Formal Methods in System Design, 12, 73–115 (1998)

Relative Liveness: From Intuition to Automated Verification

R. NEGULESCU radu@maveric.uwaterloo.ca

J. A. BRZOZOWSKI brzozo@maveric.uwaterloo.ca

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

Received September 1, 1995; Revised February 18, 1997

Editor: E.M. Clarke

Abstract. We define a new liveness condition for asynchronous circuits. Although finitary (finite-execution) descriptions are not powerful enough to express general liveness properties, those liveness properties needed in practice appear to be related in a unique manner to finitary descriptions. Our liveness condition exploits this observation and is defined directly on finitary descriptions, in two forms: one on finite trace structures and the other on finite automata. We prove the equivalence of these two forms. We also introduce a safety condition and derive theorems for the modular and hierarchical verification theorems of both safety and liveness. Finally, we give an algorithm for verifying our liveness condition.

Keywords: asynchronous circuits, deadlock, fairness, finite automata, liveness, safety, trace structures, verification.

1. Introduction

Asynchronous circuits operate without the constraint of a global clock. Removing this constraint holds promise for achieving high speed and power economy. However, asynchronous circuits lag behind their synchronous counterparts in terms of techniques and tools for analysis and debugging. Part of the problem is that asynchronous circuits are exposed to very diverse dangers of failure, such as deadlock, livelock, unfairness, similar to the problems in concurrency theory. In fact, asynchronous circuits can be treated as concurrent systems. The parts of an asynchronous circuit behave like discrete-state processes synchronized on common events, where the events correspond to signal transitions in the circuit.

Two correctness concerns for asynchronous circuits and other concurrent systems are safety and liveness. Intuitively, safety properties assert that "something bad does not happen" and liveness properties assert that "something good eventually does happen" [14]. We consider unfairness and deadlock to be liveness violations, as done in [6], because both deadlock and unfairness can be regarded, intuitively, as cases where a required output action does not eventually happen.

In this paper we define, analyze and implement a correctness condition for verifying liveness in asynchronous circuits.

A major obstacle to verifying liveness is the amount of information needed by the verification procedure. In our view, a correctness condition should be expressible in terms of common descriptions of asynchronous circuits, such as Petri nets, signal transition graphs, or digital circuit schematics together with, say, logic relationships between the inputs and outputs of each component. The difficulty is that such descriptions specify only the finite

executions (sequences of events) of the circuits to verify, while two circuits with different liveness properties can have the same finite executions. For this reason, finite-execution models, such as finite trace theory, have been considered ambiguous for defining liveness (e.g., see [2]).

In this situation it may seem natural to extend the model with some representation of infinite executions, and to ask the users to specify the liveness properties of their systems explicitly. Such *user-directed* approaches (e.g. [6, 12]) 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 asynchronous circuits (i.e., finitary representations like those we listed above). Users have to specify explicitly the liveness properties or the infinite 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 [4].)

To illustrate how liveness properties are easy to overlook and hard to specify *explicitly*, let us consider a gate specified by a Boolean function. In the absence of timing concerns, it is easy to forget to specify that a gate should not deadlock internally; such an error allows for an asynchronous circuit to do nothing in response to changes in inputs. Also, consider a mutual exclusion element which ensures that two processors do not access the same resource at the same time. Trivial implementations are an arbiter that serves none of the processes [4] and an arbiter that always favors one of the processes.

Here, we propose a different approach to resolve the ambiguity of finite-execution models. In practice, liveness properties seem to be *implicitly assumed*, like deadlock-freedom and fairness in the examples above: we don't normally use Boolean gates that can deadlock or arbiters that can be unfair. Accordingly, we attach unique liveness properties to a finite-execution description. For instance, we will always impose in our model that the Boolean gate and the arbiter in the examples above should be deadlock-free and fair, respectively; failure to meet these criteria will be detected as a violation of our correctness condition. This way, the users only need to provide the finite-execution descriptions; we extract the infinite executions and the liveness properties from such descriptions. We attach unique liveness *constraints* to finite-execution implementations and unique liveness *requirements* to finite-execution specifications, and we check that the constraints suffice to guarantee the requirements. This way we obtain a *relative* liveness condition whose verdict is uniquely determined by finite-execution descriptions of the implementation and the specification. This condition can detect deadlock and unfairness in common models of asynchronous circuits.

To show that our liveness condition is suitable for modular and hierarchical verification, we prove several algebraic properties of the condition. Unfortunately, we need to impose several restrictions on the systems for which these algebraic properties are satisfied. Some of these restrictions are necessary correctness conditions themselves, mainly regarding safety. Hence, we define a safety condition and we show it has sufficient algebraic properties for modular and hierarchical verification as well. Our safety condition is closely related to "absence of computation interference" [7, 20, 23, 25], our contribution being mainly to extend that condition to get rid of connectivity restrictions.

In addition to a language-theoretic form for the liveness condition, we also give an automaton-theoretic form. We show the two forms are equivalent, and we give an algorithm for verifying our liveness condition. We show the algorithm is correct and we evaluate its complexity.

In [6] and [12], general frameworks for reasoning about liveness have been proposed, along with thorough algebraic treatments. However, those approaches are user-directed as described above: they start their verification procedures from explicit descriptions of liveness properties or infinite executions. In [1], an exhaustive characterization of liveness properties has been proposed. However, nothing is said about which of the properties in the class defined by [1] can be used for a liveness condition. The liveness condition in [15] also provides important insights. Unfortunately, that condition fails to detect certain cases of unfairness (see [18], p. 30). An interesting trace-based model of progress properties with a careful algebraic treatment has been proposed in [25]. However, of the liveness faults, [25] detects only situations where every process is blocked.

CCS [16] and CSP [11] can model high-level descriptions of asynchronous circuits in a convenient and elegant manner. However, [15] lists several problems with these formalisms when they are used to define liveness, and [6] points out difficulties in using these formalisms for modeling low-level communication.

We use double quotes "" for citations, and single quotes '' for some informal or undefined terms. Proofs of some of our results are provided in the appendices.

2. Informal Preview

In this section we discuss informally some of our motivations and results. This informal preview is followed by a formal treatment in the body of the paper.

Our expectations for a liveness condition

A liveness condition should be able to detect deadlock and unfairness, should be satisfied by correct circuits, and should satisfy certain algebraic properties. Ideally, it should also be satisfied by circuits that may be incorrect for other reasons, but have no 'deadlock-like' or 'unfairness-like' faults. A *relative* correctness condition compares an implementation to a specification, as opposed to an *absolute* correctness condition, which examines an implementation by itself.

We use automata to illustrate our correctness conditions. In figures, automata have an initial state marked with an incoming arrow, and edges marked with actions. Inputs have a ? sign and outputs have a ! sign. Inputs or outputs that are never fired and do not appear on

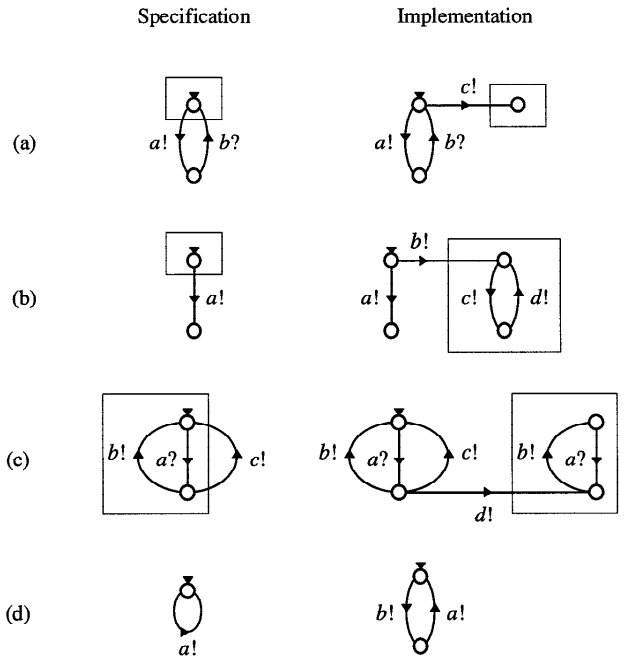

Figure 1. Traplock-freedom.

the edges are listed below the automaton, with the annotation 'unused'. The box-shaped 'frames' appearing in some of the figures are just markers.

Our liveness condition: 'traplock-freedom'

For the language form of traplock-freedom, we start from trace structure descriptions of the specification and the implementation. A trace structure contains an input alphabet, an output alphabet, and a language of finite words over the union of the alphabets. Trace structures can be represented by automata like those in Figure 1. From the language of a trace structure we compute the limit sequences, which are finite or infinite sequences that have all prefixes in the language. For example, in Figure 1 (a), on the right side, abc and $abab...$ are limit sequences. Not all of the limits are legal for a system. We call a limit an *output trap* if it *fires infinitely many times any outputs that are enabled for an infinite amount of time*. Output symbols that are enabled infinitely many times or are enabled at the end of a finite sequence can be viewed intuitively as building up a pressure to be fired, so they should be fired from time to time to release that pressure. For example, limit $abab...$ in the trace structure mentioned above is not an output trap, since c is enabled infinitely many times but never fired. In the same trace structure, limit ab is not an output trap, since it enables a and c at the end, while limits abc and a are output traps, since they do not

enable output symbols at the end. Then, *traplock-freedom forbids that an output trap of the implementation correspond to a limit of the specification which is not an output trap*, or else the implementation fails to fire some output required by the specification. We give some examples below and we formalize these ideas in later sections. The main difference from other treatments of liveness is that we extract our legal limits (the output traps) from finitary descriptions (the trace structures), instead of asking the users to specify them explicitly in addition of a finitary description.

Traplock-freedom is also defined directly in an automaton form. Basically, *traplock-freedom forbids that the implementation get stuck in some set of states and edges while the specification demands an output action to be eventually issued*. A formalization is given in later sections, and the automaton form will be shown equivalent to the language form.

Examples

- In Figure 1 (a), the specification ignores $c!$ because c is not in its alphabet. As a result, the implementation may get stuck in the framed state while the specification demands an $a!$. From the language viewpoint, c is an output trap of the implementation corresponding in the specification to ϵ, the empty word, which is not an output trap of the specification. Lack of traplock-freedom shows a danger of deadlock because then the implementation will never produce an output, while the specification expects an $a!$.

- Figure 1 (b) shows a fault which combines deadlock with divergence. Again, the specification ignores $b!$, $c!$, and $d!$. The implementation can enter and remain in the framed subgraph, without ever producing $a!$, while the specification waits for $a!$. Also, $bcdcd...$ is an output trap for the implementation, corresponding to ϵ, which is not an output trap for the specification. From the specification point of view, this fault is a deadlock-like fault, no different from that in Figure 1 (a).

- Figure 1 (c) illustrates unfairness. The implementation can stay in the framed graph, while the specification moves just within its framed subgraph. Also, $adbaba...$ is an output trap for the implementation, corresponding to $abab...$, which not an output trap of the specification. A $c!$ is thus demanded by the specification, but never comes.

- In Figure 1 (d), an $a!$ is demanded by the specification in the initial state, but the implementation cannot produce it right away; it has to produce a $b!$ first. Still, the implementation is fine, since $a!$ will eventually come as requested.

In a user-directed approach, the information in Figure 1 would be considered insufficient because no liveness properties are explicitly specified. It is conceivable that the specification in Figure 1 (a) allows for termination in the initial state, in which case no deadlock would occur. It is also conceivable that the specification in Figure 1 (c) allows for the infinite execution $ababab\ldots$, in which case that implementation would be fine. To avoid such ambiguities, one can either use more detailed descriptions, or interpret the descriptions at hand in a more precise manner. We take the second approach, by augmenting a finitary description with standard liveness properties.

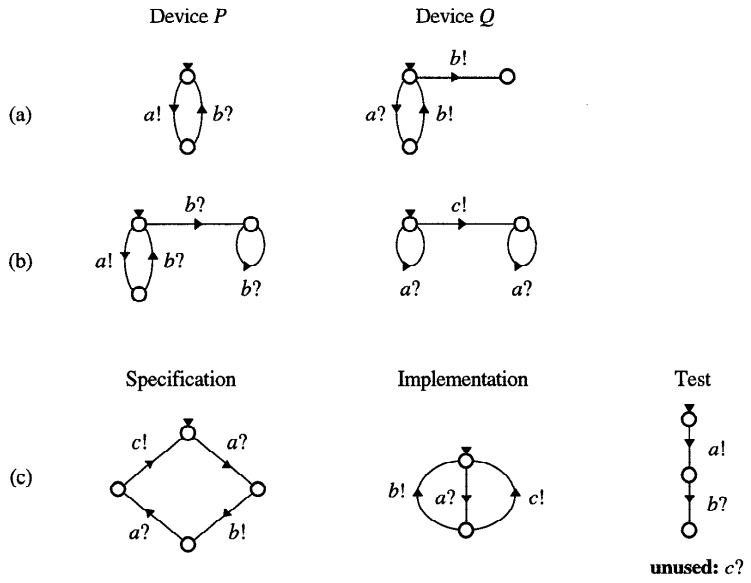

Figure 2. Safety issues.

Algebraic properties of traplock-freedom

Under certain restrictions, our liveness relationship is reflexive, transitive, and compatible with coupling of networks. Transitivity means that we can perform verification hierarchically, comparing only pairs of consecutive intermediate specifications. Compatibility with coupling means that we can perform verification modularly, replacing parts by more detailed implementations, one part at a time, and comparing only the parts with their specifications. Such structured verification can speed up significantly the overall verification procedure.

