INFORMATICA, 2015, Vol. 26, No. 4, 685–704
© 2015 *Vilnius University*
DOI: http://dx.doi.org/10.15388/Informatica.2015.71

Specifying and Verifying External Behaviour of Fair Input/Output Automata by Using the Temporal Logic of Actions

Tatjana KAPUS
University of Maribor, Faculty of Electrical Engineering and Computer Science
Smetanova ul. 17, SI-2000 Maribor, Slovenia
e-mail: tatjana.kapus@um.si

Received: August 2013; accepted: March 2015

Abstract. Fair input/output (or I/O) automata are a state-machine model for specifying and verifying reactive and concurrent systems. For the verification purposes, one is usually interested only in the sequences of interactions fair I/O automata offer to their environment. These sequences are called fair traces. The usual approach to the verification consists in proving fair trace inclusion between fair I/O automata. This paper presents a simple approach to the specification of fair traces and shows how to establish a fair trace inclusion relation for a pair of fair I/O automata by using the temporal logic of actions.

Key words: formal specification, fair input/output automaton, temporal logic of actions, fair trace, fair trace inclusion.

1. Introduction

Input/output automata (or I/O automata) are a popular set-theoretic model for the specification and analysis of reactive and concurrent systems (Tuttle, 1987; Lynch and Tuttle, 1989). They have, for example, been applied for reasoning about data communications protocols (Søgaard-Andersen *et al.*, 1993), trust management systems (Krukow and Nielsen, 2007; Trček, 2014), Web services (Mitra *et al.*, 2007), and database replication protocols (Armendáriz-Iñigo *et al.*, 2009). An I/O automaton is a state machine in which each state transition is labelled by an action. Originally, no formal language for precise description of I/O automata system models and their correctness properties has been proposed. I/O automata have usually been described by using text and pseudocode. Preconditions and effects on variables introduced to represent automata states were specified for every possible action. The IOA programming language (Garland and Lynch, 2000) formalizes this semi-formal precondition/effect style. It is similar to the usual programming languages in that it has a reach formally defined syntax for the description of systems but cannot directly be used for reasoning about them. The reasoning has to be carried out by using its semantics written in a lower-level language of a theorem prover and in a specific first-order logical language developed by its authors (Garland *et al.*, 2003).

In contrast to IOA and the mentioned lower-level languages, the temporal logic of actions (TLA) (Lamport, 1994) is a logical language which can be used for writing a system specification by directly describing the system's *execution* semantics in a simple way similar to the precondition/effect style. The formal specification as well as reasoning about systems can be carried out in TLA. In Kapus (2002, 2005), we show how TLA can be used for the formal specification of systems based on a model similar to the I/O automaton one, but containing mobility and dynamic creation of components. TLA can be used in a similar way for the ordinary I/O automaton model because the latter can be treated as a special case of the former.

In this paper, we further investigate TLA as a formal language for I/O automata. Despite the term 'actions' in its name, the implicit underlying model of TLA is a simple state machine with no action labels. In Kapus (2002, 2005), we use a special variable for recording the actions in such a way that the TLA specification of a finite-state I/O automaton may give infinitely many reachable states (here, in contrast to the formal definition for TLA, by a 'state' we mean an assignment to a limited number of variables). The aim of this paper are finite-state TLA specifications for finite-state I/O automata, so as to be able to verify them automatically by using TLC, the model checker for specifications based on TLA (TLA Toolbox, 2014).

Usually, only the external behaviour, i.e. the sequences of actions visible to the environment, is of concern in the verification of I/O automata. In the mentioned papers, we do not deal with the problem of how to specify only the external behaviour, and we consider only the specification of safety properties (i.e. 'what may happen'), but not the specification of liveness properties (i.e. 'what must happen'). Besides, we do not consider the verification.

In this paper, we consider so-called fair I/O automata (Müller, 1998), which allow the specification of liveness, and propose a simple way to specify the external behaviour of fair I/O automata, i.e. so-called traces and fair traces, by using TLA. The usual approach to the verification of fair I/O automata consists in proving (fair) trace inclusion. We show how a (fair) trace inclusion relation can be established for a pair of fair I/O automata by using TLA.

The paper proceeds as follows. Section 2 briefly presents TLA. Section 3 contains the necessary preliminaries about fair I/O automata. In Section 4, we present our approach to the specification of fair traces by using TLA. In Section 5, we show how to verify (fair) trace inclusion between fair I/O automata with TLA and give an example of using the TLC model checker for this purpose. Section 6 contains a discussion and concludes this paper.

2. An Introduction to TLA

TLA is a simple linear-time temporal logic for specifying and reasoning about safety and liveness of concurrent systems (Lamport, 1994). Besides other symbols, the language of TLA contains a countably infinite set $\mathcal{V}$ of variables, partitioned into disjoint countably infinite sets $\mathcal{V}_F$ and $\mathcal{V}_R$ of flexible and rigid variables (Merz, 2003). In the sequel, the flexible variables will be called *variables* and the rigid ones *constants*, as in Lamport (1994, 2002). The language also contains a set Val of values, which is supposed to include different kinds of numbers, sets, strings, such as "csz", and similar. The semantics of TLA is defined in terms of *behaviours*, which are infinite sequences of states. A state is an assignment of values from Val to all the variables from $\mathcal{V}_F$. The latter represent values that may change from state to state, whereas constants do not change their value. In the sequel, we will use **St** to denote the set of all possible states and $s|_V$ to denote the restriction of a state s to a set of variables $V \subset \mathcal{V}_F$.

Temporal formulas of TLA, also called *TLA formulas*, are built from actions, Boolean operators, and temporal operators $\Box$ and $\exists$. No variable $v \in \mathcal{V}_F$ should be of the form v' because $'$ ('prime') is a special symbol of TLA, which can be applied to variables. *Actions* are first-order Boolean expressions which can contain values, constants, (unprimed) variables and primed variables. They are evaluated over pairs of states, called steps.

Informally, the unprimed variables denote their value in the 'current' state s and the primed ones in the 'next' one, t, of a state pair $\langle s, t\rangle$. If A is true in a state pair, the latter is called an *A step*. For example, action $y < 0 \wedge y' = y + 1$ informally says that $y < 0$ in the current state and that it is incremented by 1 in the next one. For v a variable or a tuple of variables, $[A]_v$ is short for $A \vee \textsc{unchanged}\ v$, and $\textsc{unchanged}\ v$ means $v' = v$ ($'$ applied to a tuple of variables is the same as applying it to each of them). $\langle A\rangle_v$ is short for $A \wedge (v' \neq v)$. A *predicate P* is an action with no primed variables, with the consequence that it is true in a state pair $\langle s, t\rangle$ iff it is true in s. The predicate $\textsc{Enabled}\ A$ (read as *action A is enabled*) is defined to be true in a state s iff there exists such a state t that $\langle s, t\rangle$ is an A step. For example, action $y < 0 \wedge y' = y + 1$ is enabled in every state in which $y < 0$.

