ELSEVIER

Information Processing Letters 59 (1996) 245-250

Information Processing Letters

A note on fairness in I/O automata [1]

Judi Romijn [2], Frits Vaandrager *

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Received 24 January 1996; revised 24 June 1996
Communicated by F.B. Schneider

Abstract

Notions of weak and strong fairness are studied in the setting of the I/O automaton model of Lynch and Tuttle. The concept of a *fair I/O automaton* is introduced and it is shown that a fair I/O automaton paired with the set of its fair executions is a live I/O automaton provided that (1) in each reachable state at most countably many fairness sets are enabled, and (2) input actions cannot disable strong fairness sets. This result, which generalizes previous results known from the literature, was needed to solve a problem posed by Broy and Lamport for the Dagstuhl Workshop on Reactive Systems.

Keywords: Concurrency; I/O automata; Weak fairness; Strong fairness; Liveness

1. Introduction

Many specification formalisms for reactive systems incorporate notions of weak and strong fairness (see, for instance, [5-8]). Informally, the requirement of weak fairness disallows executions in which certain sets of transitions are continually enabled but not taken beyond a certain point, whereas the requirement of strong fairness disallows executions in which certain sets of transitions are enabled infinitely often but taken only finitely many times. A natural criterion that any acceptable notion of fairness should satisfy is that it induces liveness properties in the sense of [2]: it should be possible to extend every finite execution to a fair one. Several authors have observed that weak and strong fairness induce liveness properties if the number of fairness sets (sets of transitions for which fairness is required) is countable [7,1]. If this number is uncountable then one does not obtain liveness properties in general: since in a transition system each execution contains at most a countable number of transitions, it is impossible to give fair turns to uncountably many fairness sets.

In most practical cases, the restriction to a countable number of fairness sets is unproblematic. However, there are classes of applications where this restriction cannot be made. A nice example here is the RPC-Memory specification problem proposed by Broy and Lamport [3] for the Dagstuhl Workshop on Reactive Systems. In this problem, there is a set of pro-

* Corresponding author. Current address: Computing Science Institute, University of Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands. Email: frits.vaandrager@cs.kun.nl. Supported by the HCM network EXPRESS.

[1] The results reported in this paper have been obtained as part of the research project "Specification, Testing and Verification of Software for Technical Applications", which is being carried out by the Stichting Mathematisch Centrum for Philips Research Laboratories under Contract RWC-061-PS-950006-ps.

[2] Email: judi@cwi.nl.

PII S0020-0190(96)00122-6

cesses that can concurrently issue procedure calls to a memory component, which responds to these calls by issuing returns. Because there are no constraints on the number of processes and each call should eventually lead to a corresponding return, it is impossible to specify the required liveness properties using only a bounded number of fairness sets. Essentially, the main result of this note is that liveness is also ensured if one does not impose a global constraint on the number of fairness sets, but instead assumes that in each reachable state only a countable number of fairness sets is enabled. The latter restriction applies to the Dagstuhl example since in each reachable state the number of outstanding calls is finite. The key argument in our proof is not difficult, but distinctly different from the arguments used in the proofs of [1,7].

We have stated our results in terms of the I/O automaton model [7,4], since the first author needed this for her I/O automata solution to the Dagstuhl problem [9]. We propose a model of *fair I/O automata*, which is a generalization of the original I/O automaton model of [7]. Our main result is that under certain assumptions fair I/O automata can be viewed as a special case of the *live I/O automata* of [4], another generalization of the original model. Roughly speaking, this result says that each finite execution can be extended to a fair one independently of the inputs provided by the environment. The notion of a live I/O automaton is very general but its definition is complex and cumbersome to use: in order to prove that a certain structure is a live I/O automaton one has to exhibit a winning strategy in an infinite two-player game. Since it appears that all liveness properties that one needs in practice can be specified using weak and strong fairness properties only [5,6,8] and since it is usually trivial to check that a structure is a fair I/O automaton, we think that there will be many situations where, after one has described a system as a fair I/O automaton, our result provides one with a live I/O automaton description almost for free.