Unfortunately, the algebraic properties mentioned above hold only under certain restrictions, having to do mainly with safety. However, to maximize practical applicability, we have tried to use only safety restrictions that are themselves conditions for the correct functioning of a system. We use an absolute condition and a relative condition for safety. (The relative condition is actually derived from the absolute condition.)

Safety

Our absolute condition for safety basically demands that, *whenever an event can be generated by all processes that have that event as an output, that event can also be accepted by all processes that have it as an input.* Figure 2 illustrates this condition.

- In Part (a), Q can generate $b!$ in the initial state, while P cannot accept it. Hence the network is not safe.

- In Part (b), input $b?$ of P is dangling. In most cases this would lead to safety violations because dangling inputs can generate transitions at any time (in circuits, dangling inputs behave like antennae). In this case, however, there is no safety violation because of the dangling $b?$, since P can accept $b?$ at any time, and $b?$ is the only input of P.

Our relative safety condition, comparing an implementation to a specification, is obtained by testing the implementation and the specification in various environments, with respect to safety. Basically, *the implementation should 'pass' at least those tests that the specification passes, where the tests consist of checking the absolute safety condition* in the respective environments.

- In Figure 2 (c), the automaton on the left is the specification, that on the middle is the implementation, and that on the right is a discriminating test. The implementation may produce a safety violation for the test, if the test issues an a event and the implementation responds with a c event. However, the network of the specification and the test has no safety violations, since the test stops after two events and then the specification just waits for an input event without issuing any b or c events. Thus, this implementation is not 'good' for this specification, with respect to our safety condition.

Although safety was well understood before, e.g. in [6, 7, 20, 23, 25], previous conditions have restrictions (either explicit or built into the model) on the ports of the processes they can compare or connect, and on the theorems for structured verification. With our safety conditions, we have eliminated all such restrictions. Our absolute safety condition agrees with absence of computation interference [7, 20] if there are no dangling inputs and no connected outputs, but disagrees otherwise. We provide more details in Section 4.

3. Trace Structures

We now develop the formal model in which we will define our liveness condition.

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, $aba \leq abaa$.

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 $\mathbf{lg}P$ be 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., [8]) 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.

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.

Networks and 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 $t_P = t_{\{P\}}$.

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. One verifies that their composite is $\langle\{a,b\},\,\{c,d\},\,\mathbf{pref}((a\,\cup\,b)\,(c\,(da\,\cup\,db\,\cup\,ad\,\cup\,bd))^*)\rangle$. (The language of the composite was computed as a language intersection as we indicate below. Nevertheless, one can verify directly that the language of the composite satisfies the definition given above.) 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.

The composite of a network is well-defined: the input and output alphabets of the composite are disjoint, and the language of the composite is prefix-closed and non-empty.

For trace structure P, we also use the following notation:

$$\mathbf{Lg}P \;=\; \{t \in \mathcal{U}^* \mid t_P \in \mathbf{lg}P\}\ .$$

Informally speaking, we indicate by a capital letter that we arbitrarily interleave outside symbols in the words of a particular language. For example, if $\mathbf{lg}P = \mathbf{pref}((ab)^*)$, $\mathbf{a}P = \{a,b\}$ and $\mathcal{U} = \{a,b,c\}$, then $\mathbf{Lg}P = \mathbf{pref}((c^*ac^*b)^*)$.

One verifies that, for every trace structure P, the language $\mathbf{Lg}P$ is prefix-closed and non-empty. It follows from the definition of $\|N$ that

$$\begin{array}{lcl} \mathbf{a}\|N & = & \bigcup_{P\in N} \mathbf{a}P\ , \text{ and} \\ \mathbf{Lg}\|N & = & \bigcap_{P\in N} \mathbf{Lg}P\ . \end{array}$$

For trace structures P and Q, we also use the shorthand $P\|Q$ for the composite of the network $\{P,Q\}$. We call this binary $\|$ operation on trace structures *parallel composition*. It follows from the definition of the composite of a network that parallel composition is idempotent, associative, and commutative.

Networks are our model of asynchronous circuits: each part of the circuit is represented by a trace structure and the whole circuit is represented by the set of trace structures of its parts. Thus, connecting two circuits corresponds to the union of networks. Note that a network is a set of trace structures while a composite is a single trace structure, which represents some but not all aspects of the network; two different networks may have the same composite. The same holds for the binary parallel composition operation: it yields a single trace structure which represents some aspects of a two-element network formed by the operands.

Input and output breaks

The words outside the language of the composite of a network are classified as input breaks or output breaks. Informally speaking, input breaks and output breaks are illegal executions, as follows. In an output break, an element of the network produces an illegal output. In an

input break, an element of the network receives an illegal input that is not an illegal output for any other element of the network. It is the network's responsibility to avoid the occurrence of output breaks, and it is the environment's responsibility to avoid the occurrence of input breaks.

Definition 1. For network N, the *set of output breaks* and the *set of input breaks* are, respectively,

$$\begin{aligned} \mathbf{ob}N &= \{uav \in (\mathbf{a}\|N)^* \mid u \in \mathbf{lg}\|N \wedge \exists Q \in N : a \in \mathbf{o}Q \wedge u_Q a \notin \mathbf{lg}Q\} , \\ \mathbf{ib}N &= \{uav \in (\mathbf{a}\|N)^* \mid u \in \mathbf{lg}\|N \wedge \exists Q \in N : a \in \mathbf{i}Q \wedge u_Q a \notin \mathbf{lg}Q\} \\ &\quad - \mathbf{ob}N . \end{aligned}$$

Note that all output breaks have been excluded by definition from the set of input breaks. For example, consider network

$$N = \{\langle\{a\}, \{b\}, \mathbf{pref}(ab)^*\rangle, \langle\{b\}, \{c\}, \mathbf{pref}(bc)^*\rangle\} .$$

We have $\mathbf{ob}N = (abc)^* \cdot \{b, c, ac, abb\} \cdot \{a, b, c\}^*$, and $\mathbf{ib}N = (abc)^* \cdot \{aa, aba\} \cdot \{a, b, c\}^*$. Notice that abb is not an input break, although it ends with an illegal input for the second element of N; the reason is that the second b is also an illegal output for the first element of N, and thus abb is an output break for N.

The following proposition gives the relationships among the language of the composite, the set of output breaks, and the set of input breaks for a network.

PROPOSITION 1 *For network N,*

(a) $\mathbf{ob}N \cap \mathbf{ib}N = \mathbf{ob}N \cap \mathbf{lg}\|N = \mathbf{ib}N \cap \mathbf{lg}\|N = \emptyset$ *, and*
(b) $\mathbf{ob}N \cup \mathbf{ib}N \cup \mathbf{lg}\|N = (\mathbf{a}\|N)^*$.

For network N, we also use the following notation:

$$\begin{aligned} \mathbf{Ib}N &= \{t \in \mathcal{U}^* \mid t_N \in \mathbf{ib}N\} , \\ \mathbf{Ob}N &= \{t \in \mathcal{U}^* \mid t_N \in \mathbf{ob}N\} . \end{aligned}$$

As we did for **Lg**, by using a capital letter we indicate that symbols from outside the alphabet of the network are interleaved arbitrarily in the input and output breaks.

4. Safety

Safety has been extensively studied in trace theory. Conditions covering safety concerns have been proposed, for example, in [6, 7, 8, 10, 13, 20, 23, 25]. 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

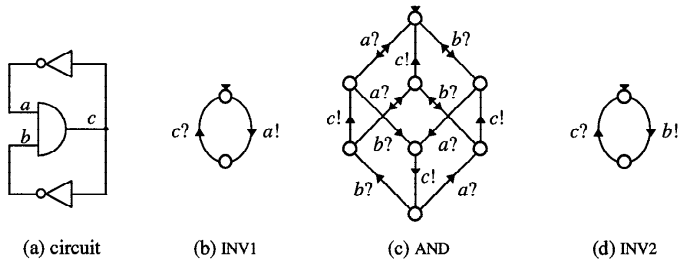

Figure 3. Hazard detection using the safety condition.

theorems. The fact that connectivity restrictions are not needed for the treatment of safety was surprising, particularly in the theorems for modular and hierarchical verification (see below).

Our absolute and relative safety conditions are defined as follows.

Definition 2. Network N is *safe* if for all words t in $\mathcal{U}^*$ that satisfy

$$\forall P \in N : \; t_P \in \mathbf{lg}P \cdot (\mathbf{i}P \cup \{\epsilon\})$$

we have

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

Network I *refines* network S *with respect to safety tests on networks*, written $S \sqsubseteq_{stn} I$ if, for any network N, if $N \cup S$ is safe then $N \cup I$ is also safe.

For an intuitive explanation, we refer to the 'such that' part in the definition above 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 always holds because $u_P \in \mathbf{lg}P$ and $a \in \mathbf{i}P$. For any P unrelated to a, the precondition also holds 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 be allowed to happen after u by all its sinks P.

The safety condition can be used to verify, for instance, absence of static hazards in a circuit. Static hazards are situations where a signal transition is enabled and then disabled without being performed.

Let INV1, AND, and INV2 be the three gates in Figure 3 (a), from top to bottom. The trace structures of the three gates are represented by the automata in Figure 3 (b), (c), and (d), respectively. (The double arrows in Figure 3 (c) stand for pairs of edges labelled by the same action and heading in opposite directions.) Hazards are forbidden by omitting certain input edges in these automata. For instance, word aba is not in the language of AND because the second a would disable an output transition, and, accordingly, word ab leads to a state from which no a event is allowed. To see whether the circuit is hazard-free, we check whether the network of INV1, AND, and INV2 is safe. Consider word $t = abcac$. We have that $t_{\mathrm{INV1}} = acac \in \mathbf{lg}\mathrm{INV1}$, $t_{\mathrm{AND}} = abcac \in \mathbf{lg}\mathrm{AND}$, and $t_{\mathrm{INV2}} = bcc \in \mathbf{lg}\mathrm{INV2} \cdot \mathbf{i}\mathrm{INV2}$. Thus, $\forall P \in \{\mathrm{INV1}, \mathrm{AND}, \mathrm{INV2}\}$, we have $t_P \in \mathbf{lg}P \cdot (\mathbf{i}P \cup \epsilon)$. However, $t_{\mathrm{INV2}} = bcc \notin \mathbf{lg}\mathrm{INV2}$, thus $t_{\{\mathrm{INV1},\mathrm{AND},\mathrm{INV2}\}} \notin \mathbf{lg}\|\{\mathrm{INV1}, \mathrm{AND}, \mathrm{INV2}\}$. Therefore the network $\{\mathrm{INV1}, \mathrm{AND}, \mathrm{INV2}\}$ is not safe.

The violation can be interpreted as follows. The offending word represents a danger of a hazard. After $abca$, both the input and the output signals of INV2 are high, and an output event is enabled. However, another input transition c can occur first, changing the input voltage to low and disabling the output transition. Also see the "oscor" example in [21], p. 167.

Let us check the safety condition on the examples in Figure 2. In Figure 2 (a), word b satisfies the precondition, since $b \in \mathbf{lg}Q$ and $\epsilon \in \mathbf{lg}P$ and $b \in \mathbf{i}P$. However, b does not satisfy the postcondition, since $b \notin \mathbf{lg}P$. Thus the network is not safe. For Figure 2 (b), we note that any word t that satisfies the precondition must satisfy $t \downarrow \{a, b\} \in (\mathbf{pref}((ab)^*bb^*)) \cdot (\epsilon \cup b)$ because of P, and $t \downarrow \{a, c\} \in (\mathbf{pref}(a^*ca^*)) \cdot (\epsilon \cup a)$ because of Q. By inserting c^* and b^* into the words of these languages, we obtain $t \downarrow \{a, b, c\} \in (\mathbf{pref}(c^*(ac^*bc^*)^*bc^*(bc^*)^*)) \cdot (\epsilon \cup b)c^* \cap (\mathbf{pref}(b^*(ab^*)^*cb^*(ab^*)^*)) \cdot (\epsilon \cup a)b^*$. After computing this language intersection, we obtain $t \in \mathbf{pref}((ab)^*(bb^* \cup c(ab)^*bb^*)) = \mathbf{lg}P\|Q$. Since t was arbitrary, the network $\{P, Q\}$ is safe. In the example in Figure 2 (c), we compare a specification consisting of a TOGGLE to an implementation consisting of a SELECTOR. The network of the specification is $S = \{\langle\{a\}, \{b, c\}, \mathbf{pref}((abac)^*)\rangle\}$ and the network of the implementation is $I = \{\langle\{a\}, \{b, c\}, \mathbf{pref}((a(b \cup c))^*)\rangle\}$. Figure 2 (c) also shows a "test" network $N = \{R\}$ where $R = \langle\{b, c\}, \{a\}, \mathbf{pref}(ab)\rangle$. We have that $N \cup S$ is safe, as follows. First, note that the only words in $\mathbf{lg}R \cdot (\mathbf{i}R \cup \epsilon)$ are $\epsilon, b, c, a, ab, ac, abb, abc$. For each of these words, one checks that it is either in $\mathbf{lg}R \cap \mathbf{lg}\mathrm{TOGGLE}$ (the words $\epsilon, a,$ and ab) or outside $\mathbf{lg}\mathrm{TOGGLE} \cdot (\mathbf{i}\mathrm{TOGGLE} \cup \epsilon)$ (the words $b, c, ac, abb,$ and abc). Since $\mathbf{a}\mathrm{TOGGLE} = \mathbf{a}R = \{a, b, c\}$, we have that, for any word t, $t_{\mathrm{TOGGLE}} = t_R = t_{N\cup S} = t \downarrow \{a, b, c\}$. Thus the safety condition for $N \cup S$ is $\mathbf{lg}R \cdot (\mathbf{i}R \cup \epsilon) \cap \mathbf{lg}\mathrm{TOGGLE} \cdot (\mathbf{i}\mathrm{TOGGLE} \cup \epsilon) \subseteq \mathbf{lg}R \cap \mathbf{lg}\mathrm{TOGGLE}$, and, by the arguments above, this condition is satisfied. However, the safety condition for $N \cup I$ is not satisfied, as follows. Consider word $t = ac$. We have that $t = t_R \in \mathbf{lg}R \cdot (\mathbf{i}R \cup \epsilon)$ and $t = t_{\mathrm{SELECTOR}} \in \mathbf{lg}\mathrm{SELECTOR} \subseteq \mathbf{lg}\mathrm{SELECTOR} \cdot (\mathbf{i}\mathrm{SELECTOR} \cup \epsilon)$, but $t = t_R \notin \mathbf{lg}R$ and thus $t_{I\cup N} \notin \mathbf{lg}\|(I \cup N)$. Since $N \cup S$ is safe and $N \cup I$ is not safe, we have that $S \not\sqsubseteq_{stn} I$.