The temporal formula $\Box F$ (read *always F*) is true of a behaviour iff F is true of all its suffixes. $\Diamond F$ (read *eventually F*) is defined as $\neg\Box\neg F$. For x a variable, $\exists x : F$ essentially means that there is some way of choosing a sequence of values for x such that the temporal formula F holds, but we do not care what these values are. It is said that x is hidden in F. If x is a tuple of variables $x_1, \ldots, x_n$, $\exists x : F$ is short for $\exists x_1 : \ldots \exists x_n : F$.

For an action A, $WF_{vars}(A) \triangleq \Box\Diamond\langle A\rangle_{vars} \vee \Box\Diamond(\neg\textsc{Enabled}\langle A\rangle_{vars})$ ($\triangleq$ means 'by definition') specifies a *weak fairness* condition. It asserts that either action $\langle A\rangle_{vars}$ occurs infinitely often during a behaviour or it is infinitely often disabled, or equivalently, that always, if action $\langle A\rangle_{vars}$ is always enabled, it eventually occurs. $SF_{vars}(A) \triangleq \Box\Diamond\langle A\rangle_{vars} \vee \Diamond\Box(\neg\textsc{Enabled}\langle A\rangle_{vars})$ specifies a *strong fairness* condition. It asserts that either action $\langle A\rangle_{vars}$ occurs infinitely often during a behaviour or it is disabled from some state on.

A temporal formula (or an action or a predicate) is said to be *valid* iff it is true of all possible behaviours (in all possible pairs of states or, respectively, in all states).

Most system specifications can be written as a 'canonical-form' TLA formula $\exists x : Init \wedge \Box[Next]_{vars} \wedge Liveness$ (Lamport, 2002), where *Init* is a predicate, *Next* an action (usually a disjunction of actions), *vars* a tuple of variables the system may change, x a tuple consisting of some of the variables from *vars* (e.g. 'internal' variables of the system), and *Liveness* a conjunction of fairness conditions on (some disjuncts of) *Next*. Informally, a behaviour satisfies this formula iff it is possible to choose the values for x such that *Init* is true for the initial state, every pair of consecutive states is a $[\mathit{Next}]_{\mathit{vars}}$ step and the behaviour satisfies *Liveness*.

Suppose that we would like to specify a program (i.e. its runs) which operates on variables **x** and **y**. Suppose that they have already been initialised to 0 during their definition in the program and that the rest of the code consists of the following sequence of statements: $\mathbf{x} = \mathbf{x} + \mathbf{1}$; $\mathbf{y} = \mathbf{y} + \mathbf{1}$;. Therefore, it is expected to have a single program run—the following sequence of states: $\langle\langle x = 0, y = 0\rangle, \langle x = 1, y = 0\rangle, \langle x = 1, y = 1\rangle\rangle$. As TLA is defined to specify infinite sequences of states, it can actually be used for the specification of this run with infinitely many states $\langle x = 1, y = 1\rangle$ added at the end. The TLA specification could be as follows (cf. Lamport, 1994): $\exists pc : (pc = 0) \wedge (x = 0) \wedge (y = 0) \wedge \Box[\mathit{Next}]_{\mathit{vars}} \wedge \mathit{WF}_{\mathit{vars}}(\mathit{Next})$, where $\mathit{vars} \triangleq \langle pc, x, y\rangle$ and $\mathit{Next} \triangleq ((pc = 0) \wedge (pc' = 1) \wedge (x' = x + 1) \wedge \text{UNCHANGED}\, y) \vee ((pc = 1) \wedge (pc' = 2) \wedge (y' = y + 1) \wedge \text{UNCHANGED}\, x)$.

A 'program counter' variable is introduced and hidden because it serves as an auxiliary variable in order to be able to properly specify the changes of x and y. The specification allows behaviours which exactly agree with the program run in variables x and y: in the initial state x and y are equal to 0, in the next state x becomes equal to 1, in the state following this one y becomes equal to 1, and in all the subsequent states they remain equal to 1. The weak fairness condition assures that behaviours in which both x and y always remain equal to 0 or y does are not allowed. However, notice that the specification allows behaviours in which x is not incremented immediately after the initial state and behaviours in which y is not incremented immediately after the change of x. This is because TLA is unable to specify the exact state at which an enabled action (i.e. a state change) must occur. It can only specify that it eventually occurs. By definition, every TLA formula allows stuttering steps.

Formally, a step $\langle s, t\rangle$ is called a *V-stuttering* (or stuttering if V is evident from the context) step iff $s|_V = t|_V$, i.e. iff s and t are V-equivalent (meaning that they assign the same values to the variables in V). A behaviour is called *V-stuttering free* iff it does not contain V-stuttering steps except possibly at the end. Two behaviours are said to be *V-stuttering equivalent* iff they differ only in the number of consecutive V-equivalent states. TLA formulas are *stuttering-invariant*, which means the following. Suppose that F is a TLA formula and V the set of its free variables, such as, for instance, the set $\{x, y\}$ in the program specification given in the above example. F cannot distinguish between V-stuttering equivalent behaviours.

Notice that formulas of the form $\Box[\mathit{Next}]_{\mathit{vars}}$ are stuttering-invariant because they allow *vars*-stuttering steps. Let us stress that formulas of the form $\exists x : F$ are also stuttering-invariant (Lamport, 1994). For example, the program specification given above does not distinguish between behaviours which differ only in the number of consecutive states in which x and y do not change. It follows that it does not specify only the behaviours that directly correspond to the program run in x and y, but also the behaviours that are $\{x, y\}$-stuttering equivalent to them, i.e. it specifies an equivalence class.

If F is a TLA formula specifying a system and G one specifying a property, the system has that property iff the implication $F \Rightarrow G$ is valid. For any TLA formulas F and G, if x, y are tuples of variables, the implication $(\exists x : F) \Rightarrow (\exists y : G)$ can be proved by exhibiting the validity of $F \Rightarrow \overline{G}$, where $\overline{G}$ is obtained by substituting $\overline{y_i}$ for the free occurrences of y_i in G for all i. Each $\overline{y_i}$ is a function of variables that occur in F. The functions are collectively called a *refinement mapping*.

3. Fair I/O Automata

We consider the kind of I/O automaton model called fair I/O automaton (Müller, 1998) in order to be able to specify systems with the help of fairness conditions as with many specification formalisms (Romijn and Vaandrager, 1996), including TLA. In this section, let us temporarily forget about TLA and look at the I/O automata terminology (cf. Müller, 1998; Søgaard-Andersen *et al.*, 1993; Tuttle, 1987).

