P \in N$, $t_P \in \mathbf{lg}\,P \cdot (\mathbf{i}\,P \cup \{\varepsilon\})$ (precondition, taking $P \in N$) and $M \sqsubseteq_\sigma N$, we have $t_N \in \mathbf{lg}\,\|N$ also.

Case 2 If $t_{M\cup O} \in \mathbf{lg}\,\|(M \cup O) \cdot \mathbf{o}\|(M \cup O)$, then let $t = uav$, where $a \in \mathbf{a}\|(M \cup O)$ and $v_{M\cup O} = \varepsilon$. We have $t_{M\cup O} = u_{M\cup O}a$, thus $u_{M\cup O} \in \mathbf{lg}\,\|(M \cup O)$ and $a \in \mathbf{o}\overline{\|(M \cup O)}$. One verifies that, consequently, $t_M = (ua)_M \in \mathbf{lg}\,\|M \cdot (\mathbf{o}\|M \cup \{\varepsilon\}) = \mathbf{lg}\,\overline{\|M} \cdot (\mathbf{i}\,\overline{\|M} \cup \{\varepsilon\})$ and that $\forall\ P \in O$, $t_P \in \mathbf{lg}\,P \cdot (\mathbf{o}P \cup \{\varepsilon\})$. Now, taking $P \in O$ in the precondition, we also have that $\forall\ P \in O$, $t_P \in \mathbf{lg}\,P \cdot (\mathbf{i}\,P \cup \{\varepsilon\})$, and, since $\mathbf{i}\,P \cap \mathbf{o}P = \emptyset$, we have $\forall\ P \in O$, $t_P \in \mathbf{lg}\,P$, i.e., $t_O \in \mathbf{lg}\,\|O$. By $M \sqsubseteq_\sigma N$, knowing that $t_M \in \mathbf{lg}\,\overline{\|M} \cdot (\mathbf{i}\,\overline{\|M} \cup \{\varepsilon\})$ and that $\forall\ P \in N$, $t_P \in \mathbf{lg}\,P \cdot (\mathbf{i}\,P \cup \{\varepsilon\})$ (precondition, taking $P \in N$), we have $t_M \in \mathbf{lg}\,\overline{\|M} = \mathbf{lg}\,\|M$ and $t_N \in \mathbf{lg}\,\|N$.

In both cases, from $t_M \in \mathbf{lg}\,\overline{\|M}$, $t_N \in \mathbf{lg}\,\|N$, and $t_O \in \mathbf{lg}\,\|O$ it follows that $t_{M\cup O} \in \mathbf{lg}\,\|(M \cup O) = \mathbf{lg}\,\overline{\|M \cup O}$ and $t_{N\cup O\cup\overline{M\cup O}} \in \mathbf{lg}\,\|(N \cup O \cup \overline{M \cup O})$, which is the postcondition of $M \cup O \sqsubseteq_\sigma N \cup O$. □

The following technical lemma will be used in the proofs of transitivity for safety and liveness.

Lemma 1 *For networks M, N, and O such that $M \sqsubseteq_\sigma N$ and $N \sqsubseteq_\sigma O$ we have*

$\forall\, t \in \mathcal{U}^*$,
if $t_M \in \mathbf{lg}\,\overline{\|M} \cdot (\mathbf{i}\,\overline{\|M} \cup \{\varepsilon\})$ *and* $(\forall\, P \in O,\ t_P \in \mathbf{lg}\,P \cdot (\mathbf{i}\,P \cup \{\varepsilon\}))$,
then $t_N \in \mathbf{lg}\,\|N$.

Proof Let $t \in \mathcal{U}^*$ such that $t_M \in \mathbf{lg}\,\overline{\|M} \cdot (\mathbf{i}\,\overline{\|M} \cup \{\varepsilon\})$ and $(\forall\, P \in O,\ t_P \in \mathbf{lg}\,P \cdot (\mathbf{i}\,P \cup \{\varepsilon\}))$. We show by structural induction that every prefix u of t (in particular, t itself!) satisfies $u_N \in \mathbf{lg}\,\|N$.

Basis: $u = \varepsilon = u_N \in \mathbf{lg}\,\|N$.

Step: Let $u \in \mathcal{U}^*$ and $a \in \mathcal{U}$ such that $ua \leq t$ and $u_N \in \mathbf{lg}\,\|N$. We show $(ua)_N \in \mathbf{lg}\,\|N$. Note that, for every trace structure Q, the language $\mathbf{lg}\,Q \cdot (\mathbf{i}\,Q \cup \{\varepsilon\})$ is prefix-closed; therefore, $(ua)_M \in \mathbf{lg}\,\overline{\|M} \cdot (\mathbf{i}\,\overline{\|M} \cup \{\varepsilon\})$ and $\forall\, P \in O$, $(ua)_P \in \mathbf{lg}\,P \cdot (\mathbf{i}\,P \cup \{\varepsilon\})$. We consider three cases for a:

Case 1: $a \notin \mathbf{a}\|N$. Then, $(ua)_N = u_N \in \mathbf{lg}\,\|N$.

Case 2: $a \in \mathbf{o}\|N$. Since $u_N \in \mathbf{lg}\,\|N = \mathbf{lg}\,\overline{\|N}$ and $\mathbf{o}\|N = \mathbf{i}\,\overline{\|N}$, we have $(ua)_N \in \mathbf{lg}\,\overline{\|N} \cdot (\mathbf{i}\,\overline{\|N} \cup \{\varepsilon\})$. Since $(\forall\, P \in O,\ (ua)_P \in \mathbf{lg}\,P \cdot (\mathbf{i}\,P \cup \{\varepsilon\}))$ and $N \sqsubseteq_\sigma O$, we have $(ua)_{\overline{N} \cup O} \in \mathbf{lg}\,\|(\overline{N} \cup O)$. Thus, $(ua)_N \in \mathbf{lg}\,\|N$.