The outline of this article is as follows. In Section 2, we introduce fair I/O automata. In Section 3 we prove that a fair I/O automaton paired with the set of its fair executions is a live I/O automaton provided that (1) in each reachable state at most countably many fairness sets are enabled, and (2) input actions cannot disable strong fairness sets. In Section 4, we define a composition operation on fair I/O automata and show that this operation is compatible with the composition operation on live I/O automata defined in [4]. The Appendix recalls the basic notions of safe and live I/O automata as defined in [7,4].

2. Definitions

In this section we define the model of *fair I/O automata*, which is a generalization of the original I/O automaton model of [7]: whereas the I/O automata of [7] only allow for weak fairness, fair I/O automata permit both weak and strong fairness.

Fair I/O automata. A *fair I/O automaton* A is a triple consisting of

- a safe I/O automaton $\mathit{safe}(A)$, and
- two subsets of $\mathit{local}(\mathit{safe}(A))$, sets $\mathit{wfair}(A)$ and $\mathit{sfair}(A)$, called the *weak fairness sets* and *strong fairness sets*, respectively.

In the rest of this note we write $\mathit{local}(A)$ for $\mathit{local}(\mathit{safe}(A))$, $\mathit{steps}(A)$ for $\mathit{steps}(\mathit{safe}(A))$, etc. Also, we fix a fair I/O automaton A.

Enabling. Let U be a set of actions of A. Then U is *enabled* in a state s if and only if an action from U is enabled in s. Set U is *input resistant* if and only if, for each pair of reachable states s, s' and for each input action a,

$$s \text{ enables } U \wedge s \xrightarrow{a} s' \quad \Rightarrow \quad s' \text{ enables } U.$$

So once U is enabled, it can only be disabled by the occurrence of a locally controlled action.

Fair executions. An execution α of A is *weakly fair* iff the following conditions hold for each $W \in \mathit{wfair}(A)$:

(1) If α is finite then W is not enabled in the last state of α.

(2) If α is infinite then either α contains infinitely many occurrences of actions from W, or α contains infinitely many occurrences of states in which W is not enabled.

Execution α is *strongly fair* iff the following conditions hold for each $S \in \mathit{sfair}(A)$:

(1) If α is finite then S is not enabled in the last state of α.

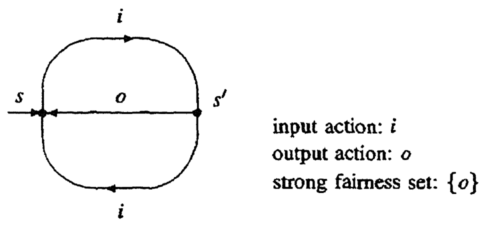

Fig. 1. A fair I/O automaton that is not live.

(2) If α is infinite then either α contains infinitely many occurrences of actions from S, or α contains only finitely many occurrences of states in which S is enabled.

Execution α is *fair* iff it is both weakly and strongly fair. In a fair execution each weak fairness set gets turns if enabled continuously, and each strong fairness set gets turns if enabled infinitely many times. We write *fairexecs*(A) for the set of fair executions of A.

3. Main result

In [4], live I/O automata are introduced as a generalization of the I/O automata of [7] with general liveness properties (see also the Appendix). Our main result, stated below, says that, if fair I/O automata A satisfies two conditions then the pair $(\mathit{safe}(A), \mathit{fairexecs}(A))$ is a live I/O automaton. The first condition states that in each reachable state at most countably many weak and strong fairness sets are enabled. This cardinality assumption allows us to define, via a diagonalization construction, a strategy for the I/O automaton that gives fair turns to each fairness set. The second condition states that all strong fairness sets are input resistant. This technical assumption excludes situations where the environment gives turns to the system only when some strong fairness set is not enabled. As an example, consider the fair I/O automaton of Fig. 1. In this I/O automaton the strong fairness set $\{o\}$ is not input resistant. As a result the I/O automaton is not live: for each strategy ρ, the outcome $\mathcal{O}_\rho(s, \lambda\, i\, i\, \lambda\, i\, i\, \lambda \cdots)$ equals the unfair execution $s\, i\, s'\, i\, s\, i\, s' \cdots$.