DEFINITION 1. A *fair I/O automaton A* consists of the following components:

- An *action signature* $sig(A) = (in(A), out(A), int(A))$ consisting of disjoint sets of *input*, *output*, and *internal actions*, respectively. $ext(A)$ denotes the set $in(A) \cup out(A)$ of *external actions*, $local(A)$ the set $out(A) \cup int(A)$ of *locally controlled actions*, and $acts(A)$ the set $ext(A) \cup int(A)$ of all actions.
- A set $states(A)$ of states.
- A nonempty set $start(A)$ of start states ($start(A) \subseteq states(A)$).
- A transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$. The transition relation $steps(A)$ must have the property that for each state $s \in states(A)$ and each input action $a \in in(A)$ there exists a state $s' \in states(A)$ such that $(s, a, s') \in steps(A)$.
- Sets $wfair(A)$ and $sfair(A)$ of *weak fairness sets* and *strong fairness sets*, respectively, which are subsets of $local(A)$.

In the sequel, by I/O automata we will mean fair I/O automata unless being evident from the context that we mean otherwise. A (fair) I/O automaton with empty sets $wfair(A)$ and $sfair(A)$ is called *safe I/O automaton*. For A a fair I/O automaton, let $safe(A)$ denote the safe I/O automaton obtained from A by setting $wfair(A)$ and $sfair(A)$ to empty sets.

An action a is *enabled* in a state s if there exists a state s' such that $(s, a, s') \in steps(A)$. A set $\mathcal{A}$ of actions is said to be enabled in state s if there exists an action $a \in \mathcal{A}$ such that a is enabled in s. An action or set of actions which is not enabled in a state s is said to be disabled in s.

An *execution* α of an I/O automaton A is a finite or infinite sequence $\alpha = \langle s_0, a_1, s_1, \ldots \rangle$ of alternating states and actions of A beginning with a state, and if it is finite, also ending with a state, such that $s_0 \in start(A)$ and for all i, $(s_i, a_{i+1}, s_{i+1}) \in steps(A)$. The set of all executions of A is denoted by $execs(A)$. A state s of A is *reachable* if there exists a finite execution of A that ends in s.

The *trace* of an execution α is the subsequence of α consisting of all the external actions of A, i.e. the restriction of sequence α to the elements of $ext(A)$. For example, if $\alpha = \langle s_0, a_1, s_1, a_2, s_2, a_3, s_3 \rangle$ and only a_1 and a_3 are external actions, then its trace is the sequence $\langle a_1, a_3 \rangle$. We say that γ is a *trace of A* if there exists an execution α of A such that γ is the trace of α. The set of all traces of A is denoted by $traces(A)$.

An execution α of a fair I/O automaton A is *weakly fair* if 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 disabled.

An execution α of A is *strongly fair* if 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.
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.

An execution α is *fair* if it is both weakly and strongly fair. The set of all fair executions of A is denoted by $\mathit{fairexecs}(A)$. The set of all traces originating from the fair executions of A is denoted by $\mathit{fairtraces}(A)$. For safe I/O automata, the notions of executions (respectively traces) and fair executions (respectively fair traces) coincide.

For A a fair I/O automaton, it is desirable that $\mathit{safe}(A)$ alone specify the safety properties of A, i.e. that these are completely determined by $\mathit{execs}(A)$. Sets $\mathit{wfair}(A)$ and $\mathit{sfair}(A)$ should only specify the liveness properties of A, i.e. it should be possible to extend every finite execution of A to a fair execution of A. Additionally, it should be possible to do that independently of the inputs provided by the environment of A (Romijn and Vaandrager, 1996). In the literature (e.g. Segala *et al.*, 1998), a pair (B, L) consisting of a safe I/O automaton B and a set $L \subseteq \mathit{execs}(B)$ such that B can always generate an execution in L independently of the environment is called a *live I/O automaton*.

Definition 2. (Cf. Müller, 1998.) Let A be a fair I/O automaton and Λ a subset of its locally controlled actions. Λ is *input-resistant* iff for each pair of reachable states s, t and each input action a, if Λ is enabled at s and $(s, a, t) \in \mathit{steps}(A)$, then Λ is enabled at t. A fair I/O automaton is said to be *input-resistant* if every set in $\mathit{sfair}(A)$ is input-resistant.

Romijn and Vaandrager (1996) prove that if a fair I/O automaton A is (i) input-resistant and (ii) at most countably many sets in $\mathit{wfair}(A) \cup \mathit{sfair}(A)$ are enabled in each reachable state of A, then A induces live automaton $\mathit{live}(A) = (\mathit{safe}(A), \mathit{fairexecs}(A))$.

We are interested in the verification of I/O automata by using implementation relations. Fair I/O automata which induce live I/O automata can be verified by using the following ones.

Definition 3. (Cf. Søgaard-Andersen *et al.*, 1993; Müller, 1998.) Let A and B be fair I/O automata with $\mathit{in}(A) = \mathit{in}(B)$ and $\mathit{out}(A) = \mathit{out}(B)$, such that they induce live automata $\mathit{live}(A) = (\mathit{safe}(A), \mathit{fairexecs}(A))$ and, respectively, $\mathit{live}(B) = (\mathit{safe}(B), \mathit{fairexecs}(B))$. There is a *safe trace inclusion*, written as $A \preceq_S B$, iff $\mathit{traces}(A) \subseteq \mathit{traces}(B)$. There is a *fair trace inclusion*, written as $A \preceq_F B$, iff $\mathit{fairtraces}(A) \subseteq \mathit{fairtraces}(B)$.

Clearly, for safe I/O automata, the safe and fair trace inclusion relations coincide.

4. Specification of Fair Traces with TLA

As the meaning of TLA is only defined for infinite (state) sequences, let us make all the finite executions of an I/O automaton A infinite as in Søgaard-Andersen *et al.* (1993). Let ζ denote a special *stuttering action* and suppose that it cannot be in the action signature of any automaton. A *stuttering step* is any triple of the form (s, ζ, s), where s is a state. For any execution $\alpha = \langle s_0, a_1, s_1, \ldots \rangle$, let the *extended execution* $\hat{\alpha} \triangleq \alpha$ if α is infinite, and $\hat{\alpha} \triangleq \langle s_0, a_1, s_1, \ldots, a_n, s_n, \zeta, s_n, \zeta, s_n, \ldots \rangle$ if α is finite and ends in s_n. Analogously, for any trace, we define the *extended trace* to be the same if it is infinite, and obtained by adding infinitely many stuttering actions at the end of it if finite.