Case 3: $a \in \mathbf{i}\,\|N$. Then, $\forall\, P \in N$, $a \notin \mathbf{o}P$. Since $u_N \in \mathbf{lg}\,\|N$, we have $(\forall\, P \in N,\ u_P \in \mathbf{lg}\,P)$. Thus, $(\forall\, P \in N,\ (ua)_P \in \mathbf{lg}\,P \cdot (\mathbf{i}\,P \cup \{\varepsilon\}))$. Since $(ua)_M \in \mathbf{lg}\,\overline{\|M} \cdot (\mathbf{i}\,\overline{\|M} \cup \{\varepsilon\})$ and $M \sqsubseteq_\sigma N$, we have $(ua)_{\overline{M} \cup N} \in \mathbf{lg}\,\|(\overline{M} \cup N)$. Thus, $(ua)_N \in \mathbf{lg}\,\|N$. □

Theorem 2 *For networks M, N, and O such that $M \sqsubseteq_\sigma N \ \wedge\ N \sqsubseteq_\sigma O$, we have $M \sqsubseteq_\sigma O$.*

Proof Let $t \in \mathcal{U}^*$ such that t satisfies the precondition of $M \sqsubseteq_\sigma O$, i.e., such that $t_M \in \mathbf{lg}\,\overline{\|M} \cdot (\mathbf{i}\,\overline{\|M} \cup \{\varepsilon\})$ and $(\forall\, P \in O, t_P \in \mathbf{lg}\,P \cdot (\mathbf{i}\,P \cup \{\varepsilon\}))$.

By Lemma 1, we have $t_N \in \mathbf{lg}\,\|N$. Therefore, $\forall\, P \in N$, $t_P \in \mathbf{lg}\,P \subseteq \mathbf{lg}\,P \cdot (\mathbf{i}\,P \cup \{\varepsilon\})$. Since $t_M \in \mathbf{lg}\,\overline{\|M} \cdot (\mathbf{i}\,\overline{\|M} \cup \{\varepsilon\})$ and $M \sqsubseteq_\sigma N$, we deduce $t_M \in \mathbf{lg}\,\|M$. Also, from $t_N \in \mathbf{lg}\,\|N$, we have $t_N \in \mathbf{lg}\,\overline{\|N} \subseteq \mathbf{lg}\,\overline{\|N} \cdot (\mathbf{i}\,\overline{\|N} \cup \{\varepsilon\})$. Since $(\forall\, P \in O$, $t_P \in \mathbf{lg}\,P \cdot (\mathbf{i}\,P \cup \{\varepsilon\}))$ and $N \sqsubseteq_\sigma O$, we deduce $t_O \in \mathbf{lg}\,\|O$.

From $t_M \in \mathbf{lg}\,\|M$ and $t_O \in \mathbf{lg}\,\|O$, we deduce $t_{\overline{M} \cup O} \in \mathbf{lg}\,\|(\overline{M} \cup O)$, i.e., t satisfies the postcondition of $M \sqsubseteq_\sigma O$. □

For sequence $e \in \mathcal{U}^\infty$, alphabets Σ and Γ such that $\Sigma \subseteq \Gamma$, and finite prefix t' of e_Σ, one verifies, by structural induction, that there exists a finite prefix t of e_Γ such that $t{\downarrow}\Sigma = t'$.

Proposition 2 *For trace structures P and Q,*
(a) $\mathbf{lim}\,(P\|Q) = \{e \in (\mathbf{a}P \cup \mathbf{a}Q)^\infty \mid e_P \in \mathbf{lim}\,P \ \wedge\ e_Q \in \mathbf{lim}\,Q\}$
(b) $\mathbf{lim}\,\overline{P} = \mathbf{lim}\,P$.

Proof
Part (a)
(⇒) Let $e \in \mathcal{U}^\infty$ such that $e_{P\|Q} \in \mathbf{lg}\,(P\|Q)$ and let t' be a finite prefix of e_P. There exists a finite prefix t of e such that $t_P = t'$. Since $e_{P\|Q} \in \mathbf{lim}\,(P\|Q)$, we

have $t_{P\|Q} \in \mathbf{lg}\,(P\|Q)$, thus $t_P = t' \in \mathbf{lg}\,P$. Since t' was arbitrary, every finite prefix of e_P is in $\mathbf{lg}\,P$. Consequently, $e_P \in \mathbf{lim}\,P$.

One proves $e_Q \in \mathbf{lim}\,Q$ similarly.

($\Leftarrow$) Let $e \in \mathcal{U}^\infty$ such that $e_P \in \mathbf{lim}\,P$ and $e_Q \in \mathbf{lim}\,Q$, and let t be a finite prefix of e. We have that $t_P \leq e_P$ and $t_Q \leq e_Q$. It follows that $t_P \in \mathbf{lg}\,P$ and $t_Q \in \mathbf{lg}\,Q$, thus $t_{P\|Q} \in \mathbf{lg}\,(P\|Q)$. Since t was arbitrary, every finite prefix of e is in $\mathbf{lg}\,(P\|Q)$. Consequently, $e_{P\|Q} \in \mathbf{lim}\,(P\|Q)$.