Theorem 1. *Suppose that fair I/O automaton A satisfies the following conditions:* (1) *each reachable state of A enables at most countably many sets in* $\mathit{wfair}(A) \cup \mathit{sfair}(A)$, *and* (2) *each set in* $\mathit{sfair}(A)$ *is input resistant. Then* $\mathit{live}(A) \triangleq (\mathit{safe}(A), \mathit{fairexecs}(A))$ *is a live I/O automaton.*

Proof. With each finite execution α we associate an infinite two-dimensional array $\mathcal{A}_\alpha$ of weak and strong fairness sets. The array contains all the weak or strong fairness sets that are enabled at some point in execution α but from which no action has been executed in the subsequent part of α. We will use array $\mathcal{A}_\alpha$ to define a strategy that treats each fairness set in a fair manner and thus establishes that *live*(A) is a live I/O automaton. The array is defined by induction on the length of α:

- If α consists of a single state s, then $\mathcal{A}_\alpha$ is constructed by filling the first row with the sets in *wfair*(A) and *sfair*(A) that are enabled in s. While filling, the sets are alternatingly taken from *wfair*(A) and *sfair*(A). Remaining positions are filled with the symbol ■.
  If s enables 6 weak fairness sets and 2 strong fairness sets, then $\mathcal{A}_\alpha$ might look like this:

	1	2	3	4	5	6	7	8	9	$\cdots$
1	W_{1_1}	S_{1_1}	W_{1_2}	S_{1_2}	W_{1_3}	W_{1_4}	W_{1_5}	W_{1_6}	■	$\cdots$
2	■	■	■	■	■	■	■	■	■	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

  Note that by Condition (1) we are able to squeeze all the enabled sets in a single row.
- If α contains $n > 1$ states and is of the form $\alpha'\, a\, s$, then $\mathcal{A}_\alpha$ is constructed from $\mathcal{A}_{\alpha'}$ by replacing each fairness set that contains action a by ■, and filling the nth row with the sets in *wfair*(A) and *sfair*(A) that are enabled in s, as in the previous case.
  The array for an execution α with 4 states might look like this:

	1	2	3	4	5	6	7	8	$\cdots$
1	W_{1_1}	S_{1_1}	■	S_{1_2}	■	■	W_{1_5}	■	$\cdots$
2	■	S_{2_1}	W_{2_2}	S_{2_2}	W_{2_3}	S_{2_3}	■	■	$\cdots$
3	S_{3_1}	■	■	S_{3_4}	S_{3_5}	S_{3_6}	■	■	$\cdots$
4	W_{4_1}	S_{4_1}	W_{4_2}	S_{4_2}	W_{4_3}	S_{4_3}	W_{4_4}	S_{4_4}	$\cdots$
5	■	■	■	■	■	■	■	■	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Let $\rho = (g, f)$ be any strategy defined on *safe*(A) that satisfies the following conditions:

(1) If $f(\alpha) = \perp$ then the last state of α enables no set in *wfair*$(A) \cup$ *sfair*(A).

(2) If $f(\alpha) = (a, s)$ then the last state of α enables a set in *wfair*$(A) \cup$ *sfair*(A), and a is member of the first set U that is enabled in the last state of α and that occurs in the sequence

$$\begin{aligned}\Omega(\alpha) \triangleq\ & \mathcal{A}_\alpha[1,1] \\ & \mathcal{A}_\alpha[1,2]\ \mathcal{A}_\alpha[2,1] \\ & \mathcal{A}_\alpha[1,3]\ \mathcal{A}_\alpha[2,2]\ \mathcal{A}_\alpha[3,1] \\ & \mathcal{A}_\alpha[1,4]\ \mathcal{A}_\alpha[2,3]\ \mathcal{A}_\alpha[3,2]\ \mathcal{A}_\alpha[4,1] \\ & \vdots\end{aligned}$$

Note that a strategy ρ satisfying these properties exists since by construction the array $\mathcal{A}_\alpha$ contains at least all the weak and strong fairness sets that are enabled in the last state of α, and sequence $\Omega(\alpha)$ enumerates all elements of $\mathcal{A}_\alpha$.