There is obviously a one-to-one correspondence between the set $execs(A)$ and the set of extended executions obtained from the executions in $execs(A)$. This is because all the infinite executions remain intact and, by definition, do not contain stuttering steps, whereas every finite execution only gets the stuttering steps at the end. It can also be proved (cf. Søgaard-Andersen *et al.*, 1993) that for all finite executions α, α satisfies the weak (respectively strong) fairness conditions as defined for finite executions exactly if $\hat{\alpha}$ satisfies the weak (respectively strong) fairness conditions as defined for infinite executions. Therefore, there is also a one-to-one correspondence between the set $fairexecs(A)$ and the set of fair executions in the set of extended executions of A.

Let the definitions of traces of the extended executions of A remain the same as for the original executions, by ζ treated as an external action, except for the extended executions which contain only steps with internal actions at the end. Since for such an execution, the original definition of traces gives a finite action sequence, the infinite number of stuttering actions ζ must be added at its end. Obviously, the traces of previously finite executions contain the infinite number of actions ζ at the end. There is, evidently, a one-to-one correspondence between the original set of (fair) traces, the set of (fair) extended traces and the set of (fair) traces obtained from the set of extended (fair) executions of A. In the sequel, we will, therefore, identify (the notations for) (fair) executions and, respectively, traces with (the notations for) their extended versions. Clearly, the definitions of both implementation relations remain valid.

From this point on, we can proceed with the preparation of TLA specifications of I/O automata in a similar way to how TLA is used for specifying program runs (e.g. Lamport, 1994; Reynolds, 1998). It means that the TLA specifications will not specify exactly the (extended) executions or traces of I/O automata, but the union of their equivalence classes, where for every execution (and consequently every trace), its equivalence class will contain, besides it, all the executions (respectively traces) which can be obtained from it by adding finitely many stuttering steps at some states (respectively stuttering actions after some actions). And, most importantly, all the results of reasoning about such TLA specifications will be valid for the original executions or traces.

4.1. *Specification of Executions*

We will first explain the specification of executions of an I/O automaton with help of an example. Let us write the TLA specification of $execs(A1)$ for I/O automaton $A1$ with

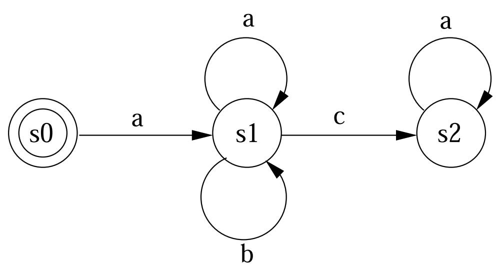


Fig. 1. State-transition graph of I/O-automaton A.

$in(A1) = \{a\}$, $out(A1) = \{c\}$, $int(A1) = \{b\}$, $states(A1) = \{s0, s1, s2\}$, $start(A1) = \{s0\}$, $steps(A1) = \{(s0, a, s1), (s1, a, s1), (s2, a, s2), (s1, b, s1), (s1, c, s2)\}$, $wfair(A1) = \{\{b, c\}\}$, $sfair(A1) = \{\}$. Figure 1 contains a graphical representation of $steps(A1)$. Examples of executions of $A1$ are $\langle s0\rangle$, $\langle s0, a, s1\rangle$, $\langle s0, a, s1, a, s1\rangle$, $\langle s0, a, s1, b, s1\rangle$, $\langle s0, a, s1, a, s1, \ldots\rangle$.

We need to represent the states with a TLA variable. Let it be st, its value "s0", "s1", or "s2" representing states s0, s1, and s2 respectively. We can specify the transitions between states in a similar way to the program statements in the example in Section 2. However, the specification also has to represent the action labels of the transitions. Let the action labels be denoted by TLA values "a", "b", and "c". We introduce a special variable Ev in which we record the action label of the transition being specified. In the sequel, let $ActInit(Ev) \triangleq Ev = \langle \text{“0”}, \langle\rangle\rangle$ and $Act(Ev, v) \triangleq Ev' = \langle \textsc{if } Ev[1] = \text{“0”} \textsc{ then } \text{“1”} \textsc{ else } \text{“0”}, v\rangle$.

The TLA specification of $execs(A1)$ can then be written as follows:

$$SpecE(st, Ev) \triangleq Init(st, Ev) \land \Box[Next(st, Ev)]_{vars}$$

where

$$vars \triangleq \langle st, Ev\rangle$$
$$StInit(st) \triangleq st = \text{“s0”}$$
$$Init(st, Ev) \triangleq StInit(st) \land ActInit(Ev)$$
$$InActionA(st, Ev) \triangleq Act(Ev, \langle\text{“a”}\rangle) \land (st' = \textsc{if } st = \text{“s0”} \textsc{ then } \text{“s1”} \textsc{ else } st)$$
$$OutActionC(st, Ev) \triangleq Act(Ev, \langle\text{“c”}\rangle) \land (st = \text{“s1”}) \land (st' = \text{“s2”})$$
$$IntActionB(st, Ev) \triangleq Act(Ev, \langle\text{“b”}\rangle) \land (st = \text{“s1”}) \land \textsc{unchanged } st$$
$$Next(st, Ev) \triangleq InActionA(st, Ev) \lor OutActionC(st, Ev) \lor IntActionB(st, Ev)$$

The TLA action $InActionA(st, Ev)$ represents the transitions from Fig. 1 labelled with action a, action $OutActionC(st, Ev)$ represents the transition labelled with c, and action $IntActionB(st, Ev)$ the transition labelled with b.

$SpecE(st, Ev)$ says that initially, st is set to "s0" and Ev is equal to the pair of values $\langle\text{“0”}, \langle\rangle\rangle$, and that subsequently, at every step, either both st and Ev remain unchanged or a currently enabled transition of the automaton occurs. If the latter is the case, the first component of Ev, denoted $Ev[1]$, changes to “1” if previously “0”, and vice versa, and in the second component of Ev, denoted $Ev[2]$, the label of the transition is recorded. For example, in the initial state, only the TLA action $InActionA(st, Ev)$ is enabled. If it occurs, Ev becomes equal to $\langle$“1”, $\langle$“a”$\rangle\rangle$ and st to “s1”, which represents the execution of the transition from $s0$ to $s1$.

We want to recognise from the Ev variable alone whether an I/O automaton action occurred. That is why we defined $Act(Ev, v)$ to unconditionally change Ev. If only the action were recorded in Ev, i.e. if $Act(Ev, v)$ were defined as $Ev' = v$, the repetition of an action could not be distinguished from stuttering in Ev. For example, the sequence of values of Ev in the behaviour representing execution $\langle s0, a, s1\rangle$ and respectively $\langle s0, a, s1, a, s1\rangle$ would be the same: $\langle\langle\rangle$, $\langle$“a”$\rangle$, $\langle$“a”$\rangle, \ldots\rangle$. For our definition of $Act(Ev, v)$, it is $\langle\langle$“0”, $\langle\rangle\rangle$, $\langle$“1”, $\langle$“a”$\rangle\rangle$, $\langle$“1”, $\langle$“a”$\rangle\rangle, \ldots\rangle$ for the first execution and $\langle\langle$“0”, $\langle\rangle\rangle$, $\langle$“1”, $\langle$“a”$\rangle\rangle$, $\langle$“0”, $\langle$“a”$\rangle\rangle$, $\langle$“0”, $\langle$“a”$\rangle\rangle, \ldots\rangle$ for the second one.