Part (b) This property follows from the facts that **lim** is uniquely determined by **lg** and that, for every trace structure P, $\mathbf{lg}\,\overline{P} = \mathbf{lg}\,P$. □

Lemma 2 *For network N and sequence $e \in \mathcal{U}^\infty$ such that $\forall\, P \in N,\ e_P \in \mathbf{otp}\,P$, we have that $e_N \in \mathbf{otp}\,\|N$.*

Proof Let sequence e as above and let $a \in \mathbf{o}\|N$ such that e_N recurrently enables a in $\mathbf{lg}\,\|N$. Since $\mathbf{o}\|N = \bigcup_{P \in N} \mathbf{o}P$, there exists $Q \in N$ such that $a \in \mathbf{o}Q$. We have that e_Q recurrently enables a in Q: Since $(\forall\, t \in e_N\ \exists\, u$ such that $(tu \leq e_N \wedge tua \in \mathbf{lg}\,\|N))$ and $\mathbf{a}\|N \supseteq \mathbf{a}Q$, we have $(\forall\, t' \leq e_Q\ \exists\, u'$ such that $(t'u' \leq e_Q \wedge t'u'a \in \mathbf{lg}\,Q))$. Since $e_Q \in \mathbf{otp}\,Q$, we have that e_Q recurrently fires a. Since $\mathbf{a}\|N \supseteq \mathbf{a}Q$, we conclude that e_N recurrently fires a, too. □

Lemma 3 *For networks M and N, and sequence $e \in \mathcal{U}^\infty$ such that $e_{M\cup N} \in \mathbf{lim}\,\|(M \cup N)$, if $M \sqsubseteq_{\omega\sigma} N$ and $e_N \in \mathbf{otp}\,\|N$, then, $\forall\, P \in N,\ e_P \in \mathbf{otp}\,P$.*

Proof
Let $P \in N$ and $a \in \mathbf{o}P$ such that e_P recurrently enables a in $\mathbf{lg}\,P$. We need to prove a is recurrently fired by e_P. For that, we show a is recurrently enabled by e_N in $\mathbf{lg}\,\|N$.

Let $u \leq e$ such that $(ua)_P \in \mathbf{lg}\,P$. Since $M \sqsubseteq_\omega N$, we have $a \notin \mathbf{o}\overline{\|M}$ and $\forall\, Q \in N - \{P\},\ a \notin \mathbf{o}Q$. Since $e_N \in \mathbf{lim}\,\|N$ and $e_M \in \mathbf{lim}\,\|M$, we have $\forall\, Q \in N \cup \overline{M},\ u_Q \in \mathbf{lg}\,Q$. Consequently, $\forall\, Q \in N \cup \overline{M},\ (ua)_Q \in \mathbf{lg}\,Q \cdot (\mathbf{i}\,Q \cup \{\varepsilon\})$. By $M \sqsubseteq_\sigma N$ it follows that $(ua)_{N\cup M} \in \mathbf{lg}\,\|(N \cup M)$, thus $(ua)_N \in \mathbf{lg}\,\|N$.

For every prefix t of e there exists v such that $tv \leq e$ and $(tva)_P \in \mathbf{lg}\,P$ (i.e., a is recurrently enabled by e_P in P). Letting $u = tv$ in the paragraph above, we have $(tva)_N \in \mathbf{lg}\,\|N$. Consequently, $\forall\, t' \leq e_N$, $\exists\, v'$ such that $(t'v' \leq e_N\ \wedge\ (t'v'a)_N \in \mathbf{lg}\,\|N)$, i.e., e_N recurrently enables a in $\mathbf{lg}\,\|N$. Since $a \in \mathbf{o}\|N$ and $e_N \in \mathbf{otp}\,\|N$, we have that e_N recurrently fires a. Since $a \in \mathbf{a}P$, we conclude that a is recurrently fired by e_P. □

Proposition 3 *For networks S and I such that $S \sqsubseteq_{\omega\sigma} I$, and sequence $e \in \mathcal{U}^\infty$ such that $e_S \in \mathbf{lim}\,\|S$,*

$$(e_I \in \mathbf{otp}\,\|I) \ \leftrightarrow\ (\forall\, P \in I,\ e_P \in \mathbf{otp}\,P).$$

Proof This proposition follows immediately from Lemma 2 and Lemma 3. □

Theorem 3 *For networks M, N, and O such that $M \sqsubseteq_\lambda N$, we have $M \cup O \sqsubseteq_\lambda N \cup O$.*

Proof Let $e \in \mathcal{U}^\infty$ such that $e_{M \cup N \cup O} \in \mathbf{lim} \,\|(M \cup N \cup O)$ and such that $\forall\, P \in N \cup O,\ e \in \mathbf{otp}\, P$. Since $M \sqsubseteq_\lambda N$, we have $e_M \in \mathbf{otp} \,\|M$. By applying Lemma 2 twice, we obtain $e_{M \cup O} \in \mathbf{otp} \,\|(\{\|M\} \cup \{\|O\})$. Since $\|(\{\|M\} \cup \{\|O\}) = \|(M \cup O)$, we conclude that $e_{M \cup O} \in \mathbf{otp} \,\|(M \cup O)$. □

The following lemma will be used to exploit the safety restrictions in the proof of transitivity for liveness.

Lemma 4 *For networks M, N, and O such that*

$$M \sqsubseteq_\sigma N \ \wedge \ N \sqsubseteq_\sigma O,$$

we have

$$\forall\, e \in \mathcal{U}^\infty, \quad (e_{M \cup O} \in \mathbf{lim} \,\|(M \cup O) \ \rightarrow \ e_N \in \mathbf{lim} \,\|N).$$

Proof
Let $e \in \mathcal{U}^\infty$ such that $e_{M \cup O} \in \mathbf{lim} \,\|(M \cup O)$, and let $t' \leq e_N$. There exists $t \leq e$ such that $t_N = t'$. We have $t_{M \cup O} \leq e_{M \cup O}$ and thus $t_{M \cup O} \in \mathbf{lg} \,\|(M \cup O)$, i.e., $t_M \in \mathbf{lg} \,\|M$ and $t_O \in \mathbf{lg} \,\|O$. By Lemma 1, we have $t' = t_N \in \mathbf{lg} \,\|N$. Since t' was arbitrary, we have $e_N \in \mathbf{lim} \,\|N$. □

Theorem 4 *For networks M, N and O such that $M \sqsubseteq_{\omega\sigma\lambda} N$ and $N \sqsubseteq_{\omega\sigma\lambda} O$, we have $M \sqsubseteq_\lambda O$.*