For the case with no dangling inputs and no connected outputs, as in Figures 2 (a) and 3, our (absolute) safety condition agrees with "absence of computation interference." (We refer the reader to the version in [8] for comparison purposes, but similar conditions have been defined at least in [7, 20, 23, 25].) 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. Informally speaking, "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).

In the case with dangling inputs, as in Figure 2 (b), our condition for absolute safety 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 [6] and [13]. However, in both [6] and [13], receptiveness is used as a model restriction on processes rather than a correctness condition on networks. 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.

Situations where outputs are shared are normally disallowed. Nevertheless, our safety condition is defined in these situations too. If outputs 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 from absence of computation interference in this case.

Algebraic properties for safety

First, we provide simpler forms for the absolute and relative safety conditions.

PROPOSITION 2 *Network N is safe iff*

$$\mathbf{Ib}N = \emptyset.$$

Rougly speaking, this form of our condition for absolute safety says that the environment cannot do anything wrong, since it is the responsibility of the environment to avoid getting into an input break. In other words, the network behavior is constrained enough to guarantee that no illegal input will be delivered to an element of the network, no matter what the environment does.

In the example in Figure 2 (a), $b \in \mathbf{ib}\{P, Q\} \subseteq \mathbf{Ib}\{P, Q\}$, because $\epsilon \in \mathbf{lg}\|\{P, Q\}, b \in \mathbf{i}P$, $b_P = b \notin \mathbf{lg}P$, and $b \notin \mathbf{ob}N$ because $b \notin \mathbf{o}P$ and $b \in \mathbf{lg}Q$. Thus, the network $\{P, Q\}$ is not safe, as we have discussed before. In Figure 2 (b), suppose there exists $t \in \mathbf{ib}\{P, Q\}$. Then, by the definition of **ib**, we have that $\exists u, v \in \mathcal{U}^*, a \in \mathcal{U}$, and $R \in \{P, Q\}$ such

that $t = uav$, $u \in \mathbf{lg}\|\{P, Q\}$, $a \in \mathbf{i}R$ and $(ua)_R = u_R a \notin \mathbf{lg}R$. Since $u \in \mathbf{lg}\|\{P, Q\}$, $u_R \in \mathbf{lg}R$. However, one checks that both P and Q have the property of *receptivity* (see, e.g., [6, 13]: $\mathbf{lg}P \cdot \mathbf{i}P \subseteq \mathbf{lg}P$ and $\mathbf{lg}Q \cdot \mathbf{i}Q \subseteq \mathbf{lg}Q$. Thus, if $u_R \in \mathbf{lg}R$ and $a \in \mathbf{i}R$, then $u_R a \in \mathbf{lg}R$, in contradiction to the choice of R. It follows that $\mathbf{ib}\{P, Q\}$ must be empty. Therefore, the network $\{P, Q\}$ is safe, as in our previous discussions of this example.

PROPOSITION 3 *For networks S and I,*

$$S \sqsubseteq_{stn} I \;\Leftrightarrow\; \mathbf{Ob}S \subseteq \mathbf{Ob}I \;\wedge\; \mathbf{Ib}S \supseteq \mathbf{Ib}I \;.$$

Informally speaking, this form of the $\sqsubseteq_{stn}$ condition demands that I produces fewer legal outputs than S does, and accepts more legal inputs than S does.

For the example in Figure 2 (c), let P and Q denote the single elements of the specification and implementation, respectively. Notice that $ac \in \mathbf{ob}\{P\} \subseteq \mathbf{Ob}\{P\}$, since $a \in \mathbf{lg}P = \mathbf{lg}\|\{P\}$, $c \in \mathbf{o}P$, and $(ac)_P = ac \notin \mathbf{lg}P$. Also notice that $ac \in \mathbf{lg}\{Q\}$, thus $(ac)_Q = ac \notin \mathbf{o}\{Q\}$, by Lemma 1 (a). Hence, $ac \notin \mathbf{Ob}\{Q\}$. It follows that $\mathbf{Ob}\{P\} \not\subseteq \mathbf{Ob}\{Q\}$, thus $\{P\} \not\sqsubseteq_{stn} \{Q\}$, in agreement to our previous discussions of this example.

The following theorem provides algebraic properties for modular and hierarchical verification, plus reflexivity.

THEOREM 1 *For networks M, N, and O,*

(a) $M \sqsubseteq_{stn} M$,
(b) $M \sqsubseteq_{stn} N \;\wedge\; N \sqsubseteq_{stn} O \;\Rightarrow\; M \sqsubseteq_{stn} O$,
(c) $M \sqsubseteq_{stn} N \;\Rightarrow\; M \cup O \sqsubseteq_{stn} N \cup O$.

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 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 non-negative integers $\{0, 1, 2, \ldots\}$. 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 $abb \leq e$, but $e \not\leq e$ because e is not finite. Since we do not use infinite prefixes at all, we refer to finite prefixes as just prefixes. We extend the projection operation to infinite sequences in the obvious way, by deleting from the sequence the symbols that are not in the projection alphabet.

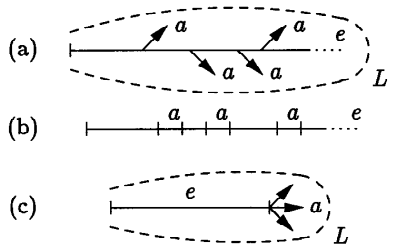

Figure 4. Recurrently enabled and fired symbols.

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

Strong liveness

Informally speaking, the *complete* executions of an asynchronous circuit are (finite or infinite) sequences of all events issued by a system throughout its operation. In contrast to that, the *partial* executions are (finite) sequences that can occur up to a finite time.

Trying to formalize a notion of complete executions of an asynchronous circuit, we have obtained a generic property that unifies a "strong fairness" property of *infinite* sequences (e.g., see [9]) with a "quiescence" property of *finite* sequences (e.g., see [12]). 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 4 (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 4 (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 4 (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 the language 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 $\mathbf{otp}P \subseteq \mathbf{lim}\,\mathbf{lg}P$, 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.

Traplock-freedom

For trace structure P, we also use the following notation:

$$\mathbf{Otp}P \;=\; \{e \in \mathcal{U}^\infty \mid e_P \in \mathbf{otp}P\}\ .$$

Our liveness condition is defined as follows.

Definition 3. Network I is *traplock-free* for network S, written $S \sqsubseteq_{tf} I$, if

$$\mathbf{Otp}\|I \cap \mathbf{lim}\,\mathbf{Lg}\|S \subseteq \mathbf{Otp}\|S\ .$$

Now we check traplock-freedom on the examples in Section 2. In these examples, let S denote the specification network and I the implementation network.

For Figure 1 (a), $c \in \mathbf{Otp}\|I$ because $\mathbf{ren}_{\mathbf{lg}\|I} c = \emptyset$. We also have $c \notin \mathbf{a}\|S$ and thus $c_S = \epsilon$. Thus, $c \in \mathbf{Lg}\|S \subseteq \mathbf{limLg}\|S$. We also have $\mathbf{ren}_{\mathbf{lg}\|S} c_S \ni a$ and $\mathbf{ren}_{\mathbf{lg}\|S} c_S \cap \mathbf{o}\|S \not\subseteq \mathbf{rfi}c_S = \emptyset$. Therefore, $c \notin \mathbf{Otp}\|S$ and the traplock-freedom condition is not satisfied. For Figure 1 (b), consider execution $e = b(cd)^\omega$. We have $e \in \mathbf{Otp}\|I$, because $e_I = e$ and $\mathbf{ren}_{\mathbf{lg}\|I} e_I = \{c, d\} = \mathbf{rfi}e_I$. We also have that $e_S = \epsilon$, thus $e \in \mathbf{limLg}\|S$. Also, $e \notin \mathbf{Otp}\|S$ because $\mathbf{ren}_{\mathbf{lg}\|S} e_s \cap \mathbf{o}\|S = \{a\} \not\subseteq \mathbf{rfi}e_S = \emptyset$. Therefore, the traplock-freedom condition is violated for this example, too. For Figure 1 (c), consider execution $e = ad(ba)^\omega$. We have $e \in \mathbf{Otp}\|I$, because $e_I = e$ and $\mathbf{ren}_{\mathbf{lg}\|I} e_I = \{a, b\} = \mathbf{rfi}e_I$. We also have that $e_S = (ab)^\omega$, thus $e \in \mathbf{limLg}\|S$. Also, $e \notin \mathbf{Otp}\|S$ because $\mathbf{ren}_{\mathbf{lg}\|S} e_s \cap \mathbf{o}\|S = \{b, c\}$, but $c \notin \mathbf{rfi}e_S = \{a, b\}$. Therefore, the traplock-freedom condition is violated for this example, too. For Figure 1 (d), we have that the only output trap of $\|I$ is $e = (ba)^\omega$, because any finite execution recurrently enables either b or a and recurrently fires nothing. For any $e' \in \mathbf{Otp}\|I$, we have $e'_I = e$ and $e'_S = e' \downarrow \{a\} = (e' \downarrow \{a, b\}) \downarrow \{a\} = (e'_I)_S = e_S = a^\omega$. We have that $\mathbf{ren}_{\mathbf{lg}\|S} a^\omega \cap \mathbf{o}\|S = \{a\} = \mathbf{rfi}a^\omega$, thus $a^\omega \in \mathbf{otp}\|S$. It follows that $\mathbf{Otp}\|I \subseteq \mathbf{Otp}\|S$, which implies that $\mathbf{Otp}\|I \cap \mathbf{limLg}\|I \subseteq \mathbf{Otp}\|S$, i.e. the traplock-freedom condition is satisfied.

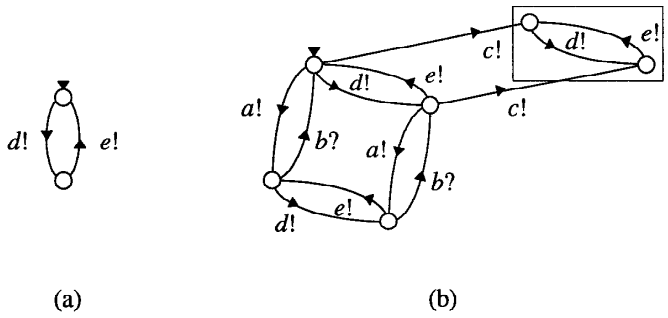

Figure 5. A disconnected part does not fix a flaw.

Details of the traplock-freedom condition

A correctness condition should not be perturbed by the insertion of a redundant, disconnected part in a system. If the $d!$, $e!$ loop in Figure 5 (a) is introduced in the implementation in Figure 1 (a), one obtains the automaton in Figure 5 (b). For us, the newly added part neither fixes nor changes the nature of the fault, because the specification sees exactly the same behavior as before. The danger of deadlock remains, even if some part never stops in the implementation!

For the example in Figure 5, the implementation actually has two elements: $P = \langle\{b\}, \{a, c\}, \mathbf{pref}((ab)^*c)\rangle$ and $Q = \langle\emptyset, \{d, e\}, \mathbf{pref}((de)^*)\rangle$. The composite of P and Q is shown by Figure 5 (b). Consider execution $e = c(de)^\omega$. We have $\mathbf{ren}_{\mathbf{lg}\|I} e_I \cap \mathbf{o}\|I = \{d, e\} \cap \mathbf{o}\|I = \{d, e\}$, and $\mathbf{rfi}e_I = \{d, e\}$. Thus $e \in \mathbf{Otp}\|I$. Also, $e_S = \epsilon$, thus $e \in \mathbf{limLg}\|S$. We have $\mathbf{ren}_{\mathbf{lg}\|S} e_S \cap \mathbf{o}\|S = \{a\} \cap \mathbf{o}\|I = \{a\}$, and $\mathbf{rfi}e_S \not\ni a$. Thus $e \notin \mathbf{Otp}\|S$ and the traplock-freedom condition is violated.