Let A denote a fair I/O automaton (Definition 1), V the set of TLA variables encoding the automaton states, and Ev a TLA variable different from those from V. From the above example, it can easily be seen that generally, a TLA specification of $execs(A)$ can be written as follows:

$$Init(V, Ev) \wedge \Box[Next(V, Ev)]_{\langle V, Ev\rangle} \tag{1}$$

where

- $Init(V, Ev) \triangleq StInit(V) \wedge ActInit(Ev)$ with $StInit(V)$ a TLA predicate such that for all $s \in \mathbf{St}$, it is true in s iff $s|_V \in start(A)$, and
- $Next(V, Ev) \triangleq \bigvee_{a \in acts(A)} Next_a(V, Ev)$ with $Next_a(V, Ev) \triangleq NState_a(V) \wedge Act(Ev, a)$, where $NState_a(V)$ is a TLA action such that for all $s, t \in \mathbf{St}$, $NState_a(V) \wedge Act(Ev, a)$ is true in $\langle s, t\rangle$ iff $(s|_V, a, t|_V) \in steps(A)$.

$StInit(V)$ specifies the initial states of A, and $Next_a(V, Ev)$ describes all the possible transitions labelled by a. The $\{V, Ev\}$-stuttering steps represent the stuttering steps as defined for I/O automata. Please, note that the restriction to V has to be applied to the states of TLA behaviours in order to obtain the corresponding states of A because the states in TLA assign values to all the variables of TLA. It follows that one execution of A is in fact represented by the set of all the TLA behaviours which agree with the execution in the values of V and in the actions recorded in Ev, but assign arbitrary values to all the other variables. For the simplicity of writing, we sometimes ignore this fact as usual in papers on TLA.

4.2. *Specification of Fair Executions*

In TLA, an occurrence of action a of a fair I/O automaton A can be expressed by $\langle Next_a(V, Ev)\rangle_{\langle V, Ev\rangle}$ (which happens to be equivalent to $Next_a(V, Ev)$). It can easily be seen that the enabledness of action a in a state of A can be expressed with the ENABLED predicate. Formally, for all $s \in \mathbf{St}$, if $s|_V \in states(A)$, then $\text{ENABLED}\langle Next_a(V, Ev)\rangle_{\langle V, Ev\rangle}$

is true in s iff action a is enabled in state $s|_V$. The conjunct $Act(Ev, a)$ in $Next_a(V, Ev)$ ensures that this is the case even if the transition relation of A contains steps $(s|_V, a, s|_V)$ for some a and s.

Consequently, the TLA specification of $fairexecs(A)$ can be obtained by adding weak and strong fairness conditions to (1):

$$Init(V, Ev) \wedge \Box[Next(V, Ev)]_{\langle V, Ev\rangle} \wedge Liveness(V, Ev) \tag{2}$$

where $Liveness(V, Ev) \triangleq \wedge \bigwedge_{W \in wfair(A)} WF_{\langle V, Ev\rangle}(\bigvee_{a \in W} Next_a(V, Ev))$
$\wedge \bigwedge_{S \in sfair(A)} SF_{\langle V, Ev\rangle}(\bigvee_{a \in S} Next_a(V, Ev))$.

V and Ev in the parentheses in the above introduced symbols (and similar in the rest of the paper) denote the free variables of the formulas they represent. Please, observe that instead of writing V in the parentheses and in the subscripts such as $\langle V, Ev\rangle$, we should list the variables of V. We write V for convenience.

Notice that the set $wfair(A1)$ of our example automaton $A1$ is nonempty. The TLA specification of $fairexecs(A1)$ can be written by adding the fairness condition to the specification of $execs(A1)$:

$$SpecFE(st, Ev) \triangleq SpecE(st, Ev) \wedge Liveness(st, Ev)$$

where

$$Liveness(st, Ev) \triangleq WF_{vars}(OutActionC(st, Ev) \vee IntActionB(st, Ev)).$$

Examples of fair executions of $A1$ are $\langle s0\rangle$, $\langle s0, a, s1, c, s2\rangle$, $\langle s0, a, s1, a, s1, c, s2\rangle$, $\langle s0, a, s1, b, s1, c, s2\rangle$, $\langle s0, a, s1, b, s1, b, s1, \ldots\rangle$ (i.e. an infinite execution in which action b is repeated indefinitely in state $s1$), but not for instance $\langle s0, a, s1\rangle$, because b and c, which are together in a set of $wfair(A1)$, are enabled in the last state.

4.3. *Specification of Traces*

Now, we are interested in writing the TLA specification of traces of a fair I/O automaton A. The only free variable of the specification should be one in which only the occurrences of the external actions of A would be recorded (we shall call it Ev as before), thus representing traces, and the specification should be stuttering-invariant. If A has no internal actions, the specification can be obtained by hiding the variables representing its states in the TLA specification of its executions (1) by using $\exists\!\!\!\exists$. If not, the specification of $traces(A)$ can be written in the same way as (1), except that in the definition of $Next_a(V, Ev)$ for every internal action a, Ev should remain unchanged, i.e. the occurrence of a should not be recorded in Ev, and V should finally be hidden:

$$\exists\!\!\!\exists V : Init(V, Ev) \wedge [NextH(V, Ev)]_{\langle V, Ev\rangle} \tag{3}$$

where $\mathit{NextH}(V, \mathit{Ev}) \triangleq \bigvee_{a \in ext(A)} \mathit{Next}_a(V, \mathit{Ev}) \vee \bigvee_{a \in int(A)} \mathit{Next}_a(V, \mathit{Ev})$ with $\mathit{Next}_a(V, \mathit{Ev}) \triangleq \mathit{NState}_a(V) \wedge \mathit{Act}(\mathit{Ev}, a)$ for all $a \in ext(A)$ and $\mathit{Next}_a(V, \mathit{Ev}) \triangleq \mathit{NState}_a(V) \wedge \mathit{Ev}' = \mathit{Ev}$ for all $a \in int(A)$.