Proof Let $e \in \mathcal{U}^\infty$ such that $e_{M \cup O} \in \mathbf{lim} \,\|(M \cup O)$ and $\forall\, P \in O,\ e \in \mathbf{otp}\, P$. By Lemma 4, we have $e_N \in \mathbf{lim} \,\|N$. Since $N \sqsubseteq_\lambda O$, we have $e_N \in \mathbf{otp} \,\|N$. By Lemma 3, we have $\forall\, P \in N,\ e_P \in \mathbf{otp}\, P$. Since $M \sqsubseteq_\lambda N$, we conclude that $e_M \in \mathbf{otp} \,\|M$. □

Proposition 4 *For networks M and N, if $M \sqsubseteq_{\omega\sigma} N$ then*

$$M \sqsubseteq_\lambda N \ \Leftrightarrow \ \{\|M\} \sqsubseteq_\lambda \{\|N\}.$$

Proof **($\Rightarrow$)** Let $e \in \mathbf{lim} \,\|(\{\|M\} \cup \{\|N\})$ such that $e_N \in \mathbf{otp} \,\|N$. By Lemma 3, $\forall\, P \in N,\ e_P \in \mathbf{otp}\, P$. Also, we have $e \in \mathbf{lim} \,\|(M \cup N)$, because $\|(M \cup N) = \|(\{\|M\} \cup \{\|N\})$. Since $M \sqsubseteq_\lambda N$, we conclude $e_M \in \mathbf{otp} \,\|M$.

($\Leftarrow$) Let $e \in \mathbf{lim} \,\|(M \cup N)$ such that $\forall\, P \in N,\ e_P \in \mathbf{otp}\, P$. By Lemma 2, we have $e_N \in \mathbf{otp} \,\|N$. Also, we have $e \in \mathbf{lim} \,\|(\{\|M\} \cup \{\|N\})$ (see the ($\Rightarrow$) part). Since $\{\|M\} \sqsubseteq_\lambda \{\|N\}$, we conclude that $e_M \in \mathbf{otp} \,\|M$. □

Appendix B. Behavior Automaton Proofs

Proposition 5 *For behavior automaton A and sequence e in* $\mathbf{lim\,tr}\,A$, $A \diamond e$ *is a knot.*

Proof If e is infinite, the fact that $A \diamond e$ is a well-formed state graph follows from noting that, for any edge (q, a, q') of $\mathbf{ed}\,(A \diamond e)$, that edge is passed infinitely often by e, thus both q and q' are reached infinitely often by e, thus both q and q' are in $\mathbf{st}\,(A \diamond e)$. If e is finite, $A \diamond e$ is trivially a valid state graph because it has just one state and no edge.

The non-voidness of $A \diamond e$ is trivial if e is finite, because $A \diamond e$ has one state. If e is infinite, the non-voidness follows from the pigeonhole principle: Considering that our automata have only finitely many states, at least one state of A will be reached infinitely often by e.

The reachability follows from the fact that every state recurrently reached by e must be led-to by at least one prefix of e.

The strong connectivity is trivial if e is finite, because e has just one state.

To show strong connectivity for the case with an infinite e, suppose $\mathbf{st}\,(A \diamond e)$ could be partitioned into non-void sets S_1 and S_2 ($S_1 \cap S_2 = \emptyset \ \wedge\ S_1 \cup S_2 = \mathbf{st}\,(A \diamond e)$) such that there exists no path from states of S_1 to states of S_2. However, since e reaches infinitely many times the states of S_1 and the states of S_2, e passes infinitely often through the set of edges of $\mathbf{gr}\,A$ that leave S_1, i.e., the set of edges (q, a, q') such that $q \in S_1$, but $q' \notin S_1$. Since there are only finitely many such edges, by the pigeonhole principle again, at least one such edge will be passed infinitely often, and thus has to be in $\mathbf{ed}\,(A \diamond e)$. For such an edge, $q \in S_1$ and q' cannot be in S_1. Since $A \diamond e$ is a valid state graph, we have $q' \in S_2$, and thus the edge (q, a, q') constitutes a path from S_1 to S_2, contradiction.

Finally, for two states p and p' in $\mathbf{st}\,(A \diamond e)$, suppose that p' were not reachable from p by a path in $A \diamond e$. It follows that $\mathbf{st}\,(A \diamond e)$ can be partitioned into the set S_1 of states reachable from p (containing at least p) and the set S_2 of states not reachable from p (containing at least p'), with no path from S_1 to S_2, which leads to a contradiction as we have shown above. We conclude that, for every two states p and p' of $\mathbf{st}\,(A \diamond e)$, p' is reachable from p by a path in $A \diamond e$. Thus, $A \diamond e$ is strongly connected, q.e.d. □

Proposition 6 *For behavior automaton A and knot G in A, there exists a sequence e in* $\mathbf{lim\,tr}\,A$ *such that* $A \diamond e = G$.

Proof If G has no edge, then, since G is strongly connected, G has only one state. Let p denote that state. Since G is reachable, there exists a path from $\mathbf{init}A$ to p. Let u be the word spelled by that path; we have $A \diamond u = p$. Let e be the finite execution spelling u. We have $\mathbf{st}\,(A \diamond e) = \{p\}$ and $\mathbf{ed}\,(A \diamond e) = \emptyset$, thus $A \diamond e = G$.

Now, we consider the case where G has at least one edge. We know that G can have only finitely many edges, thus we can ennumerate them as follows:

$$(p^1, a^1, q^1), \ldots, (p^n, a^n, q^n)$$

where $n > 0$.

Since G is reachable, there exists a path from $\mathbf{init}A$ to p^1. Let u^{01} be the word spelled by that path. We have $A \diamond u^{01} = p^1$.

Since G is strongly connected, for every i and j in $\{1, \ldots, n\}$, such that $j = i \bmod n + 1$, there exists a path π^{ij} in G from q^i to p^j. Let u^{ij} be the word spelled by that path.