We show that *live*(A) is a live I/O automaton by proving that the outcome $\alpha' = \mathcal{O}_\rho(\alpha, \mathcal{I})$ is fair for each finite execution α and each environment sequence $\mathcal{I}$.

Assume that α' is a finite execution. Then $\mathcal{I}$ contains only finitely many input actions and, for s the last state of α', $f(\alpha') = \perp$. Therefore, by the first assumption about strategy ρ, the last state of α' enables no set in *wfair*(A) or *sfair*(A). Hence α' is fair.

Thus we may assume that α' is infinite. We prove that α' is fair by contradiction. Suppose α' is not fair. We distinguish between two cases:

(1) α' is not strongly fair.
Then some strong fairness set S is enabled in an infinite number of states of α' and α' contains only finitely many occurrences of actions in S. Since S is input resistant, it is enabled in an infinite number of states in which a system move is allowed by $\mathcal{I}$. From the definition of strategy ρ it follows that S is enabled in an infinite number of states in which a locally controlled action occurs. Since there are only finitely many occurrences of actions in S, there is a state in α' after which no action in S occurs. Nevertheless, there is a subsequent state of α', say the ith state, in which S is enabled. Therefore, there is a position $[i, j]$ such that, if α_k is the finite prefix of α' with k states, $\mathcal{A}_{\alpha_k}[i, j] = S$, for all $k \geqslant i$. Let $l = i + j - 1$. Then, for each $n \geqslant l$, each position preceding $[i, j]$ in the strategy's sequence that is filled with $\blacksquare$ in the array $\mathcal{A}_{\alpha_n}$, is also filled with $\blacksquare$ in any array $\mathcal{A}_{\alpha_m}$ with $m > n$. Each locally controlled action that occurs after the lth state from a state that enables S causes a fairness set at a position preceding $[i, j]$ in the strategy's sequence to be replaced by $\blacksquare$ in the array. This happens infinitely many times. But this is a contradiction since the number of preceding positions is finite.

(2) α' is not weakly fair.
Then some weak fairness set W is enabled in all states of an infinite suffix of α' with only finitely many occurrences of actions from W.
By an argument that is almost identical to the one used in the previous case we arrive at a contradiction.

Hence α' is fair and we may conclude that *live*(A) is a live I/O automaton. $\square$

4. Composition

Building on the work of [7,4], there is an obvious way to define composition of fair I/O automata.

We say that two fair I/O automata A_1 and A_2 are *compatible* if *safe*(A_1) and *safe*(A_2) are compatible. Suppose that A_1 and A_2 are compatible fair I/O automata. Then the *composition* $A_1 \| A_2$ is the fair I/O automaton A given by

- *safe*$(A) =$ *safe*$(A_1) \|$ *safe*(A_2),
- *wfair*$(A) =$ *wfair*$(A_1) \cup$ *wfair*(A_2) and *sfair*$(A) =$ *sfair*$(A_1) \cup$ *sfair*(A_2).

Thus we simply compose the underlying safe I/O automata and take the unions of the weak and strong fairness sets. The following theorem, which is easy to prove, states that the above composition operation for fair I/O automata is compatible with the composition operation for live I/O automata of [4].

Theorem 2. *Suppose that A_1 and A_2 are compatible fair I/O automata. Then*

$$live(A_1 \| A_2) = live(A_1) \| live(A_2).$$

Acknowledgements

We thank Ed Brinksma, Leslie Lamport, Roberto Segala and Jan Springintveld for useful comments on this paper.

Appendix. Safe and live I/O automata

In this appendix we review some basic definitions from [7,4].