For our example automaton $A1$, the specification of traces is as follows:

$$\mathit{SpecT}(\mathit{Ev}) \triangleq \exists st : \mathit{Init}(st, \mathit{Ev}) \wedge \Box[\mathit{NextH}(st, \mathit{Ev})]_{vars}$$

where

$$\begin{aligned}
&vars \triangleq \langle st, \mathit{Ev}\rangle \\
&\mathit{StInit}(st) \triangleq st = \text{“s0”} \\
&\mathit{Init}(st, \mathit{Ev}) \triangleq \mathit{StInit}(st) \wedge \mathit{ActInit}(\mathit{Ev}) \\
&\mathit{InActionA}(st, \mathit{Ev}) \triangleq \mathit{Act}(\mathit{Ev}, \langle\text{“a”}\rangle) \wedge (st' = \textsc{if}\ st = \text{“s0”}\ \textsc{then}\ \text{“s1”}\ \textsc{else}\ st) \\
&\mathit{OutActionC}(st, \mathit{Ev}) \triangleq \mathit{Act}(\mathit{Ev}, \langle\text{“c”}\rangle) \wedge (st = \text{“s1”}) \wedge (st' = \text{“s2”}) \\
&\mathit{IntActionBH}(st, \mathit{Ev}) \triangleq (st = \text{“s1”}) \wedge \textsc{unchanged}\ \langle st, \mathit{Ev}\rangle \\
&\mathit{NextH}(st, \mathit{Ev}) \triangleq \mathit{InActionA}(st, \mathit{Ev}) \vee \mathit{OutActionC}(st, \mathit{Ev}) \vee \mathit{IntActionBH}(st, \mathit{Ev})
\end{aligned}$$

Notice that instead of the TLA action $\mathit{IntActionB}(st, \mathit{Ev})$, the specification contains the TLA action $\mathit{IntActionBH}(st, \mathit{Ev})$, which does not record the occurrence of action b in Ev. Examples of traces of $A1$ are $\langle\rangle$, $\langle a\rangle$, $\langle a, a\rangle$, $\langle a, a, \ldots\rangle$, $\langle a, c\rangle$.

4.4. *Specification of Fair Traces*

Clearly, if a fair I/O automaton A does not contain internal actions, the specification of its fair traces can be obtained by hiding V in the specification of its fair executions.

Otherwise, following the way the specification of $\mathit{fairexecs}(A)$ (2) was obtained from the specification of $\mathit{execs}(A)$ (1), one is tempted to write the specification of $\mathit{fairtraces}(A)$ by adding the fairness conditions for the TLA actions that represent fairness sets from $\mathit{wfair}(A)$ and $\mathit{sfair}(A)$ to (3):

$$\exists V : \mathit{Init}(V, \mathit{Ev}) \wedge [\mathit{NextH}(V, \mathit{Ev})]_{\langle V, \mathit{Ev}\rangle} \wedge \mathit{LivenessH}(V, \mathit{Ev}) \tag{4}$$

where $\mathit{LivenessH}(V, \mathit{Ev}) \triangleq \wedge \bigwedge_{W \in \mathit{wfair}(A)} \mathit{WF}_{\langle V, \mathit{Ev}\rangle}(\bigvee_{a \in W} \mathit{Next}_a(V, \mathit{Ev}))$
$\wedge \bigwedge_{S \in \mathit{sfair}(A)} \mathit{SF}_{\langle V, \mathit{Ev}\rangle}(\bigvee_{a \in S} \mathit{Next}_a(V, \mathit{Ev}))$.

It should, however, be noticed that the transition relation of an I/O automaton A may contain steps (s, a, s) for some $a \in acts(A)$ and $s \in states(A)$. It follows that $\mathit{NState}_a(V)$ does not necessarily change V. In (1) and (2), the I/O automaton transitions are represented by $\mathit{Next}_a(V, \mathit{Ev})$ in which the conjunct $\mathit{Act}(\mathit{Ev}, a)$ guarantees that the state in TLA changes for every I/O automaton step. In (3) and consequently in (4), this is not the case anymore, as we leave Ev unchanged in $\mathit{Next}_a(V, \mathit{Ev})$ for all $a \in int(A)$.

Reynolds (1998) shows that if some TLA actions do not always change the state, it can happen that the canonical-form TLA specification including weak and/or strong fairness conditions on such actions does not specify all the behaviours of the system being specified. Some legal behaviours of the system might not be allowed by the specification as not being weakly or strongly fair.

This would, for instance, be the case for our example automaton $A1$. Following (4), the liveness specification $\mathit{LivenessH}(V, Ev)$ for $A1$ would be:

$$\mathit{WF}_{\langle st,Ev\rangle}(\mathit{OutActionC}(st, Ev) \vee \mathit{IntActionBH}(st, Ev)) \qquad (5)$$

which is defined as

$$\begin{aligned}&\vee\Box\Diamond\langle \mathit{OutActionC}(st, Ev) \vee \mathit{IntActionBH}(st, Ev)\rangle_{\langle st,Ev\rangle}\\ &\vee\Box\Diamond(\neg \textsc{Enabled}\ \langle \mathit{OutActionC}(st, Ev) \vee \mathit{IntActionBH}(st, Ev)\rangle_{\langle st,Ev\rangle}).\end{aligned}$$

As $\mathit{IntActionBH}(st, Ev)$ does change neither st nor Ev, $\langle \mathit{OutActionC}(st, Ev) \vee \mathit{IntActionBH}(st, Ev)\rangle_{\langle st,Ev\rangle}$ is equivalent to $\langle \mathit{OutActionC}(st, Ev)\rangle_{\langle st,Ev\rangle}$, giving that (5) is equivalent to $\mathit{WF}_{\langle st,Ev\rangle}(\mathit{OutActionC}(st, Ev))$, which expresses a stronger weak fairness condition than defined by $\mathit{wfair}(A)$. So, for instance, from the fair execution $\langle s0, a, s1, b, s1, b, s1, \ldots\rangle$ of $A1$, we obtain fair trace $\langle a\rangle$, but it is not allowed by the specification of the form (4) for this automaton because it does not satisfy the weak fairness condition (5). The latter namely requires that if c is always enabled from some state on (note that this is the case if only b is executed all the time after reaching $s1$ from $s0$), it must eventually occur.

Therefore, the specification of fair traces in the form (4) is generally possible only under the condition that for all $W \in \mathit{wfair}(A)$ and $S \in \mathit{sfair}(A)$, for all $a \in W \cap \mathit{int}(A)$ and, respectively, for all $a \in S \cap \mathit{int}(A)$, $\mathit{NState}_a(V)$ (or more precisely, $\mathit{NState}_a(V) \wedge T$ where T is the type invariant for V) implies $V' \neq V$, i.e. that all the TLA actions representing the internal actions of A for which fairness conditions are imposed, change some variable of the specification other than Ev.

Nevertheless, we would like to have a generally valid pattern for the specification of fair traces. We have found that it is possible to apply a solution similar to the one proposed by Reynolds (1998) for the specification of concurrent programs based on multiset rewriting. Instead of the TLA formulae of the form $\mathit{WF}_{vars}(\mathcal{A})$ and $\mathit{SF}_{vars}(\mathcal{A})$, the TLA formulae of the form $\mathit{VWF}_{vars}(\mathcal{A})$ and $\mathit{ASF}_{vars}(\mathcal{A})$ introduced by Reynolds (1998) should be used for the specification of fairness sets.