We now concatenate the words spelled by these paths and the symbols on these edges to form an infinite sequence. Let $e = u^{01}(a^1 u^{12} \cdots a^n u^{n1})^\omega$. Note that, since $n > 0$, the string $a^1 u^{12} \cdots a^n u^{n1}$ is non-void and, thus, e is infinite.

One verifies that, for all $k \geq 0$ and l such that $0 \leq l < n$,

$$A \diamond t^{kl} = p^{l+1}$$

where

$$t^{kl} = u^{01}(a^1 u^{12} \cdots a^n u^{n1})^k a^1 u^{12} \cdots a^l u^{l\,l+1}$$

(Note that $l + 1 = l \bmod n + 1$ because $l < n$, and that the string $a^1 u^{12} \cdots a^l u^{l\,l+1}$ is empty if $l = 0$.)

Therefore, for each l as above, state p^{l+1} is led-to by infinitely many prefixes t^{kl} of e. Similarly, since $t^{kl} a^{l+1} \leq e$, edge $(p^{l+1}, a^{l+1}, q^{l+1})$ is passed through by e infinitely many times. We conclude that $\mathbf{st}\, G \subseteq \mathbf{st}\, (A \diamond e)$ and $\mathbf{ed}\, G \subseteq \mathbf{ed}\, (A \diamond e)$.

Conversely, note that the paths π^{ij} are in G, thus they pass only through states and edges of G. In conclusion, we also have $\mathbf{st}\, G \supseteq \mathbf{st}\, (A \diamond e)$ and $\mathbf{ed}\, G \supseteq \mathbf{ed}\, (A \diamond e)$.

Therefore, $G = A \diamond e$, q.e.d. □

Lemma 5 *For behavior automata A and B and word t in $\mathbf{lg}\,(A \| B)$, we have*

$$t_A \in \mathbf{lg}\, A \ \wedge\ ((A \| B) \diamond t)_A = A \diamond t_A, \ \textit{and}$$
$$t_B \in \mathbf{lg}\, B \ \wedge\ ((A \| B) \diamond t)_B = B \diamond t_B.$$

Proof Since the two parts are similar, we prove only the first one. We use structural induction over t.

• Basis: $t = \varepsilon$. We have, trivially, that $\varepsilon_A = \varepsilon \in \mathbf{lg}\, A$, and thus $((A \| B) \diamond \varepsilon)_A = (\mathbf{init}(A \| B))_A = \mathbf{init}A = A \diamond \varepsilon = A \diamond \varepsilon_A$.

• Step: Let $t = ub$, where b is an action in $\mathbf{a}(A \| B)$. Assume the property holds for u. We consider two cases for b:

− Case $b \notin \mathbf{a}A$. By the inductive assumption, we have $\mathbf{lg}\, A \ni u_A = (ub)_A$. By the definition of compatible triples, we have $((A \| B) \diamond (ub))_A = ((A \| B) \diamond u)_A$. By the inductive assumption, $((A \| B) \diamond u)_A = A \diamond u_A$. Since $b \notin \mathbf{a}A$, it follows that $(ub)_A = u_A$, and $A \diamond u_A = A \diamond (ub)_A$. Thus, $((A \| B) \diamond (ub))_A = A \diamond (ub)_A$.

− Case $b \in \mathbf{a}A$.

By the definition of $\diamond$, there exists an edge in $A \| B$ of the form

$$(\ ((A \| B) \diamond u),\ b,\ ((A \| B) \diamond (ub))\ ).$$

Since $b \in \mathbf{a}A$, there exists an edge in A of the form $(q,\ b,\ q')$ where $q = ((A \| B) \diamond u)_A$ and $q' = ((A \| B) \diamond (ub))_A$.

Thus, there is a path from $\mathbf{init}A$ to q spelling u_A, to which we add the edge (q, b, q') to obtain a path from $\mathbf{init}A$ to q' spelling $(ub)_A$. Thus, $t_A \in \mathbf{lg}\, A$.
By the induction hypothesis, $q = A \diamond u_A$.
By the definition of $\diamond$ again, and because $b \in \mathbf{a}A$, we have $q' = A \diamond u_A b = A \diamond (ub)_A$.
In conclusion, $((A\|B) \diamond (ub))_A = q' = A \diamond (ub)_A$ q.e.d. □

Lemma 6 *For behavior automata A and B and word t in $(\mathbf{a}(A\|B))^*$, we have*

$$t \in \mathbf{lg}\, A\|B \;\Leftrightarrow\; t_A \in \mathbf{lg}\, A \;\wedge\; t_B \in \mathbf{lg}\, B.$$

Proof
(⇒) This part is an immediate consequence of Lemma 5.
(⇐) We use structural induction over t.
– Basis: $t = \varepsilon$. We have $\varepsilon \in \mathbf{lg}\,(A\|B)$.
– Step: Let $t = uc$, where $c \in \mathbf{a}A \cup \mathbf{a}B$ and $u \in \mathbf{lg}\,(A\|B)$. Let $p = A \diamond u_A$, $p' = A \diamond t_A$, $q = B \diamond u_B$, and $q' = B \diamond u_B$.

We have that $(p, c, p') \in \mathbf{ed}\, A \;\vee\; c \notin \mathbf{a}A$ and $(q, c, q') \in \mathbf{ed}\, B \;\vee\; c \notin \mathbf{a}B$, thus $((p, q), c, (p', q')) \in \mathbf{ed}\,(A\|B)$.

By Lemma 5, $(A\|B) \diamond u = (p, q)$. Thus, by the definition of the $\diamond$ extension, there exists a path in $A\|B$ from $\mathbf{init}(A\|B)$ to (p, q) spelling u. By appending $((p, q), c, (p', q'))$ to that path, we obtain a path starting at $\mathbf{init}(A\|B)$ and spelling ua. Thus, $t \in \mathbf{lg}\,(A\|B)$, q.e.d. □

Theorem 5 *For behavior automata A and B, we have* $\mathbf{tr}\,(A\|B) = \mathbf{tr}\, A \parallel \mathbf{tr}\, B$.