Some subtle situations are shown in Figure 6. In Part (a), the specification is a FORK (e.g., see [8]) and the implementation is an ASYMMETRIC FORK (unequal branch delays). The specification reflects the fact that branch delays are not known in advance and may vary. In VLSI circuits, one branch delay may turn out to be longer than the other. Note that this is normally acceptable. Our liveness condition is satisfied here. This point of view follows naturally from considering fairness with respect to symbols rather than words. Our condition is such that firing c in the framed specification cycle exempts the execution point from eventually having to take the exit c from the cycle. In Part (b), the specification and implementation are the asynchronous components SELECTOR and TOGGLE (e.g., see [8]). We think that TOGGLE is a good realization of a SELECTOR in most practical situations. Again, this is due to our use of fairness with respect to symbols rather than words. In [18], we also define a liveness condition with respect to words, and discuss its disadvantages.

For the examples in Figure 6, notice that $\mathbf{lg}\|I \subseteq \mathbf{lg}\|S$, thus $\mathbf{limLg}\|I \subseteq \mathbf{limLg}\|S$, and thus $\mathbf{Otp}\|I \subseteq \mathbf{limLg}\|S$. To show traplock-freedom under these restrictions, we only need to prove $\mathbf{Otp}\|I \subseteq \mathbf{Otp}\|S$. Since $\mathbf{a}\|I = \mathbf{a}\|S$, it suffices to show $\mathbf{otp}\|I \subseteq \mathbf{otp}\|S$. In

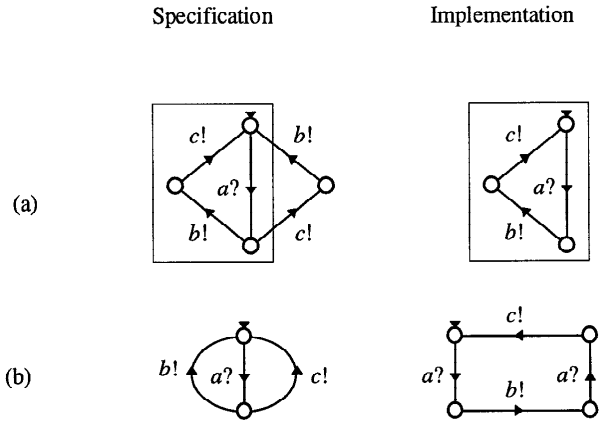

Figure 6. Word-fairness is not desirable.

Figure 6 (a), the only finite output traps of $\|I$ are those traces that do not recurrently enable an output of $\|I$, that is, the words in $(abc)^*$. All these words are output traps for $\|S$, because they do not recurrently enable an output of $\|S$. The only infinite output trap of $\|I$ can be $e = (abc)^\omega$, because this is the only element of $\mathbf{lim\,lg}\|I$. We don't need to prove that e is actually an output trap of $\|I$, but we'll show directly that it is an output trap of $\|S$. We have $\mathbf{ren}_{\mathbf{lg}\|S}e_S = \{a, b, c\}$. Thus, $\mathbf{ren}_{\mathbf{lg}\|S}e_S \cap \mathbf{o}\|S = \{b, c\} = \mathbf{rfi}e_S$. Therefore, the traplock-freedom condition is satisfied. Similarly, in Figure 6 (b), the only finite output traps of $\|I$ are the words in $(abac)^* \cup (abac)^*ab$. All these words are output traps for $\|S$, because they do not recurrently enable an output of $\|S$. The only infinite output trap of $\|I$ can be $e = (abac)^\omega$, because this is the only element of $\mathbf{lim\,lg}\|I$. We have $\mathbf{ren}_{\mathbf{lg}\|S}e_S = \{a, b, c\}$. Thus, $\mathbf{ren}_{\mathbf{lg}\|S}e_S \cap \mathbf{o}\|S = \{b, c\} = \mathbf{rfi}e_S$. Therefore, the traplock-freedom condition is satisfied.

Figure 7 illustrates some subtleties involving choice (output non-determinism). For Part (a), we take the position that the implementation is not distinguishable from the specification by any external experiment on the implementation. In any experiment, the implementation operates only once. Traplock-freedom is satisfied here. If there are 'many' such implementations available for the experiment, or if the implementation can be used 'many' times, then a more appropriate representation should be that of Part (b), where the choice must be made infinitely often. Here, our condition is not satisfied, because the implementation stays forever in the framed subgraph, while the specification demands a $b!$. In [18], we also define a liveness condition which considers Part (a) (and similar cases) unacceptable, and discuss the disadvantages of that point of view.

In Figure 7 (a), the output traps of $\|I$ can only be finite, like the limits of $\mathbf{lg}\|I$. Thus, the output traps of $\|I$ do not recurrently enable an output. Thus, the only output trap of $\|I$ is the word a. Since $\mathbf{a}\|S = \mathbf{a}\|I$, we have that, for any execution $e \in \mathcal{U}^\infty$ such that $e_I = a$, we

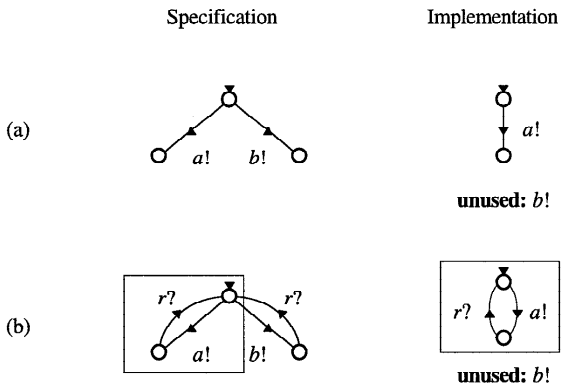

Figure 7. Illustrations of choice.

have $e_S = a$ as well. Thus e_S is an output trap of $\|S$, since $\mathbf{ren}_{\mathbf{lg}\|S} a = \emptyset$. In Figure 7 (b), consider execution $e = (ar)^\omega$. We have that $e_I = e$ and $\mathbf{ren}_{\mathbf{lg}\|I} e_I = \{a, r\} = \mathbf{rfi} e_I$, thus $e_I \in \mathbf{otp} I$. We also have that $e_S = e \in \mathbf{lim\,lg}\|S$. However, $\mathbf{ren}_{\mathbf{lg}\|S} e_S = \{a, b, r\}$ while $b \notin \mathbf{rfi} e_S$. Thus, $e_S \notin \mathbf{otp}\|S$ and traplock-freedom is violated.

A criterion for finding output traps is provided by the following property.

PROPOSITION 4 *For network N, we have*

$$\bigcap_{P \in N} \mathbf{Otp} P \subseteq \mathbf{Otp}\|N.$$

The example in Figure 8 is adapted from [2]. The circuit in Figure 8 (a) represents an unfair implementation for an arbiter. The implementation uses two "mutual exclusion elements", or "mutex" for short, whose trace structures are shown by the automaton in Figure 8 (b). A typical application of a mutex is to arbitrate the exclusive access of two independent processes to a shared resource. Each of the two processes sees an input and an output of the mutex. A process requests the resource by raising the corresponding input, and the mutex grants the resource by raising the corresponding output in response to a raised input. The resource is released by the process by lowering the corresponding input, after which the mutex lowers the corresponding output. Mutual exclusion is realized by ensuring that the two outputs are not both high at any particular time. The specification is also a mutex, whose terminals are labelled as in Figure 8 (c). The implementation also contains several "delay elements", a flip-flop, which we assume to start in the state where Q is high, and two forks. The automata in Figure 8 (d), (f), and (e) represent the trace structure of a delay element with input I and output O, the trace structure of the flip-flop (starting with Q high) and the trace structure of a fork with input I and outputs O0 and O1. The Q output of the flip-flop may change when S or R are high: it is reset when R is high and it is set when S is high and R is low. In the circuit in Figure 8 (a), we assume that n is initially high and all other

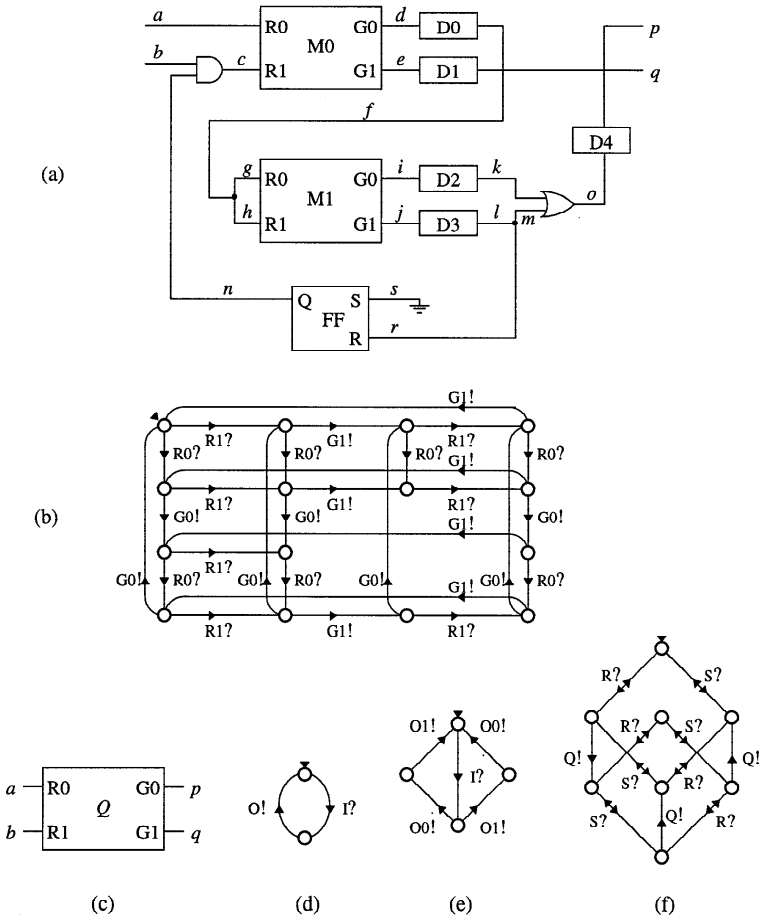

Figure 8. Unfair implementation of mutual exclusion.

signals are initially low. The second MUTEX has the inputs wired together so that it enters a metastable state when a rising edge is applied. If the G1 branch wins, then the flip-flop is reset. After that, any c request is retracted and no further c requests are taken by the first MUTEX, because of the AND gate. Because of this danger, the implementation is unfair. Also, the traplock-freedom condition is violated, as follows. Consider the infinite execution $e = \alpha\beta(\gamma\gamma\delta\delta)^\omega$, where $\alpha = adbcfghjlrmnocp$, $\beta = adfghjlrmop$, $\gamma = adfghikop$, and $\delta = adfghjlmrop$. Execution e resets the flip-flop and then never grants a q in response to the initial b event. One verifies that, for each element P of the implementation network, $e_P \in \mathbf{otp}P$. In particular, note that $e_{\mathrm{M}0} = adccad(adadadad)^\omega$, $\mathbf{ren}_{\mathbf{lg}_{\mathrm{M}0}} e_{\mathrm{M}0} = \{a, c, d\}$, and $\mathbf{rfi}e_{\mathrm{M}0} = \{a, d\} \supseteq \{a, c, d\} \cap \{d, e\} = \mathbf{ren}_{\mathbf{lg}_{\mathrm{M}0}} e_{\mathrm{M}0} \cap \mathbf{o}\mathrm{M}0$. Let I be the implementation network. Since $\forall P \in I : e \in \mathbf{Otp}P$, then, by Proposition 4, we have that $e \in \mathbf{Otp}\|I$.

However, let S be the specification network, and let Q be the only element of S, which is the mutex shown in Figure 8 (c). We have that $e_Q = abpap(apapapap)^\omega = abp(ap)^\omega$, and $\mathbf{rfi}e_Q = \{a, p\}$, while $\mathbf{ren}_{\mathbf{lg}Q}e_Q = \{a, b, p, q\}$ and thus $\mathbf{rfi}e_Q \not\supseteq \mathbf{ren}_{\mathbf{lg}Q}e_Q \cap \mathbf{o}Q = \{p, q\}$. Thus, $e \notin \mathbf{Otp}\|S = \mathbf{Otp}Q$. Also, one verifies that $e_Q \in \mathbf{lim\,lg}Q$ thus $e \in \mathbf{limLg}Q$. Therefore, by the counter-example of e, we have $\mathbf{Otp}\|I \cap \mathbf{limLg}\|S \not\subseteq \mathbf{Otp}\|S$, which violates the traplock-freedom condition.

Algebraic properties of traplock-freedom

The traplock-freedom condition satisfies reflexivity, and, under certain restrictions, transitivity and compatibility with union, which are useful for hierarchical and modular verification.

PROPOSITION 5 *For network N, $N \sqsubseteq_{tf} N$.*

THEOREM 2 *For networks M, N and O,*

$$M \sqsubseteq_{tf} N \;\wedge\; N \sqsubseteq_{tf} O \;\wedge\; M \sqsubseteq_{stn} N \;\wedge\; N \sqsubseteq_{stn} O \;\Rightarrow M \sqsubseteq_{tf} O\,.$$

The following definition introduces a connectivity (alphabet) restriction needed for the compatibility with union theorem.