$\mathit{VWF}_{vars}(\mathcal{A})$ expresses *very weak fairness*:

$\mathit{VWF}_{vars}(\mathcal{A}) \triangleq \mathit{WF}_{vars}(\mathcal{A}) \vee \Box\Diamond\textsc{Enabled}\ (\mathcal{A} \wedge vars' = vars)$.

$\mathit{ASF}_{vars}(\mathcal{A})$ expresses *almost strong fairness*:

$\mathit{ASF}_{vars}(\mathcal{A}) \triangleq \mathit{SF}_{vars}(\mathcal{A}) \vee \Box\Diamond\textsc{Enabled}\ (\mathcal{A} \wedge vars' = vars)$.

The specification of fair traces of any fair I/O automaton A can be obtained from the specification of traces (3) in a uniform way. It can be written in the same form as (4), but $\mathit{LivenessH}(V, Ev)$ has to be as follows:

$$\begin{aligned}\mathit{LivenessH}(V, Ev) \triangleq\ &\wedge \textstyle\bigwedge_{W\in \mathit{wfair}(A)} \mathit{VWF}_{\langle V,Ev\rangle}(\bigvee_{a\in W} \mathit{Next}_a(V, Ev))\\ &\wedge \textstyle\bigwedge_{S\in \mathit{sfair}(A)} \mathit{ASF}_{\langle V,Ev\rangle}(\bigvee_{a\in S} \mathit{Next}_a(V, Ev)).\end{aligned}$$

Let us remark that it is not necessary to apply the new version of fairness conditions for fairness sets W and S which do not contain internal actions a labelling steps (s, a, s) for some states s. However, instead of checking as to whether such actions exist, it is generally easier to use the new versions for all the fairness sets. The new fairness conditions for fairness sets W and S which do not contain the critical internal actions are, anyway, equivalent to the old ones.

We will demonstrate in Section 5 that by using the new form of fairness conditions we obtain a proper TLA specification of the fair traces of our example automaton $A1$. Its fairness part should be:

$$\begin{aligned} \mathit{LivenessH}(st, Ev) \triangleq\ & \vee \mathit{WF}_{\langle st,Ev\rangle}(\mathit{OutActionC}(st, Ev) \vee \mathit{IntActionB}(st, Ev)) \\ & \vee \Box\Diamond \textsc{Enabled}\ (\wedge \mathit{OutActionC}(st, Ev) \vee \mathit{IntActionBH}(st, Ev) \\ & \qquad \wedge \langle st, Ev\rangle' = \langle st, Ev\rangle) \end{aligned}$$

which is equivalent to

$$\mathit{WF}_{\langle st,Ev\rangle}(\mathit{OutActionC}(st, Ev)) \vee \Box\Diamond \textsc{Enabled}\ (\mathit{IntActionBH}(st, Ev)).$$

5. Fair I/O Automata Implementation Relations in TLA

Suppose that $\mathit{Tr}_A(Ev)$ and $\mathit{FTr}_A(Ev)$ denote TLA specifications of traces and, respectively, fair traces of an I/O automaton A. Analogous to the (fair) executions of an I/O automaton A, since the states in TLA are assignments to all the variables in $\mathcal{V}_F$, not only to Ev, one (fair) trace of an I/O automaton has a set of representative behaviours that satisfy the TLA specification of the (fair) traces of A. This set contains all the Ev-stuttering free behaviours with the values of Ev corresponding to the actions in the (fair) trace and all the behaviours which are Ev-stuttering equivalent to them. For a trace α, let this set (the 'equivalence class' for α) be denoted by $\mathit{equ}_{Ev}(\alpha)$. For a behaviour σ and a TLA formula F, let $\sigma \models F$ denote that F is true of σ.

Proposition 1. *For fair I/O automata A and B with $in(A) = in(B)$ and $out(A) = out(B)$ which induce the corresponding live automata, $A \preceq_F B$ iff $\mathit{FTr}_A(Ev) \Rightarrow \mathit{FTr}_B(Ev)$ is valid.*

Proof.
<u>$\Rightarrow$:</u>
Assume that $A \preceq_F B$. Hence, by Definition 3, $\mathit{fairtraces}(A) \subseteq \mathit{fairtraces}(B)$, and so for every $\alpha \in \mathit{fairtraces}(A)$, $\alpha \in \mathit{fairtraces}(B)$. For a behaviour σ, assume that $\sigma \models \mathit{FTr}_A(Ev)$. We must prove that $\sigma \models \mathit{FTr}_B(Ev)$. We know that $\sigma \in \mathit{equ}_{Ev}(\alpha)$ for some $\alpha \in \mathit{fairtraces}(A)$. We know that $\alpha \in \mathit{fairtraces}(B)$ and that all the behaviours in $\mathit{equ}_{Ev}(\alpha)$ satisfy $\mathit{FTr}_B(Ev)$. Henceforth, $\sigma \models \mathit{FTr}_B(Ev)$.
<u>$\Leftarrow$:</u>
Assume the validity of $\mathit{FTr}_A(Ev) \Rightarrow \mathit{FTr}_B(Ev)$. Hence, for every σ such that $\sigma \models \mathit{FTr}_A(Ev)$, $\sigma \models \mathit{FTr}_B(Ev)$. Assume that $\alpha \in \mathit{fairtraces}(A)$. We must prove that $\alpha \in$

fairtraces(B). We know that for all $\sigma \in equ_{Ev}(\alpha)$, $\sigma \models FTr_A(Ev)$ and thus $\sigma \models FTr_B(Ev)$. It follows that $\sigma \in equ_{Ev}(\beta)$ for some $\beta \in$ *fairtraces*(B). From the fact that σ cannot be in the equivalence classes for two different fair traces, it follows that $\alpha = \beta$ and hence, $\alpha \in$ *fairtraces*(B). □

Proposition 2. *For fair I/O automata A and B with* $in(A) = in(B)$ *and* $out(A) = out(B)$ *which induce the corresponding live automata,* $A \preceq_S B$ *iff* $Tr_A(Ev) \Rightarrow Tr_B(Ev)$ *is valid.*

Proof. Analogous to the proof of Proposition 1, or by the latter by taking into account that *traces*$(A) =$ *fairtraces*$($*safe*$(A))$ and the same for B. □