Safe I/O automata. A *safe I/O automaton B* consists of the following components:

- A set $states(B)$ of *states* (possibly infinite).
- A nonempty set $start(B) \subseteq states(B)$ of *start states*.
- A set $acts(B)$ of *actions*, partitioned into three sets $in(B)$, $int(B)$ and $out(B)$ of *input*, *internal* and *output* actions, respectively. Actions in $local(B) \triangleq out(B) \cup int(B)$ are called *locally controlled*.
- A set $steps(B) \subseteq states(B) \times acts(B) \times states(B)$ of *transitions*, with the property that for every state s and input action $a \in in(B)$ there is a transition $(s, a, s') \in steps(B)$.

We let $s, s', \ldots$ range over states, and $a, \ldots$ over actions. We write $s \xrightarrow{a}_B s'$, or just $s \xrightarrow{a} s'$ if B is clear from the context, as a shorthand for $(s, a, s') \in steps(B)$.

Enabling. An action a is *enabled* in a state s iff $s \xrightarrow{a} s'$ for some s'. Since every input action is enabled in every state, safe I/O automata are said to be *input enabled*. The intuition behind the input-enabling condition is that input actions are under control of the environment and that a system that is modeled by a safe I/O automaton cannot prevent its environment from doing these actions.

Executions. An *execution fragment* of a safe I/O automaton B is a finite or infinite alternating sequence $s_0 a_1 s_1 a_2 s_2 \cdots$ of states and actions of B, beginning with a state, and if it is finite also ending with a state, such that for all i, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. An *execution* is an execution fragment that begins with a start state. We write $execs^*(B)$ for the set of finite executions of B, and $execs(B)$ for the set of all executions of B. A state s of B is *reachable* iff it is the last state of some finite execution of B.

Live I/O automata. Intuitively, a *live I/O automaton* is a pair of a safe I/O automaton B and a set L of executions of B such that B can always generate an execution in L independently of the input provided by its environment. Formally, live I/O automata can be defined in terms of a two person game between a system player and an environment player. The goal of the system player is to construct an execution in L, and the goal of the environment player is to prevent this. The pair (B, L) is a live I/O automaton iff there exists a strategy by which the system player can always win the game, irrespective of the behavior of the environment player.

A *strategy* defined on a safe I/O automaton B is a pair of functions (g, f) where $g : execs^*(B) \times in(B) \to states(B)$ and $f : execs^*(B) \to (local(B) \times states(B)) \cup \{\bot\}$ such that

(1) $g(\alpha, a) = s \Rightarrow \alpha\, a\, s \in execs^*(B)$,

(2) $f(\alpha) = (a, s) \Rightarrow \alpha\, a\, s \in execs^*(B)$.

An *environment sequence* for B is an infinite sequence of symbols from $in(B) \cup \{\lambda\}$ with infinitely many occurrences of λ. The symbol λ represents the points at which the system is allowed to move. The occurrence of infinitely many λ symbols in an environment sequence guarantees that each environment move consists of only finitely many input actions.

Let $\rho = (g, f)$ be a strategy defined on B, $\mathcal{I} = a_1 a_2 a_3 \cdots$ an environment sequence for B, and α a finite execution of B. Then the *outcome* $\mathcal{O}_\rho(\alpha, \mathcal{I})$ is the limit of the sequence $(\alpha_i)_{i \geqslant 0}$ of finite executions defined inductively by

- $\alpha_0 = \alpha$.
- If $i > 0$ then
  (1) $a_i = \lambda \wedge f(\alpha_{i-1}) = (a, s) \Rightarrow \alpha_i = \alpha_{i-1}\, a\, s$,
  (2) $a_i = \lambda \wedge f(\alpha_{i-1}) = \bot \Rightarrow \alpha_i = \alpha_{i-1}$,
  (3) $a_i \in in(B) \wedge g(\alpha_{i-1}, a_i) = s \Rightarrow \alpha_i = \alpha_{i-1} a_i s$.

A *live I/O automaton* is a pair (B, L) with B a safe I/O automaton and $L \subseteq execs(B)$ such that there exists a strategy ρ defined on B with for any finite execution α of B and any environment sequence $\mathcal{I}$ for B, $\mathcal{O}_\rho(\alpha, \mathcal{I}) \in L$.