Definition 4. Network N is *output consistent* for network M, written $M \sqsubseteq_{oc} N$, if

$$\begin{array}{l} \quad \forall P, Q \in N : \mathbf{o}P \cap \mathbf{o}Q = \emptyset \\ \wedge\; \forall P \in N, Q \in M : \mathbf{o}P \cap \mathbf{i}Q = \emptyset\,. \end{array}$$

THEOREM 3 *For networks M and N such that $M \sqsubseteq_{tf} N$, we have*

(a) *if $M \sqsubseteq_{stn} N$ and $M \cup O \sqsubseteq_{oc} N \cup O$ then $M \cup O \sqsubseteq_{tf} N \cup O$*
(b) *if $\mathbf{Ib}N = \emptyset$ and $(\forall P, Q \in N : \mathbf{o}P \cap \mathbf{o}Q = \emptyset)$ then $M \cup O \sqsubseteq_{tf} N \cup O$*

Theorem 2 grants hierarchical verification for traplock-freedom, provided that the relative safety condition is satisfied as well. This is not a hard restriction, since correct implementations need to satisfy the relative safety condition as well, with respect to their specifications.

Theorem 3 grants modular verification for traplock-freedom, also under certain restrictions. In Part (a), there is a relative safety restriction like that for transitivity. There is also a connectivity restriction. One part of the restriction in Definition 4 demands that the implementation network should not have connected outputs. This first part of the restriction is justifiable at least for digital circuits, where outputs of parts cannot be connected anyway. The other part of the restriction demands that the outputs of the implementation should not be inputs for any process of the specification; this demand may be too strong. In Part (b) of Theorem 3, we only use the first part of the connectivity restriction above, but we impose an absolute safety condition on the implementation. Thus, Part (b) of the theorem provides another criterion for compatibility with union.

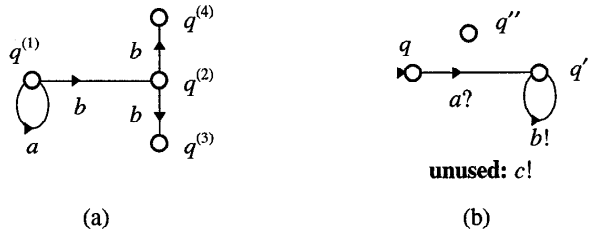

Figure 9. (a) A state graph; (b) a behavior automaton.

6. Non-Deterministic Processes as 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 asynchronous circuits using so-called behavior automata, which, formally, are 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.

Our automaton model was inspired by the "state graphs" used previously in trace theory to represent trace structures having regular languages.

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 labelled *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 9 (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 *unambiguous* if no two edges leaving the same state have the same label. For example, the state graph in Figure 9 (a) is not unambiguous.

The unambiguity restriction imposes that the state graphs be deterministic. However, we prefer the label 'unambiguous' to avoid confusion with deterministic systems. We do model non-deterministic systems by unambiguous state graphs; see, for example, Figure 1 (c). In fact, deterministic automata can represent exactly the same languages and alphabets as non-deterministic ones, and, as we shall see, we use automata only to specify languages and alphabets. The unambiguity restriction means just that the options of a choice should be represented in our model by different output symbols; hence the term "unambiguous". 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 [18] we also stated a version of our liveness condition in automata with ambiguous choice and with a CCS-style silent action. The condition we obtained is quite complex compared to traplock-freedom. We settled for the unambiguity restriction, for simplicity without loss of modeling power.

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.

Behavior automata are rendered as described in Section 2. Figure 9 (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 9 (b); then $\mathbf{a}A = \{a, b, c\}$ and $\mathbf{lg}A = \mathbf{pref}(ab^*)$.

Trace structures of behavior automata

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 9 (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 9 (b), then $A \diamond \epsilon = q$ and $A \diamond abb = q'$. For arbitrary behavior automaton A, we have $A \diamond \epsilon = \mathbf{init}A$.

The leads-to operation is extended to infinite sequences. For behavior automaton A and sequence e in $\mathbf{lim\,lg\,tr}A$, 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 $\mathbf{lim\,lg\,tr}A$ 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 6 *For behavior automaton A and sequence e in $\mathbf{lim\,lg\,tr}A$, $A \diamond e$ is a knot.*

PROPOSITION 7 *For behavior automaton A and knot G in A, there exists a sequence e in $\mathbf{lim\,lg\,tr}A$ 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:

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

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*.

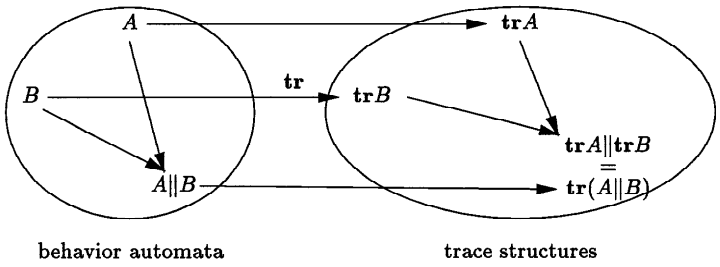

Figure 10. Commutative diagram of parallel compositions.

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 $e_A = e \downarrow \mathbf{a}A$.

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

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

The following theorem links the parallel compositions of behavior automata and trace structures by the **tr** mapping, as illustrated by the commutative diagram in Figure 10. 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 4 *For behavior automata A and B, we have $\mathbf{tr}(A\|B) = \mathbf{tr}A \parallel \mathbf{tr}B$.*

Theorem 4 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 8 *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* $\mathbf{lim}\,\mathbf{lg}\,\mathbf{tr}(A\|B)$, $((A\|B)\diamond e)_A = A \diamond e_A$ *and* $((A\|B)\diamond e)_B = B \diamond e_B$.

7. Traplock-freedom on Behavior Automata

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 $\mathbf{en}_A p = \{a \in \mathbf{a}A \mid \exists p' \in \mathbf{st}A \text{ such that } (p,a,p') \in \mathbf{ed}A\}$, 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 9 (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* $\mathbf{lim}\,\mathbf{lg}\,\mathbf{tr}A$ *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 5. 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.

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.

To illustrate the traplock-freedom condition on automata, consider again the examples in Section 2. In Figure 1 (a), (b), and (c), the framed knot of the implementation is a trap of the implementation, and it corresponds to the framed knot of the specification, which is not a trap of the specification. Thus, traplock-freedom is violated. The implementation in Figure 1 (d) is traplock-free for its specification, because the only trap of the implementation is the whole graph, corresponding to the whole graph of the specification, which is also a trap.

Equivalence of the traplock-freedom conditions

The following theorem shows the equivalence of our traplock-freedom conditions on automata and trace structures.

THEOREM 5 *For behavior automata S and I, I is traplock-free for S iff* $\{\mathbf{tr}S\} \sqsubseteq_{tf} \{\mathbf{tr}I\}$.

Theorem 5 has important consequences. First, it proves the equivalence of two forms of our liveness condition 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 8. 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. An Algorithm for Verification of Liveness

In this section we introduce an algorithm for verifying the traplock-freedom condition.

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.

1. **predicate** Is_traplock-free? (S, I)
2. **for** each b in $\mathbf{o}S$ **do**
3. build C by removing from $\mathbf{gr}(S\|I)$ all edges firing b
4. **for** each state (q', q'') of C such that $b \in \mathbf{en}_S q'$ **do**

```
5.              repeat
6.                  let H be the strongly connected component
7.                      of (q', q'') in C
8.                  if H_I is an output trap in I then
9.                      return false
10.                 build C by removing from H all states (p', p'')
11.                     that satisfy en_I p'' ∩ oI ⊈ fiH
12.             while (q', q'') ∈ stC
13.         end for
14.     end for
15.     return true
16. end predicate
```

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 data structures in this algorithm.

Practical considerations

The method for verification of liveness sketched in [6] has worst-case time costs growing exponentially with the square of the size of the specification. This is due to the use of non-deterministic automata in [6]. Due to the use of deterministic automata, the costs of our method are polynomial in the sizes of the state graphs involved. To assess the practicality of our algorithm, we compare it to the verification algorithm for *safety* in [6], which also uses deterministic automata ([6] p. 81). 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 [6]. 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 [6], most states of a flawed implementation are typically such spurious states.

Although the algorithm has polynomial time, the costs of the overall verification method are still exponential. These costs include computing I as a parallel composition, which are exponential in the number of components (just as in [6]). 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. Another caveat is that we start our verification procedure from behavior automata, which are (formally) deterministic. If non-deterministic automata were used instead, there would be extra costs for determinization or for direct checking of traplock-freedom on non-deterministic automata. Note, however, that in our method non-deterministic automata would have exactly the same modeling power as behavior automata, since we interpret automata just as trace structures, consisting of two alphabets and a language of finite words.

9. Concluding Remarks

The main benefit of our approach to liveness verification is that the users do not have to specify explicitly liveness properties as long as the users provide trace structures or equivalent descriptions, which specify explicitly the alphabets and the finite executions of their systems. Notice that our behavior automata have many similarities to omega-automata; however, they also have differences that are very important for our purposes. From a practical point of view, in our approach the users do not specify explicitly an omega-language, by annotating states as is done for Büchi automata or for the "mixed automata" in [6], or by selecting pairs of states as is done for Streett or Rabin automata. From a theoretical point of view, we only interpret our automata as trace structures, through

the **tr** mapping, and this interpretation is sufficient to decide the traplock-freedom condition, by Theorem 5. Recall that a trace structure consists of two alphabets and a finitary language; thus, the only information we use from a behavior automaton consists of its alphabets and finitary language. On the other hand, two omega-automata with the same alphabets and the same finitary languages may have different omega-languages. The "mixed automata" in [6] can also have different omega-languages for the same alphabets and finitary languages. Thus, omega-automata and Dill's mixed automata contain more information than specified by their alphabets and finitary languages.

We have implemented a program using our algorithm for the verification of traplock-freedom. The program uses explicit state-space exploration, that is, it visits the states of $S\|I$ one by one. The explicit state-space exploration approach is not very efficient, and we have only used the program on small examples, for the purpose of testing the condition. A more efficient implementation might use BDD-based manipulation instead of explicit state-space exploration. This is another direction for further work.

As we have pointed out in the introduction, asynchronous circuits are just a particular case of concurrent systems. Now, notice that there are no formal restrictions in our approach to liveness verification that would prevent its applicability to other types of concurrent systems, since concurrent programs and protocols can also be represented as networks of components where each component is represented by its alphabets and finite traces. Further research should check whether our liveness condition with its implicitly assumed liveness properties is indeed meaningful for other types of concurrent systems as well, and, if not, what are the limitations.

Acknowledgments

We are grateful for comments on previous versions to Joanne Atlee, David Dill, and to the University of Waterloo Maveric group, especially Robert Berks and John Segers. We are grateful to Igor Benko for pointing out a flaw in the condition for relative safety we had used in previous versions of this manuscript. We are also indebted to Jo Ebergen, Tom Verhoeff, and Charles Molnar for very important comments and suggestions.

This research was supported by a grant and a scholarship from the Information Technology Research Centre of Ontario, by an Ontario Graduate Scholarship, and by Grant No. OGP0000871 from the Natural Sciences and Engineering Research Council of Canada. Earlier versions of this paper were published as [17] and [18].

Appendix A

Trace Structure Proofs

PROPOSITION 1 *For network N,*
(a) $\mathbf{ob}N \cap \mathbf{ib}N = \mathbf{ob}N \cap \mathbf{lg}\|N = \mathbf{ib}N \cap \mathbf{lg}\|N = \emptyset$, *and*
(b) $\mathbf{ob}N \cup \mathbf{ib}N \cup \mathbf{lg}\|N = (\mathbf{a}\|N)^*$.

Proof:
(a) Let $t \in \mathbf{ob}N$. Then $\exists P \in N$ such that $t = uav$ with $u \in \mathbf{lg}\|N$, $a \in \mathbf{o}P$, and $u_P a \notin \mathbf{lg}P$. Thus, $ua \notin \mathbf{lg}\|N$, and, since $\mathbf{lg}\|N$ is prefix-closed, $t \notin \mathbf{lg}\|N$.

Let $t \in \mathbf{ib}N$. We have $\exists P \in N$ such that $t = uav$ with $u \in \mathbf{lg}\|N$, $a \in \mathbf{i}P$, and $u_P a \notin \mathbf{lg}P$. Thus, $ua \notin \mathbf{lg}\|N$, and, since $\mathbf{lg}\|N$ is prefix-closed, $t \notin \mathbf{lg}\|N$.

By the definition of $\mathbf{ib}N$, we also have $\mathbf{ib}N \cap \mathbf{ob}N = \emptyset$.

(b) Let $t \in (\mathbf{a}\|N)^* - \mathbf{lg}\|N$. Since $\epsilon \in \mathbf{lg}\|N$, there exist u, a, v such that $t = uav$, $u \in \mathbf{lg}\|N$, $a \in \mathbf{a}\|N$, $ua \notin \mathbf{lg}\|N$. Consider the following cases.