Proof Immediate using Lemma 6. □

Proposition 7 is proven immediately after the following lemma.

Lemma 7 *For behavior automata A and B and sequence e in* $\mathbf{lim}\,\mathbf{tr}\,(A\|B)$, $((A\|B) \diamond e)_A = A \diamond e_A$ *and* $((A\|B) \diamond e)_B = B \diamond e_B$.

Proof Since finite sequences lead to single-state no-edge subgraphs, the property for a finite sequence e trivially follows from Lemma 5.

For infinite sequences, we only prove the A-part. The B-part is similar to the A-part. We show: **(a)** $\mathbf{st}\,((A\|B) \diamond e)_A \subseteq \mathbf{st}\,(A \diamond e_A)$, **(b)** $\mathbf{st}\,((A\|B) \diamond e)_A \supseteq \mathbf{st}\,(A \diamond e_A)$, **(c)** $\mathbf{ed}\,((A\|B) \diamond e)_A \subseteq \mathbf{ed}\,(A \diamond e_A)$, and **(d)** $\mathbf{ed}\,((A\|B) \diamond e)_A \supseteq \mathbf{ed}\,(A \diamond e_A)$.

(a) Let $p \in \mathbf{st}\,((A\|B) \diamond e)_A$.
By the definition of subgraph projections, $\exists\, o \in \mathbf{st}\,((A\|B) \diamond e)$ such that $o_A = p$.
By the definition of the $\diamond$ extension, $\forall\, t \leq e$, $\exists\, u$ such that $tu \leq e$ and $(A\|B) \diamond (tu) = o$.
By Lemma 5, for each such t and u, we have $o_A = A \diamond (tu)_A$ and thus $p = A \diamond (t_A u_A)$.
One verifies that, for each $t' \leq e_A$, there exists $t \leq e$ such that $t_A = t'$, and therefore there exists $u' = u_A$ (where u is constructed as above) such that $t'u' \leq e_A$ and $p = A \diamond (t'u')$.

We conclude that $p \in \mathbf{st}\,(A \diamond e_A)$, by the definition of the $\diamond$ extension.

(b) Let $p \in \mathbf{st}\,(A \diamond e_A)$. Let $t \leq e$ arbitrary.
By the definition of the $\diamond$ extension, $\exists\ u'$ such that $t_A u' \leq e_A$ and $A \diamond (t'u') = p$.
One verifies that, $\forall\ u'$ such that $t_A u' \leq e$, $\exists\ u$ such that ($tu \leq e$ and $u_A = u'$).
Thus, we have $\forall\ t \leq e,\ \exists\ u$ such that $tu \leq e\ \wedge\ A \diamond (tu)_A = p$.
Since $e \in \mathbf{lim\ tr}\,(A\|B)$, we have that $tu \in \mathbf{lg}\,(A\|B)$ and thus we can apply Lemma 5 to obtain $\forall\ t \leq e,\ \exists\ u$ such that $tu \leq e\ \wedge\ ((A\|B) \diamond (tu))_A = p$.
That is, since e is infinite, there are infinitely many prefixes (tu) of e such that $(A\|B) \diamond (tu)_A = p$.
Since there are only finitely many states o of $A\|B$ such that $o_A = p$, by the pigeonhole principle there exist one state o' of $A\|B$ such that there are infinitely many prefixes v of e which lead-to o':

$$\exists\ o' \in \mathbf{st}\,(A\|B) \text{ such that}$$
$$(o'_A = p\ \wedge\ \forall\, t \leq e,\ \exists\ u \text{ such that } tu \leq e\ \wedge\ (A\|B) \diamond (tu) = o')$$

Therefore, $\exists\ o' \in \mathbf{st}\,((A\|B) \diamond e)$ such that $o'_A = p$; thus, $p \in \mathbf{st}\,((A\|B) \diamond e)_A$.

(c) Part (c) is almost identical to Part (a) discussed above.
Let $(p, d, p') \in \mathbf{ed}\,((A\|B) \diamond e)_A$.
By the definition of subgraph projections, $\exists\ o \in \mathbf{st}\,((A\|B) \diamond e)$ such that $o_A = p$.
By the definition of the $\diamond$ extension, $\forall\ t \leq e,\ \exists\ u$ such that $tud \leq e$ and $(A\|B) \diamond (tu) = o$.
By Lemma 5, for each such t and u, we have $o_A = A \diamond (tu)_A$ and thus $p = A \diamond (t_A u_A)$.
One verifies that, for each $t' \leq e_A$, there exists $t \leq e$ such that $t_A = t'$, and therefore there exists $u' = u_A$ (where u is constructed as above) such that $t'u'd \leq e_A$ and $p = A \diamond (t'u')$.

We conclude that $(p, d, p') \in \mathbf{ed}\,(A \diamond e_A)$, by the definition of the $\diamond$ extension.

(d) Part (d) is almost identical to Part (b) discussed above.
Let $(p, d, p') \in \mathbf{ed}\,(A \diamond e_A)$. Let $t \leq e$ arbitrary.
By the definition of the $\diamond$ extension, $\exists\ u'$ such that $t_A u'd \leq e_A$ and $A \diamond (t'u') = p$.
One verifies that, $\forall\ u'$ such that $t_A u'd \leq e$, $\exists\ u$ such that ($tud \leq e$ and $u_A = u'$).
Thus, we have $\forall\ t \leq e,\ \exists\ u$ such that $tud \leq e\ \wedge\ A \diamond (tu)_A = p$.
Since $e \in \mathbf{lim\ tr}\,(A\|B)$, we have that $tu \in \mathbf{lg}\,(A\|B)$ and thus we can apply Lemma 5 to obtain $\forall\ t \leq e,\ \exists\ u$ such that $tu \leq e\ \wedge\ ((A\|B) \diamond (tu))_A = p$.
That is, since e is infinite, there are infinitely many prefixes (tu) of e such that $(A\|B) \diamond (tu)_A = p$ and $tud \leq e$.
Since there are only finitely many states o of $A\|B$ such that $o_A = p$, by the pigeonhole principle there exist one state o' of $A\|B$ such that there are infinitely many prefixes v of e which lead-to o' and such that $vd \leq e$:

$$\exists\ o' \in \mathbf{st}\,(A\|B) \text{ such that}$$
$$(o'_A = p\ \wedge\ \forall\, t \leq e,\ \exists\ u \text{ such that } tud \leq e\ \wedge\ (A\|B) \diamond (tu) = o')$$