Composition. Two safe I/O automata B_1 and B_2 are *compatible* iff $out(B_1) \cap out(B_2) = \emptyset$, $int(B_1) \cap acts(B_2) = \emptyset$, and $int(B_2) \cap acts(B_1) = \emptyset$. The *com-*

position $B_1 \| B_2$ of compatible safe I/O automata B_1 and B_2 is the safe I/O automaton B defined by

- $states(B) = states(B_1) \times states(B_2)$,
- $start(B) = start(B_1) \times start(B_2)$,
- $acts(B) = in(B) \cup out(B) \cup int(B)$, where

$$\begin{aligned} in(B) &= (in(B_1) \cup in(B_2)) \\ &\quad - (out(B_1) \cup out(B_2)), \\ out(B) &= out(B_1) \cup out(B_2), \\ int(B) &= int(B_1) \cup int(B_2), \end{aligned}$$

- $steps(B)$ is the set of triples $((s_1, s_2), a, (s'_1, s'_2))$ in $states(B) \times acts(B) \times states(B)$ such that, for $i \in \{1,2\}$, if $a \in acts(B_i)$ then $s_i \xrightarrow{a}_{B_i} s'_i$ else $s_i = s'_i$.

Let B_1, B_2 be safe I/O automata, $L_1 \subseteq execs(B_1)$ and $L_2 \subseteq execs(B_2)$. The pairs (B_1, L_1) and (B_2, L_2) are *compatible* iff B_1 and B_2 are compatible. The *composition* $(B_1, L_1) \| (B_2, L_2)$ of two compatible pairs (B_1, L_1) and (B_2, L_2) is the pair (B, L) defined by

- $B = B_1 \| B_2$,
- $L = \{\alpha \in execs(B) \mid \alpha \lceil B_1 \in L_1 \text{ and } \alpha \lceil B_2 \in L_2\}$. Here $\alpha \lceil B_i$ is obtained by projecting each state in α on the ith component and by removing each action that is not in $acts(B_i)$ together with the state that follows it.

A major result of [4] is that the class of live I/O automata is closed under composition.

Theorem 3. *Let* (B_1, L_1) *and* (B_2, L_2) *be compatible live I/O automata. Then* $(B_1, L_1) \| (B_2, L_2)$ *is a live I/O automaton.*

References

[1] M. Abadi and L. Lamport, An old-fashioned recipe for real time, *ACM Trans. Programming Languages Systems* **16** (5) (1994) 1543–1571.

[2] B. Alpern and F.B. Schneider, Defining liveness, *Inform. Process. Lett.* **21** (1985) 181–185.

[3] M. Broy and L. Lamport, Specification problem, in: *Proc. Dagstuhl-Seminar on the RPC-Memory Specification Problem*, Lecture Notes in Computer Science (Springer, Berlin, 1996), to appear.

[4] R. Gawlick, R. Segala, J.F. Søgaard-Andersen and N. Lynch, Liveness in timed and untimed systems, in: S. Abiteboul and E. Shamir, eds., *Proc. 21th ICALP*, Lecture Notes in Computer Science **820** (Springer, Berlin, 1994); a full version is available as MIT Tech. Rept. MIT/LCS/TR-587.

[5] B. Jonsson, Compositional specification and verification of distributed systems, *ACM Trans. Programming Languages Systems* **16** (2) (19940 259-303.

[6] L. Lamport, The temporal logic of actions, *ACM Trans. Programming Languages Systems* **16** (3) (1994) 872–923.

[7] N.A. Lynch and M.R. Tuttle, Hierarchical correctness proofs for distributed algorithms, in: *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing* (1987) 137–151; A full version is available as MIT Tech. Rept. MIT/LCS/TR-387.

[8] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification* (Springer, Berlin, 1992).

[9] J.M.T. Romijn, Tackling the RPC-Memory specification problem with I/O automata, in: *Proc. Dagstuhl-Seminar on the RPC-Memory Specification Problem*, Lecture Notes in Computer Science (Springer, Berlin, 1996), to appear.