- There exists $P \in N$ such that $a \in \mathbf{o}P$ and $(ua)_P \notin \mathbf{lg}P$. In this case, $t \in \mathbf{ob}N$, by the definition of **ob**.
- There does not exist $P \in N$ such that $a \in \mathbf{o}P$ and $(ua)_P \notin \mathbf{lg}P$. We show that $t \in \mathbf{ib}N$. Since $ua \notin \mathbf{lg}\|N$, there exists $P \in N : (ua)_P \notin \mathbf{lg}P$. By the case assumption, $a \notin \mathbf{o}P$. However, since $u_P \in \mathbf{lg}P$, we have that $a \in \mathbf{a}P$. Thus, $a \in \mathbf{i}P$. To show $t \in \mathbf{ib}N$, it only remains to show that $t \notin \mathbf{ob}N$.

  Suppose that $t \in \mathbf{ob}N$. By the definition of $\mathbf{ob}N$, there exist $Q \in N$, $u', v' \in \mathcal{U}^*$, and $a \in \mathcal{U}$ such that $t = u'av'$, $u' \in \mathbf{lg}\|N$, $a \in \mathbf{o}Q$, and $(u'a)_Q \notin \mathbf{lg}Q$. Thus, $u'a \notin \mathbf{lg}\|N$. Since $\mathbf{lg}\|N$ is prefix-closed, both u and u' satisfy the defining property of the longest prefix of t in $\mathbf{lg}\|N$. Thus $u' = u$. It follows that there exists $Q \in N$ such that $(ua)_Q \notin \mathbf{lg}Q$ and $a \in \mathbf{o}Q$, in contradiction to the case assumption.

  Therefore, in this case we cannot have $t \in \mathbf{ob}N$, and, by the properties of ua discussed above, we conclude $t \in \mathbf{ib}N$ in this case.

We have shown that $(\mathbf{a}\|N)^* - \mathbf{lg}\|N \subseteq \mathbf{ib}N \cup \mathbf{ob}N$, which proves the property, considering that $\mathbf{lg}\|N$, $\mathbf{ib}N$, and $\mathbf{ob}N$ are subsets of $(\mathbf{a}\|N)^*$. ■

PROPOSITION 2 *Network N is safe iff* $\mathbf{Ib}N = \emptyset$.

Proof:

($\Rightarrow$) Suppose $\exists t \in \mathbf{Ib}N$. Then, $t_N \in \mathbf{ib}N$, and, by the definition of **ib**, there exists $P \in N$ such that $t_N = uav$ with $u \in \mathbf{lg}\|N$, $ua \notin \mathbf{lg}\|N$, and $a \in \mathbf{i}P$.

For any Q such that $a \in \mathbf{o}Q$, we have $u_Q a \in \mathbf{lg}Q$; otherwise we would have $t_N \in \mathbf{ob}N$, contradicting, by Proposition 1 (a), the fact that $t_N \in \mathbf{ib}N$. Thus, $\forall Q \in N$, we have either $a \notin \mathbf{a}Q$, in which case $(ua)_Q = u_Q \in \mathbf{lg}Q$, or $a \in \mathbf{o}Q$, in which case $(ua)_Q \in \mathbf{lg}Q$, or $a \in \mathbf{i}Q$, in which case $(ua)_Q \in \mathbf{lg}Q \cdot \mathbf{i}Q$. Hence,

$$\forall Q \in N : (ua)_Q = u_Q a \in \mathbf{lg}Q \cdot (\mathbf{i}Q \cup \epsilon) \ .$$

Still, $(ua)_P \notin \mathbf{lg}P$, hence $(ua)_N \notin \mathbf{lg}\|N$, contradicting the hypothesis that N is safe.

($\Leftarrow$) Suppose the safety condition is violated by at least one word, i.e., there exists $t \in \mathcal{U}^*$ such that $\forall P \in N : t_P \in \mathbf{lg}P \cdot (\mathbf{i}P \cup \epsilon)$ but $t_N \notin \mathbf{lg}\|N$.

Since $\forall P \in N : t_P \in \mathbf{lg}P \cdot (\mathbf{i}P \cup \epsilon)$, we have that $t \notin \mathbf{ob}N$. By Proposition 1 (b),

$$t_N \notin \mathbf{lg}\|N \wedge t_N \notin \mathbf{ob}N \Rightarrow t_N \in \mathbf{ib}N \ ,$$

contradicting the hypothesis that $\mathbf{Ib}N = \emptyset$. ■

LEMMA 1 *For networks M and N,*

$$\mathbf{Ib}(M \cup N) = \emptyset \Leftrightarrow \mathbf{Ib}N \cap \overline{\mathbf{Ob}M} = \mathbf{Ib}M \cap \overline{\mathbf{Ob}N} = \emptyset \ .$$

Proof:
($\Rightarrow$) Suppose $\mathbf{Ib}N \cap \overline{\mathbf{Ob}M}$ is non-empty and let $t \in \mathbf{Ib}N \cap \overline{\mathbf{Ob}M}$. By Proposition 1 (a), $t \notin \mathbf{Lg}\|(M \cup N)$. Since $\mathbf{Lg}\|(M \cup N)$ is prefix-closed and contains ϵ, we have that $\exists u, a, v : t = uav \wedge u \in \mathbf{Lg}\|(M \cup N) \wedge ua \notin \mathbf{Lg}\|(M \cup N)$. That is, u is the longest prefix of t in $\mathbf{Lg}\|(M \cup N)$. Since $\mathbf{Lg}\|N$ and $\mathbf{Lg}\|M$ are also prefix-closed and contain ϵ, there exist t' the longest prefix of t in $\mathbf{Lg}\|N$ and t'' the longest prefix of t in $\mathbf{Lg}\|M$. There are the following cases for t.
• $t' \leq t''$. Since $\mathbf{Lg}\|(M \cup N) = \mathbf{Lg}\|M \cap \mathbf{Lg}\|N$, and any prefix of t'' is also in $\mathbf{Lg}\|M$, we have that t' is the longest prefix of t in $\mathbf{Lg}\|(M \cup N)$, thus $u = t'$. Since $t \in \mathbf{Ib}N$, we have that $\exists P \in N : (a \in \mathbf{i}P \wedge ua \notin \mathbf{Lg}P)$ and $t \notin \mathbf{Ob}N$, thus $\forall Q \in N : (a \in \mathbf{o}Q \rightarrow ua \in \mathbf{Lg}Q)$. Since $t \notin \mathbf{Ob}M$ either, $\forall Q \in M : (a \in \mathbf{o}Q \rightarrow ua \in \mathbf{Lg}Q)$. Summing up, we have

$$\begin{array}{l} t = uav \wedge u \in \mathbf{Lg}\|(M \cup N) \\ \wedge \exists P \in N \cup M : (a \in \mathbf{i}P \wedge ua \notin \mathbf{Lg}P) \\ \wedge \forall Q \in N \cup M : (a \in \mathbf{o}Q \rightarrow ua \in \mathbf{Lg}Q) \end{array}$$

By the definitions of **Ib** and **ib**, it follows that $t \in \mathbf{Ib}(M \cup N)$.
• $t'' \leq t'$ and $t'' \neq t'$. Then, similarly to the previous case, $u = t''$. Thus $t'' \neq t$ and thus $t \notin \mathbf{Lg}\|M$. By Proposition 1, since $t \notin \mathbf{Ob}M$ either, we have $t \in \mathbf{Ib}M$. Thus, $\exists P \in M : (a \in \mathbf{i}P \wedge ua \notin \mathbf{Lg}P)$ and $\forall Q \in M : (a \in \mathbf{o}Q \rightarrow ua \in \mathbf{Lg}Q)$. Also, $u \leq t' \leq t$, $u \neq t'$, and $ua \leq t$ imply that $ua \leq t'$. Thus $ua \in \mathbf{Lg}\|N$, since $\mathbf{Lg}\|N$ is prefix-closed. Thus $\forall Q \in N : ua \in \mathbf{Lg}Q$. Summing up, we have

$$\begin{array}{l} t = uav \wedge u \in \mathbf{Lg}\|(M \cup N) \\ \wedge \exists P \in M \cup N : (a \in \mathbf{i}P \wedge ua \notin \mathbf{Lg}P) \\ \wedge \forall Q \in M \cup N : (a \in \mathbf{o}Q \rightarrow ua \in \mathbf{Lg}Q) \end{array}$$

By the definitions of **Ib** and **ib**, it follows that $t \in \mathbf{Ib}(M \cup N)$.

In both cases, we obtain that $\mathbf{Ib}(M \cup N)$ is non-empty, in contradiction to the hypothesis.

Thus, we have $\mathbf{Ib}N \cap \overline{\mathbf{Ob}M} = \emptyset$, and, symmetrically, $\mathbf{Ib}M \cap \overline{\mathbf{Ob}N} = \emptyset$.
($\Leftarrow$) Suppose $\mathbf{Ib}(M \cup N) \neq \emptyset$ and let $t \in \mathbf{Ib}(M \cup N)$. It follows from the definitions of **Ib**, **ib**, and **Lg** that

$$\begin{array}{l} \exists u, a, v : t = uav \wedge u \in \mathbf{Lg}\|(M \cup N) \\ \quad \wedge \exists P \in M \cup N : (a \in \mathbf{i}P \wedge ua \notin \mathbf{Lg}P) \\ \quad \wedge \forall Q \in M \cup N : (a \in \mathbf{o}Q \rightarrow ua \in \mathbf{Lg}Q) \end{array}$$

Since $\exists P \in M \cup N : ua \notin \mathbf{Lg}P$, we have that $ua \notin \mathbf{Lg}\|(M \cup N)$. From $t = uav$, $u \in \mathbf{Lg}\|(M \cup N)$, $ua \notin \mathbf{Lg}\|(M \cup N)$, and $\forall Q \in M \cup N : (a \in \mathbf{o}Q \rightarrow ua \in \mathbf{Lg}Q)$, it follows that $t \notin \mathbf{Ob}N$ and $t \notin \mathbf{Ob}M$.

Since $\exists P \in M \cup N : (a \in \mathbf{i}P \wedge ua \notin \mathbf{Lg}P)$, we have either that $P \in M$ or $P \in N$.

If $P \in M$, then

$$\begin{array}{l} \exists u, a, v : t = uav \wedge u \in \mathbf{Lg}\|M \\ \quad \wedge \exists P \in M : (a \in \mathbf{i}P \wedge ua \notin \mathbf{Lg}P) \\ \quad \wedge \forall Q \in M : (a \in \mathbf{o}Q \rightarrow ua \in \mathbf{Lg}Q) \end{array}$$

and thus $t \in \mathbf{Ib}M$. Since $t \notin \mathbf{Ob}N$, $\mathbf{Ib}M \cap \overline{\mathbf{Ob}N} \neq \emptyset$. Similarly, if $P \in N$ we obtain $\mathbf{Ib}N \cap \overline{\mathbf{Ob}M} \neq \emptyset$. In both cases, the hypothesis is contradicted and thus the assumption that $\mathbf{Ib}(M \cup N) \neq \emptyset$ is false. ■

LEMMA 2 *For networks M and N,*

(a) $\mathbf{Lg}\|N \cap \mathbf{Ob}M \subseteq \mathbf{Ob}(N \cup M)$
(b) $\mathbf{Lg}\|N \cap \mathbf{Ib}M \subseteq \mathbf{Ib}(N \cup M)$
(c) $\mathbf{Ib}N \cap \mathbf{Ib}M \subseteq \mathbf{Ib}(N \cup M)$

Proof:
(a) Let $t \in \mathbf{Lg}\|N \cap \mathbf{Ob}M$. Let u, a, v such that $t = uav$, $u \in \mathbf{Lg}\|M$, and $\exists P \in M : a \in \mathbf{o}P \wedge ua \notin \mathbf{Lg}P$.

Since $\mathbf{Lg}\|N$ is prefix-closed, we have $u \in \mathbf{Lg}\|N$. Thus, $u \in \mathbf{Lg}\|(M \cup N)$ and $\exists P \in (M \cup N) : a \in \mathbf{o}P \wedge ua \notin \mathbf{Lg}P$. Thus, $t \in \mathbf{Ob}(M \cup N)$.
(b) Let $t \in \mathbf{Lg}\|N \cap \mathbf{Ib}M$. Let u, a, v such that $t = uav$, $u \in \mathbf{Lg}\|M$, $\exists P \in M : a \in \mathbf{i}P \wedge ua \notin \mathbf{Lg}P$, and $\neg\exists Q \in M : a \in \mathbf{o}Q \wedge ua \notin \mathbf{Lg}Q$.

Also, we have $\neg\exists Q \in N : a \in \mathbf{o}Q \wedge ua \notin \mathbf{Lg}Q$, because $\mathbf{Lg}\|N$ is prefix-closed and thus $ua \in \mathbf{Lg}\|N$. Also, $u \in \mathbf{Lg}\|(M \cup N)$ and $\exists P \in (M \cup N) : a \in \mathbf{i}P \wedge ua \notin \mathbf{Lg}P$. Thus, $t \in \mathbf{Ib}(M \cup N)$.
(c) Let $t \in \mathbf{Ib}N \cap \mathbf{Ib}M$. Let u, a, v such that $t = uav$, $u \in \mathbf{Lg}\|M$, $\exists P \in M : a \in \mathbf{i}P \wedge ua \notin \mathbf{Lg}P$, and $\neg\exists Q \in M : a \in \mathbf{o}Q \wedge ua \notin \mathbf{Lg}Q$. Let u', a', v' such that $t = u'a'v'$, $u' \in \mathbf{Lg}\|N$, $\exists P' \in N : a' \in \mathbf{i}P' \wedge u'a' \notin \mathbf{Lg}P'$, and $\neg\exists Q \in N : a' \in \mathbf{o}Q \wedge u'a' \notin \mathbf{Lg}Q$.