Therefore, $\exists\ o' \in \mathbf{st}\,((A\|B) \diamond e)$ such that $o'_A = p$; thus, $(p, d, p') \in \mathbf{ed}\,((A\|B) \diamond e)_A$.
□

Proposition 7 *For behavior automata A and B and knot G in $A\|B$, G_A is a knot in A and G_B is a knot in B.*

Proof
By Proposition 6, there exists a sequence e in $\mathbf{lim}\,(A\|B)$ such that $(A\|B)\diamond e = G$.
By Lemma 7, we have that $A \diamond e_A = ((A\|B)\diamond e)_A$.
By Proposition 5, we conclude that G_A is a knot in A.
One proves similarly that G_B is a knot in B. □

Lemma 8 *For behavior automaton A and sequence e in A, we have* $\mathbf{rfi}\, e = \mathbf{fi}\,(A\diamond e)$ *and* $\mathbf{ren}_{\mathbf{lg}\,A} e = \mathbf{en}_A(A \diamond e)$.

Proof We distinguish two cases.
• Case e finite.
– The first equality is trivial: $\mathbf{rfi}\, e = \emptyset = \mathbf{fi}\,(A \diamond e)$, since $(A \diamond e)$ has no edges.
– For the second equality, we note:

$$\begin{aligned}
&\mathbf{en}_A(A \diamond e)\\
&= \{a \in \mathbf{a}A \mid \exists\, p' \in \mathbf{st}\, A \text{ such that } ((A \diamond e), a, p') \in \mathbf{ed}\, A\}\\
&= \{a \in \mathbf{a}A \mid ea \in \mathbf{lg}\, A\}\\
&= \{a \in \mathbf{a}A \mid \forall\, t \leq e\ \exists\, u \text{ such that } tu \leq e \ \wedge\ tua \in \mathbf{lg}\, A\} \qquad (e \text{ is finite})\\
&= \mathbf{en}_{\mathbf{lg}\,A} e
\end{aligned}$$

• Case e infinite.
– We prove $\mathbf{rfi}\, e \ \subseteq\ \mathbf{fi}\,(A \diamond e)$. Let $a \in \mathbf{rfi}\, e$, i.e., e fires a infinitely many times. Since there are only finitely many edges labelled with symbol a in A, by the pigeonhole principle, at least one such edge is passed through infinitely often by e. That edge is thus an edge in $(A \diamond e)$. Since that edge has symbol a, we conclude $a \in \mathbf{en}_A(A \diamond e)$.
– We prove $\mathbf{fi}\,(A \diamond e) \ \subseteq\ \mathbf{rfi}\, e$. Let $a \in \mathbf{fi}\,(A \diamond e)$, i.e., such that there exists an edge in $A \diamond e$ labelled with symbol a. By the definition of $A \diamond e$, that edge is passed through infinitely often by e, thus a is fired infinitely often by e.
– We prove $\mathbf{ren}_{\mathbf{lg}\,A} e \ \subseteq\ \mathbf{en}_A(A \diamond e)$. Let $a \in \mathbf{ren}_{\mathbf{lg}\,A} e$, i.e., a is immediately enabled in $\mathbf{lg}\, A$ by infinitely many prefixes of e. Let $S = \{A \diamond t \mid t \leq e \ \wedge\ ta \in \mathbf{lg}\, A\}$, i.e., the set of states led-to by these prefixes; we have $\forall\, p \in S,\ \mathbf{en}_A p \ni a$. Since there are only finitely many states in A, one of the states in S, by the pigeonhole principle, is led-to by infinitely many prefixes of e, i.e., $\exists\, p \in S$ such that $\forall\, t \leq e,\ \exists\, u$ such that $tu \leq e \ \wedge\ A \diamond (tu) = p$. Thus, $p \in \mathbf{st}\,(A \diamond e)$. Since $\mathbf{en}_A p \ni a$, we conclude that $a \in \mathbf{en}_A(A \diamond e)$.
– We prove $\mathbf{en}_A(A \diamond e) \ \subseteq\ \mathbf{ren}_{\mathbf{lg}\,A} e$. Let $a \in \mathbf{en}_A(A \diamond e)$, i.e., such that there exists a state p in $A \diamond e$ such that $a \in \mathbf{en}_A p$. We have that p is led-to by infinitely many prefixes of e, i.e., $\forall\, t \leq e,\ \exists\, u$ such that $tu \leq e \ \wedge\ A \diamond (tu) = p$. For each such prefix tu, we have that $tua \in \mathbf{lg}\, A$, because $a \in \mathbf{en}_A p$. Therefore, a is immediately enabled by infinitely many prefixes of e, and thus $a \in \mathbf{ren}_{\mathbf{lg}\,A} e$. □

Lemma 9 *For behavior automaton A, knot G in A, and sequence e in* $\mathbf{lim}\,\mathbf{tr}\, A$ *such that $A \diamond e = G$, we have that G is an output trap in A iff e is an output trap for* $\mathbf{tr}\, A$.