We have either $u \leq u'$ or $u' \leq u$ (or both). Assume w.l.o.g. $u \leq u'$. Then, $u \in \mathbf{Lg}\|M$, $\exists P \in M : a \in \mathbf{i}P \wedge ua \notin \mathbf{Lg}P$, $\neg\exists Q \in M : a \in \mathbf{o}Q \wedge ua \notin \mathbf{Lg}Q$. There are two subcases for $u \leq u'$. If $u \neq u'$, then $ua \leq u'$ and $ua \in \mathbf{Lg}\|N$. If $u = u'$, then $a = a'$. Thus, in both subcases, $\neg\exists Q \in N : a \in \mathbf{o}Q \wedge ua \notin \mathbf{Lg}Q$. Thus, $\neg\exists Q \in (M \cup N) : a \in \mathbf{o}Q \wedge ua \notin \mathbf{Lg}Q$, and thus $ua \in \mathbf{Ib}(M \cup N)$ and $t \in \mathbf{Ib}(M \cup N)$. ■

PROPOSITION 3 *For networks S and I,*

$$S \sqsubseteq_{stn} I \;\Leftrightarrow\; \mathbf{Ob}S \subseteq \mathbf{Ob}I \;\wedge\; \mathbf{Ib}S \supseteq \mathbf{Ib}I \;.$$

Proof:
($\Rightarrow$) Suppose $\exists t \in \mathbf{Ib}I - \mathbf{Ib}S$. We construct a 'test' network that invalidates $S \sqsubseteq_{stn} I$. Let $t = uav$ with $u \in \mathbf{Lg}\|I$, $a \in \mathcal{U}$, and $ua \notin \mathbf{Lg}\|I$. Let $u = a_1 \ldots a_n$, with $a_1, \ldots, a_n \in \mathcal{U}$. Let $A = \mathbf{a}\|S \cup \mathbf{a}\|I \cup \{a\} \cup \bigcup_{1 \leq i \leq n}\{a_i\}$ be the set of actions appearing in ua, S, or I.

Let P_1 denote the trace structure $\langle \emptyset, A, \mathbf{pref}\{ua\}\rangle$.

We have that $\mathbf{Ib}(S \cup \{P_1\}) = \emptyset$, because:

- Let $t' \in \mathcal{U}^*$ such that $t'_A = ua$. We have $ua \in \mathbf{Lg}P_1 \cap \overline{\mathbf{Ib}S}$, thus $t' \in \mathbf{Lg}P_1 \cap \overline{\mathbf{Ib}S}$ and, by Lemma 2 (a), $t' \in \overline{\mathbf{Ib}(S \cup \{P_1\})}$;
- For $t' \in \mathcal{U}^*$ such that $t'_A \leq u$, we have $t' \in \mathbf{Lg}(S \cup \{P_1\})$ because $\mathbf{pref}u \subseteq \mathbf{Lg}\|S \cap \mathbf{Lg}P_1 = \mathbf{Lg}\|(S \cup \{P_1\})$.

- For $t' \in \mathcal{U}^*$ such that $t'_A \not\leq ua$, let v be the longest word with the properties that $v \leq t'$ and $v_A \leq ua$. We have $t' = vbw$ for some $b \in A$ and $w \in \mathcal{U}^*$, and $vb \notin \mathbf{Lg}P_1$ and $b \in \mathbf{o}P_1$. Thus $t' \in \mathbf{Ob}(S \cup \{P_1\})$.

In all three cases, $t' \notin \mathbf{Ib}(S \cup \{P_1\})$. Since these cases cover all $\mathcal{U}^*$, we have that $\mathbf{Ib}(S \cup \{P_1\})$ is empty.

However, $\mathbf{Ib}(I \cup \{P_1\})$ is non-empty. Since $ua \in \mathbf{Lg}\|\{P_1\}$ and $ua \in \mathbf{Ib}I$, we have, by Lemma 2 (b), that $ua \in \mathbf{Ib}(I \cup \{P_1\})$.

We obtain that $S \not\sqsubseteq_{stn} I$, in contradiction to the hypothesis. Thus, the assumption that $\mathbf{Ib}I - \mathbf{Ib}S$ is non-empty is false. We conclude that $\mathbf{Ib}I \subseteq \mathbf{Ib}S$.

Now suppose $\exists t \in \mathbf{Ob}S - \mathbf{Ob}I$. As we did above, we construct a 'test' network that invalidates $S \sqsubseteq_{stn} I$. Let $t = uav$ with $u \in \mathbf{Lg}\|S$, $a \in \mathcal{U}$, and $ua \notin \mathbf{Lg}\|S$. Let $u = a_1 \ldots a_n$, with $a_1, \ldots, a_n \in \mathcal{U}$. Let $A = \mathbf{a}\|S \cup \mathbf{a}\|I \cup \{a\} \cup \bigcup_{1 \leq i \leq n} \{a_i\}$ be the set of actions appearing in ua, S, or I.

Let P_2 denote the trace structure $\langle \{a\}, A - \{a\}, \mathbf{pref}\{u\} \rangle$.

We have that $\mathbf{Ib}(S \cup \{P_1, P_2\}) = \emptyset$, because:

- For $t' \in \mathcal{U}^*$ such that $t'_A \leq u$, we have $t' \in \mathbf{Lg}\|(S \cup \{P_1, P_2\})$ because $\mathbf{pref}u \subseteq \mathbf{Lg}\|S \cap \mathbf{Lg}\|\{P_1, P_2\} = \mathbf{Lg}\|(S \cup \{P_1, P_2\})$.

- For $t' \in \mathcal{U}^*$ such that $t'_A \not\leq u$, let v be the longest word with the properties that $v \leq t'$ and $v_A \leq u$. Note that $\mathbf{Lg}\{P_1, P_2\} = \mathbf{pref}u$. We have $t' = vbw$ for some $b \in A$ and $w \in \mathcal{U}^*$, and $vb \notin \mathbf{Lg}\|\{P_1, P_2\}$ and $b \in \mathbf{o}P_1$. Thus $t' \in \mathbf{Ob}(S \cup \{P_1, P_2\})$.

In either case, $t' \notin \mathbf{Ib}(S \cup \{P_1, P_2\})$. Since these cases cover all $\mathcal{U}^*$, we have that $\mathbf{Ib}(S \cup \{P_1, P_2\})$ is empty.

However, $\mathbf{Ib}(I \cup \{P_1, P_2\})$ is non-empty. Since $ua \in \mathbf{Lg}\|\{P_1\}$ and $ua \in \mathbf{Ib}\{P_2\}$, we have that $t \in \mathbf{Ib}\{P_1, P_2\}$. Since $t \notin \mathbf{Ob}I$, then either $t \in \mathbf{Lg}\|I$ or $t \in \mathbf{Ib}I$.

If $t \in \mathbf{Lg}\|I$, then, by Lemma 2 (b), $t \in \mathbf{Ib}(I \cup \{P_1, P_2\})$. If $t \in \mathbf{Ib}I$, then, by Lemma 2 (c), $t \in \mathbf{Ib}(I \cup \{P_1, P_2\})$.

We obtain that $S \not\sqsubseteq_{stn} I$, in contradiction to the hypothesis. Thus, the assumption that $\mathbf{Ob}S - \mathbf{Ob}I$ is non-empty is false. We conclude that $\mathbf{Ob}S \subseteq \mathbf{Ob}I$.

($\Leftarrow$) Let network N such that $\mathbf{Ib}(S \cup N) = \emptyset$. By Lemma 1, $\mathbf{Ib}S \cap \overline{\mathbf{Ob}N} = \mathbf{Ib}N \cap \overline{\mathbf{Ob}S} = \emptyset$.

$$\begin{array}{lll} \mathbf{Ib}I \subseteq \mathbf{Ib}S & \Rightarrow & \mathbf{Ib}I \cap \overline{\mathbf{Ob}N} \subseteq \mathbf{Ib}S \cap \overline{\mathbf{Ob}N} = \emptyset \\ \mathbf{Ob}S \subseteq \mathbf{Ob}I & \Rightarrow & \mathbf{Ib}N \cap \overline{\mathbf{Ob}I} \subseteq \mathbf{Ib}N \cap \overline{\mathbf{Ob}S} = \emptyset \end{array}$$

By Lemma 1 again, $\mathbf{Ib}(I \cup N) = \emptyset$. ■

THEOREM 1 *For networks M, N, and O,*

(a) $M \sqsubseteq_{stn} M$,
(b) $M \sqsubseteq_{stn} N \ \wedge \ N \sqsubseteq_{stn} O \ \Rightarrow \ M \sqsubseteq_{stn} O$,
(c) $M \sqsubseteq_{stn} N \ \Rightarrow \ M \cup O \sqsubseteq_{stn} N \cup O$.

Proof:
(a) We have $\mathbf{Ob}M \subseteq \mathbf{Ob}M$ and $\mathbf{Ib}M \supseteq \mathbf{Ib}M$.

(b) We have

$$\begin{array}{lll} \mathbf{Ob}M \subseteq \mathbf{Ob}N & \wedge & \mathbf{Ob}N \subseteq \mathbf{Ob}O \Rightarrow \mathbf{Ob}M \subseteq \mathbf{Ob}O \\ \mathbf{Ib}M \supseteq \mathbf{Ib}N & \wedge & \mathbf{Ib}N \supseteq \mathbf{Ib}O \Rightarrow \mathbf{Ib}M \supseteq \mathbf{Ib}O \end{array}$$

(c) For any network O' such that $\mathbf{Ib}(M \cup O \cup O') = \emptyset$, since $M \sqsubseteq_{stn} N$, we have $\mathbf{Ib}(N \cup O \cup O') = \emptyset$. ■

PROPOSITION 4 *For network N, we have*

$$\bigcap_{P \in N} \mathbf{Otp}P \subseteq \mathbf{Otp}\|N.$$

Proof: By the definition of **Otp**, it suffices to show that, 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 $e_N \in \mathbf{otp}\|N$.

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 $(\forall t \leq e_N\ \exists u$ 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. ■

PROPOSITION 5 *For network N, $N \sqsubseteq_{tf} N$.*

Proof: Follows immediately from the definition of $\sqsubseteq_{tf}$. ■

LEMMA 3 *For networks M, N, and O such that $M \sqsubseteq_{stn} N$ and $N \sqsubseteq_{stn} O$, we have*

(a) $\mathbf{Lg}\|M \cap \mathbf{Lg}\|O \subseteq \mathbf{Lg}\|N$
(b) $\mathbf{limLg}\|M \cap \mathbf{limLg}\|O \subseteq \mathbf{limLg}\|N$

Proof:
(a) Suppose $\exists t \in \mathbf{Lg}\|M \cap \mathbf{Lg}\|O - \mathbf{Lg}\|N$. By Proposition 1 (a), there are two cases for t.

- If $t \in \mathbf{Ib}\|N$, then $\mathbf{Ib}\|N \not\subseteq \mathbf{Ib}\|M$ and thus $M \not\sqsubseteq_{stn} N$.
- If $t \in \mathbf{Ob}\|N$, then $\mathbf{Ob}\|N \not\subseteq \mathbf{Ob}\|O$ and thus $N \not\sqsubseteq_{stn} O$.

In both cases, the hypothesis is contradicted. Thus the assumption that $\exists t \in \mathbf{Lg}\|M \cap \mathbf{Lg}\|O - \mathbf{Lg}\|N$ is false, and we have $\mathbf{Lg}\|M \cap \mathbf{Lg}\|O \subseteq \mathbf{Lg}\|N$.
(b) Let $e \in \mathbf{limLg}\|M \cap \mathbf{limLg}\|O$, i.e., $\forall t \leq e : t \in \mathbf{Lg}\|M \wedge t \in \mathbf{Lg}\|O$. By part (a) of this Lemma, $\forall t \leq e : t \in \mathbf{Lg}\|N$, thus $e \in \mathbf{limLg}\|N$. ■

THEOREM 2 *For networks M, N and O,*

$$M \sqsubseteq_{tf} N \ \wedge\ N \sqsubseteq_{tf} O \ \wedge\ M \sqsubseteq_{stn} N \ \wedge\ N \sqsubseteq_{stn} O \Rightarrow M \sqsubseteq_{tf} O\,.$$

Proof: We show that $\mathbf{Otp}\|O \cap \mathbf{limLg}\|M \subseteq \mathbf{Otp}\|M$.

Let $e \in \mathbf{Otp}\|O \cap \mathbf{limLg}\|M$. Since $\mathbf{Otp}\|O \subseteq \mathbf{limLg}\|O$, we have, by Lemma 3 (b), that $e \in \mathbf{limLg}\|N$. Since $N \sqsubseteq_{tf} O$, $e \in \mathbf{Otp}\|O$, and $e \in \mathbf{limLg}\|N$, we have $e \in \mathbf{Otp}\|N$. Since $M \sqsubseteq_{tf} N$, $e \in \mathbf{Otp}\|N$, and $e \in \mathbf{limLg}\|M$, we have $e \in \mathbf{Otp}\|M$. ■

LEMMA 4 *For networks M and N,*

(a) *if $M \sqsubseteq_{stn} N$ and $M \sqsubseteq_{oc} N$*
then $\mathbf{Otp}\|N \cap \mathbf{limLg}\|M \subseteq \bigcap_{P \in N} \mathbf{Otp}P$

(b) *if $\mathbf{Ib}N = \emptyset$ and $(\forall P, Q \in N : \mathbf{o}P \cap \mathbf{o}Q = \emptyset)$*
then $\mathbf{Otp}\|N \cap \mathbf{limLg}\|M \subseteq \bigcap_{P \in N} \mathbf{Otp}P$

Proof:

(a) Let $e \in \mathbf{Otp}\|N \cap \mathbf{limLg}\|M$, and let $P \in N$. We show that $e \in \mathbf{Otp}P$.

Let $a \in \mathbf{ren}_{\mathbf{lg}P} e_P \cap \mathbf{o}P$ arbitrary. By the definition of **ren**, we have $\forall t \leq e : \exists u : t \leq u \leq e \wedge u_P a \in \mathbf{lg}P \wedge a \in \mathbf{o}P$.

We show that, for u and a as above, we have $ua \in \mathbf{Lg}\|N$.

Since $e \in \mathbf{Otp}\|N \subseteq \mathbf{limLg}\|N$, we have $u \in \mathbf{Lg}\|N$. Pick an arbitrary $Q \in N$. If $Q = P$, then $ua \in \mathbf{Lg}Q$. If $Q \neq P$, then by $M \sqsubseteq_{oc} N$ we have $\mathbf{o}Q \cap \mathbf{o}P = \emptyset$ and thus $a \notin \mathbf{o}Q$. It follows that $ua \notin \mathbf{Ob}N$.

Now suppose that $ua \in \mathbf{Ib}N$. Since $M \sqsubseteq_{stn} N$, $ua \in \mathbf{Ib}M$ as well. Since $e \in \mathbf{limLg}\|M$ and $u \leq e$, we have $u \in \mathbf{Lg}\|M$. Since $ua \in \mathbf{Ib}M$, $\exists R \in M : a \in \mathbf{i}R \wedge ua \notin \mathbf{Lg}R$. It follows that $\mathbf{i}R \cap \mathbf{o}P \neq \emptyset$, in contradiction to the hypothesis $M \sqsubseteq_{oc} N$. We conclude that the assumption $ua \in \mathbf{Ib}N$ was false.

Since $ua \notin \mathbf{Ob}N$ and $ua \notin \mathbf{Ib}N$, we have by Proposition 1 (b) that $ua \in \mathbf{Lg}\|N$.

Since $a \in \mathbf{o}P$ and $P \in N$, we also have $a \in \mathbf{o}\|N$.

Thus, $\forall t \leq e : \exists u : t \leq u \leq e \wedge u_N a \in \mathbf{lg}\|N \wedge a \in \mathbf{o}\|N$. Since $e \in \mathbf{Otp}\|N$, we have $a \in \mathbf{rfi}e$. It follows that $e \in \mathbf{Otp}P$. Since P was an arbitrary element of N, we have that $e \in \bigcap_{P \in N} \mathbf{Otp}P$.

(b) The proof proceeds similarly to part (a). Let $e \in \mathbf{Otp}\|N \cap \mathbf{limLg}\|M$, and let $P \in N$. We show that $e \in \mathbf{Otp}P$.

Let $a \in \mathbf{ren}_{\mathbf{lg}P} e_P \cap \mathbf{o}P$ arbitrary. By the definition of **ren**, we have $\forall t \leq e : \exists u : t \leq u \leq e \wedge u_P a \in \mathbf{lg}P \wedge a \in \mathbf{o}P$.

We show that, for u and a as above, we have $ua \in \mathbf{Lg}\|N$.

Since $e \in \mathbf{Otp}\|N \subseteq \mathbf{limLg}\|N$, we have $u \in \mathbf{Lg}\|N$. Pick an arbitrary $Q \in N$. If $Q = P$, then $ua \in \mathbf{Lg}Q$. If $Q \neq P$, then by hypothesis we have $\mathbf{o}Q \cap \mathbf{o}P = \emptyset$ and thus $a \notin \mathbf{o}Q$. It follows that $ua \notin \mathbf{Ob}N$.

Since $ua \notin \mathbf{Ob}N$ and since, by hypothesis, $\mathbf{Ib}N = \emptyset$, and thus $ua \notin \mathbf{Ib}N$, we have by Proposition 1 (b) that $ua \in \mathbf{Lg}\|N$.

Since $a \in \mathbf{o}P$ and $P \in N$, we also have $a \in \mathbf{o}\|N$.

Thus, $\forall t \leq e : \exists u : t \leq u \leq e \wedge u_N a \in \mathbf{lg}\|N \wedge a \in \mathbf{o}\|N$. Since $e \in \mathbf{Otp}\|N$, we have $a \in \mathbf{rfi}e$. It follows that $e \in \mathbf{Otp}P$. Since P was an arbitrary element of N, we have that $e \in \bigcap_{P \in N} \mathbf{Otp}P$. ■

THEOREM 3 *For networks M and N such that $M \sqsubseteq_{tf} N$, we have*

(a) *if* $M \sqsubseteq_{stn} N$ *and* $M \cup O \sqsubseteq_{oc} N \cup O$ *then* $M \cup O \sqsubseteq_{tf} N \cup O$
(b) *if* $\mathbf{lb}N = \emptyset$ *and* $(\forall P, Q \in N : \mathbf{o}P \cap \mathbf{o}Q = \emptyset)$ *then* $M \cup O \sqsubseteq_{tf} N \cup O$

Proof:
(a)

$$
\begin{array}{ll}
 & \mathbf{Otp}\|(N \cup O) \cap \mathbf{limLg}\|(M \cup O) \qquad \text{(by Lemma 4 (a))} \\
\subseteq & \bigcap_{P \in N \cup O} \mathbf{Otp}P \cap \mathbf{limLg}\|M \\
= & \bigcap_{P \in N} \mathbf{Otp}P \cap \bigcap_{P \in O} \mathbf{Otp}P \cap \mathbf{limLg}\|M \\
\subseteq & \mathbf{Otp}\|N \cap \mathbf{Otp}\|O \cap \mathbf{limLg}\|M \qquad \text{(by Proposition 4)} \\
\subseteq & \mathbf{Otp}\|M \cap \mathbf{Otp}\|O \qquad \text{(since } M \sqsubseteq_{tf} N\text{)} \\
\subseteq & \mathbf{Otp}\|\{\|M, \|O\} \qquad \text{(by Proposition 4)} \\
\subseteq & \mathbf{Otp}\|(M \cup O) \qquad \text{(because } \|\{\|M, \|O\} = \|(M \cup O)\text{)}
\end{array}
$$

(b) Similar to part (a), except Lemma 4 (b) is used instead of Lemma 4 (a). ■

Appendix B

Behavior Automaton Proofs

PROPOSITION 6 *For behavior automaton* A *and sequence* e *in* $\mathbf{lim\,lg\,tr}A$, $A \diamond e$ *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 7 *For behavior automaton A and knot G in A, there exists a sequence e in* $\mathbf{lim\,lg\,tr}A$ *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 $e = u^{01}(a^1u^{12} \cdots a^nu^{n1})^\omega$. Note that, since $n > 0$, the string $a^1u^{12} \cdots a^nu^{n1}$ 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

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

(Note that $l + 1 = l \bmod n + 1$ because $l < n$, and that the string $a^1u^{12} \cdots a^l u^{l\,l+1}$ 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*

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

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

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

($\Rightarrow$) This part is an immediate consequence of Lemma 5.
($\Leftarrow$) We use structural induction over t.
– Basis: $t = \epsilon$. We have $\epsilon \in \mathbf{lg}(A\|B)$.
– 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 4 *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 8 is proven immediately after the following lemma.

LEMMA 7 *For behavior automata A and B and sequence e in* $\mathbf{lim\,lg\,tr}(A\|B)$, $((A\|B) \diamond e)_A = A \diamond e_A$ *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 $\diamond$ 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 $e \in \mathbf{lim\,lg\,tr}(A\|B)$, 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':

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

Thus $\exists o' \in \mathbf{st}((A\|B)\diamond e)$ 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 $e \in \mathbf{lim\,lg\,tr}(A\|B)$, 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 $vd \leq e$:

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

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 8 *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 7, there exists a sequence e in $\mathbf{lim\,lg}(A\|B)$ such that $(A\|B) \diamond e = G$.
By Lemma 7, we have that $A \diamond e_A = ((A\|B) \diamond e)_A$.
By Proposition 6, 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:

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

• 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, thus $a \in \mathbf{ren}_{\mathbf{lg}_A} e$. ■

LEMMA 9 *For behavior automaton A, knot G in A, and sequence e in $\mathbf{lim\,lg\,tr}A$ 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 5 *For behavior automata S and I, I is traplock-free for S iff $\{\mathbf{tr}S\} \sqsubseteq_{tf} \{\mathbf{tr}I\}$.*

Proof:
($\Rightarrow$) We prove that $\{\mathbf{tr}S\}\ \sqsubseteq_{tf}\ \{\mathbf{tr}I\}$. Let $e \in \mathbf{lim\,lg}(\mathbf{tr}S \| \mathbf{tr}I)$ such that $e_I \in \mathbf{otp\,tr}I$. By Theorem 4, $e \in \mathbf{lim\,lg\,tr}(S \| I)$. Let $G = (S \| I) \diamond e$; by Proposition 6, G is a knot in $S \| I$. By Proposition 8, 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 $e_S \in \mathbf{otp\,tr}S$.
($\Leftarrow$) 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 7 and Theorem 4, there exists a sequence e in $\mathbf{lim\,lg\,tr}(S \| I) = \mathbf{lim\,lg}(\mathbf{tr}S \| \mathbf{tr}I)$ 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 6, we conclude that G_S is an output trap in S. ■

Notes

1. This is only one of many possible behaviors one can associate with a XOR gate. It is the unrestricted behavior [3] in a 'single-winner' model (GSW), assuming inertial delays.

References

1. B. Alpern, F. B. Schneider, "Defining Liveness," *Information Processing Letters*, 21:181–185, 1985.
2. D. Black, "On the Existence of Delay-insensitive Fair Arbiters: Trace Theory and its Limitations," *Distributed Computing*, 1:205–225, 1986.
3. J. A. Brzozowski, C-J. H. Seger, *Asynchronous Circuits*, Springer Verlag, 1995.
4. E. Chang, Z. Manna, A. Pnueli, "The Safety-Progress Classification," Report No. STAN-CS-92-1408, Stanford University, Dept. of Computer Science, 1992.
5. D. Dill, E. Clarke, "Automatic Verification of Asynchronous Circuits Using Temporal Logic," in H. Fuchs, editor, *1985 Chapel Hill Conf. on VLSI*, Computer Science Press, 1985, pp. 127–143.
6. D. Dill, "Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits," An ACM Distinguished Dissertation, MIT Press, 1989.

7. J. C. Ebergen, "Translating programs into delay-insensitive circuits," CWI Tract 56, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1989.
8. J. C. Ebergen, "A Formal Approach to Designing Delay-Insensitive Circuits," *Distributed Computing*, 5:107–119, 1991.
9. N. Francez, *Fairness*, Springer-Verlag, 1986.
10. 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*, 13:1309–1318, 1994.
11. C. A. R. Hoare,*Communicating Sequential Processes*, Prentice-Hall, 1985.
12. B. Jonson, "Modular Verification of Asynchronous Networks," *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing*, 1987, pp. 137–151.
13. M. B. Josephs, "Receptive Process Theory," *Acta Informatica*, 29:17–31, 1992.
14. L. Lamport, N. Lynch, "Distributed Computing: Models and Methods," in J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B, Formal Methods and Semantics*, the MIT Press - Elsevier, 1990, pp. 1159–1196.
15. N. Lynch, M. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing*, 1987, pp. 137–151.
16. R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.
17. R. Negulescu and J. A. Brzozowski, "Relative Liveness: From Intuition to Automated Verification," *Proceedings of the Second Working Conference on Asynchronous Design Methodologies*, South Bank University, London, UK, IEEE Computer Society Press, May 1995, pp. 108–117.
18. R. Negulescu and J. A. Brzozowski, "Relative Liveness: From Intuition to Automated Verification," Research Report CS-95-32, Department of Computer Science, University of Waterloo, ON, Canada, July 1995.
19. M. Rem, J. L. A. van de Snepscheut, J. T. Udding, "Trace Theory and the Definition of Hierarchical Components," in R. Bryant, editor, *Third CalTech Conference on Very Large Scale Integration*, Computer Science Press, Inc., 1983, pp. 225–239.
20. 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.
21. J. Staunstrup, *A Formal Approach to Hardware Design*, Kluwer Academic Publishers, Boston/Dordrecht/London, 1994.
22. W. Thomas, "Automata on Infinite Objects," In J. van Leeuwen,editor, *Handbook of Theoretical Computer Science, vol. B, Formal Methods and Semantics*, the MIT Press - Elsevier, 1990, pp. 135–191.
23. J. T. Udding, "A Formal Model for Defining and Classifying Delay-Insensitive Circuits and Systems," *Distributed Computing*, 1:197–204, 1986.
24. J. T. Udding, "Classification and Composition of Delay-Insensitive Circuits," PhD Thesis, Department of Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1984.
25. T. Verhoeff. *A Theory of Delay-Insensitive Systems*, PhD Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1994.