Proof By Lemma 8, $(\mathbf{en}_A G \cap \mathbf{o}A = \mathbf{fi}\, G) \Leftrightarrow ((\mathbf{ren}_{\mathbf{lg}\, A} e) \cap \mathbf{o}A = \mathbf{rfi}\, e)$. By the definition of $\mathbf{tr}\, A$, $((\mathbf{ren}_{\mathbf{lg}\, A} e) \cap \mathbf{o}A = \mathbf{rfi}\, e) \Leftrightarrow ((\mathbf{ren}_{\mathbf{lg\ tr}\, A}) \cap \mathbf{o\,tr}\, A = \mathbf{rfi}\, e)$. Finally, $((\mathbf{ren}_{\mathbf{lg\ tr}\, A}) \cap \mathbf{o\,tr}\, A = \mathbf{rfi}\, e) \Leftrightarrow (e \in \mathbf{otp\ tr}\, A)$. □

Theorem 6 *For behavior automata S and I, I is traplock-free for S iff* $\{\mathbf{tr}\, S\} \sqsubseteq_\lambda \{\mathbf{tr}\, I\}$.

Proof
(⇒) We prove that $\{\mathbf{tr}\, S\} \sqsubseteq_\lambda \{\mathbf{tr}\, I\}$. Let $e \in \mathbf{lim}\, (\mathbf{tr}\, S \| \mathbf{tr}\, I)$ such that $e_I \in \mathbf{otp\ tr}\, I$. By Theorem 5, $e \in \mathbf{lim\ tr}\, (S \| I)$. Let $G = (S \| I) \diamond e$; by Proposition 5, G is a knot in $S \| I$. By Proposition 7, G_I is a knot in I and G_S is a knot in S. By Lemma 7, $G_I = I \diamond e_I$ and $G_S = S \diamond e_S$. By Lemma 9, G_I is a trap in I. Since I is traplock-free for S, G_S is a trap in S. By Lemma 9 again, we conclude that $e_S \in \mathbf{otp\ tr}\, S$.
(⇐) We prove that I is traplock-free for S. Let G be a knot in $S \| I$ such that G_I is an output trap in I. By Proposition 6 and Theorem 5, there exists a sequence e in $\mathbf{lim\ tr}\, (S \| I) = \mathbf{lim}\, (\mathbf{tr}\, S \| \mathbf{tr}\, I)$ such that $(S \| I) \diamond e = G$. By Lemma 7 and Lemma 9, $e_I \in \mathbf{otp\ tr}\, I$. Therefore, $e_S \in \mathbf{otp\ tr}\, S$ and, by Lemma 9 and Proposition 5, we conclude that G_S is an output trap in S. □

References

[AS85] B. Alpern, F. B. Schneider. Defining Liveness. *Information Processing Letters*, 21(4), 1985, pp. 181–185.

[Bl86] D. Black. On the Existence of Delay-insensitive Fair Arbiters: Trace theory and its limitations. *Distributed Computing*, (1), 1986, pp. 205–225.

[BS95] J. A. Brzozowski, C-J. H. Seger. *Asynchronous Circuits.* Springer Verlag, 1995.

[CMP92] E. Chang, Z. Manna, A. Pnueli. The Safety-Progress Classification. Report No. STAN-CS-92-1408, Stanford University, Dept. of Computer Science, 1992.

[DC85] D. Dill, E. Clarke. Automatic Verification of Asynchronous Circuits Using Temporal Logic. In: H. Fuchs, ed., *1985 Chapel Hill Conf. on VLSI*, pp. 127–143. Computer Science Press, 1985.

[Di89] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* An ACM Distinguished Dissertation, MIT Press, 1989.

[Eb86] J. C. Ebergen. A Technique to Design Delay-Insensitive VLSI Circuits. Report CS-R8622, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1986.

[Eb91] J. C. Ebergen. A Formal Approach to Designing Delay-Insensitive Circuits. *Distributed Computing*, (5), 1991, pp. 107–119.

[Fr86] N. Francez. *Fairness.* Springer-Verlag, 1986.

[GBMN94] G. Gopalakrishnan, E. Brunvand, N. Mitchell, S. M. Nowick. A Correctness Criterion for Asynchronous Circuit Validation and Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13 (11), 1994, pp. 1309-1318.

[Ho85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Jo87] B. Jonson. Modular Verification of Asynchronous Networks. In: *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing*, pp. 137–151, 1987.

[Jos92] M. B. Josephs. Receptive Process Theory. *Acta Informatica*, 29(1):17-31, 1992.

[LL90] L. Lamport, N. Lynch. Distributed Computing: Models and Methods. In J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, vol. B, Formal Methods and Semantics*, the MIT Press - Elsevier, pp. 1159–1196, 1990.

[LT87] N. Lynch, M. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In: *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing*, 137–151, 1987.

[Mi89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[NB95] R. Negulescu and J. A. Brzozowski. Relative Liveness: From Intuition to Automated Verification. In *Proceedings of the Second Working Conference on Asynchronous Design Methodologies*, South Bank University, London, UK, IEEE Computer Society Press, pp. 108-117, May 1995.

[RSU83] M. Rem, J. L. A. van de Snepscheut, J. T. Udding. Trace Theory and the Definition of Hierarchical Components. In R. Bryant, ed., *Third CalTech Conference on Very Large Scale Integration*, pp. 225–239, Computer Science Press, Inc., 1983.

[Sn83] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design*. PhD Thesis, Department of Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1983.

[Th90] W. Thomas. Automata on Infinite Objects. In J. van Leeuwen, *Handbook of Theoretical Computer Science, vol. B, Formal Methods and Semantics*, the MIT Press - Elsevier, pp. 135–191, 1990.

[Ud86] J. T. Udding. A Formal Model for Defining and Classifying Delay-Insensitive Circuits and Systems. *Distributed Computing*, 1(4):197–204, 1986.

[Ud84] J. T. Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD Thesis, Department of Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1984.

[Ve94] T. Verhoeff. *A Theory of Delay-Insensitive Systems*, Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1994.

[YLS92] A. Yakovlev, L Lavagno, A. Sangiovanni-Vincentelli. A Unified Signal Transition Graph Model for Asynchronous Control Circuit Synthesis. In *Proc. of the IEEE Int. Conf. on Computer Aided Design*, pp. 104-111, IEEE Computer Society Press, 1992.