We would also like to be able to check whether a fair I/O automaton A induces a live automaton by using TLA. Suppose that a TLA specification of *fairexecs*(A) in the form (2) is given. We are only interested in automata with countable (i.e. finite or countably infinite) sets *sfair*(A) and *wfair*(A) as is regularly the case in the literature (e.g. Müller, 1998; Søgaard-Andersen *et al.*, 1993). For such automata, the liveness part *Liveness*(V, Ev) is a conjunction of countably many fairness conditions. As the countable sets of fairness sets of A imply condition (ii) of Romijn and Vaandrager (1996) mentioned in Section 3, it follows that in order to assure that A induces a live automaton, it is only necessary to check whether it is input-resistant. Let S_A denote the TLA specification of *execs*(A) (1). Based on Definition 2 and the semantics of TLA we claim that A is input-resistant iff formula

$$\begin{aligned} S_A(V, Ev) \Rightarrow \Box[((\textstyle\bigvee_{a \in in(A)} Act(Ev, a)) \wedge \textsc{Enabled}\ \langle \mathcal{A} \rangle_{\langle V, Ev \rangle}) \\ \Rightarrow (\textsc{Enabled}\ \langle \mathcal{A} \rangle_{\langle V, Ev \rangle})']_{\textsc{Enabled}\ \langle \mathcal{A} \rangle_{\langle V, Ev \rangle}} \end{aligned}$$

is valid for every action $\mathcal{A} \triangleq \bigvee_{a \in S} Next_a(V, Ev)$ appearing in a formula $SF_{\langle V, Ev \rangle}(\mathcal{A})$ of the *Liveness*(V, Ev) part of the TLA specification of *fairexecs*(A), and analogously if the specification of *fairtraces*(A) is given. (Please, notice that for a predicate P, P' denotes its truth value in the 'next' state and $[N]_P$ is short for $N \vee (P' \equiv P)$.) The formula is similar to the one for checking the so-called μ-invariance of a predicate $\textsc{Enabled}\ \langle \mathcal{A} \rangle_w$ for a TLA specification of an open system (Abadi and Lamport, 1994).

We have verified the externally visible behaviour of our automaton $A1$ by checking trace inclusion in the way suggested by Propositions 1 and 2 with the TLC model checker. The latter is a part of the integrated development environment for TLA specifications, called the TLA Toolbox (2014). The specifications are actually written by using TLA$^+$ (Lamport, 2002), which is a complete formal specification language based on TLA. It complements the latter with a notation for the specification of data structures and modular specification, as well as a notation for writing hierarchically structured proofs (Lamport, 2014).

Roughly speaking, a TLA$^+$ module may contain a list of 'imported' modules, a declaration of module parameters, i.e. externally visible TLA constants and variables used in formulas inside the module, assumptions about the constants, definitions of symbols used in the TLA$^+$ specification, theorems expected to hold in the module under the specified

```
---------------------------- MODULE ActOps ----------------------------
ActInit(Ev) ≜ Ev = ⟨"0", ⟨⟩⟩
Toggle(v) ≜ IF v = "0" THEN "1" ELSE "0"
Act(Ev, a) ≜ Ev' = ⟨Toggle(Ev[1]), a⟩
========================================================================
```

Fig. 2. Module *ActOps*.

assumptions, and proofs of these theorems. Line comments in TLA$^+$ modules are denoted with $\backslash *$.

We prepared module *ActOps* (Fig. 2), in which *Act*(*Ev*, *a*) and *ActInit*(*Ev*) are defined as described in Section 4. This module can be imported by using keyword EXTENDS in the automata specification modules which need these definitions.

Formula *SpecT* in module *A*1 in Fig. 3 is the specification of *traces*(*A*1) (cf. Subsection 4.3) and *SpecFT* the specification of *fairtraces*(*A*1) (cf. Subsection 4.4). According to Section 4, variable *st* should be hidden in them, but it is not because TLC cannot handle formulas containing existential quantifier $\exists$. It suffices just to remember which variables are in fact internal.

It should be noticed that although *A*1 has a weak fairness condition on actions *b* and *c*, the latter does not guarantee action *c* eventually to occur because whenever *c* is enabled, internal action *b* is also enabled, and the weak fairness condition is fulfilled even if the latter always occurs. Therefore, it does not guarantee any externally visible progress of the automaton behaviour. It follows that it should be possible to prove that *fairtraces*(*A*1) = *traces*(*A*1). It should also be possible to prove that *A*1 externally behaves in the same way as the safe automaton, let us name it *A*1*Req*, which has the same start state and the same set of states as *A*1, the same input action *a* and output action *c*, but no internal actions, and the steps of which are the same as those of *A*1 shown in Fig. 1, except that there is no transition labelled with *b*. Formula *Spec* in module *A*1*Req* in Fig. 4 is the specification of (*fair*)*traces*(*A*1*Req*). Clearly, both *A*1 and *A*1*Req* induce live automata, so that we can verify them by establishing trace inclusion relations between them.

The presented modules do not only give (fair) trace specifications, but also define the sets of input (*in*) and output (*out*) actions of the automata, as well as the set of possible states (*States*). We followed the rule that it is useful first to specify and check the type invariant for the system being specified. An instance of module *A*1*Req* is included in module *A*1 and named *ReqModule* in order to be able to use the symbols defined in *A*1*Req* in *A*1. In TLA$^+$, for an instance *M*, *M*!*S* refers to symbol *S* defined in the module of which *M* is an instance.

We wrote some theorems at the end of the modules, but note that for the verification with the TLC model checker, theorems need not be written because the system specification (or, generally, the antecedent of the implication stated in the theorem) and the property (the consequent of the implication) to be checked are stated in a special menu of the TLA Toolbox. We verified all the theorems with TLC. In all the theorems except those expressing the type correctness, the consequent is in fact meant to be the original formula (*Spec*, *SpecT*, or respectively *SpecFT*) with free variable *st* replaced with a refinement mapping $\overline{st}$, which should, according to Section 2, be a function of the variables

```
┌──────────────────────────── MODULE A1 ────────────────────────────┐
EXTENDS ActOps
VARIABLES Ev, st
ReqModule ≜ INSTANCE A1Req
├───────────────────────────────────────────────────────────────────┤
vars ≜ ⟨Ev, st⟩
in ≜ {⟨"a"⟩}
out ≜ {⟨"c"⟩}
ext ≜ in ∪ out
ASSUME in = ReqModule!in ∧ out = ReqModule!out
States ≜ {"s0", "s1", "s2"}
StInit ≜ st = "s0"
Init ≜ ActInit(Ev) ∧ StInit
TypeInvariant ≜ ∧ Ev ∈ {"0", "1"} × (ext ∪ {⟨⟩})
                ∧ st ∈ States
├───────────────────────────────────────────────────────────────────┤
InActionA ≜ ∧ Act(Ev, ⟨"a"⟩)
            ∧ st′ = IF st = "s0" THEN "s1"
                                 ELSE st
OutActionC ≜ ∧ Act(Ev, ⟨"c"⟩)
             ∧ (st = "s1") ∧ (st′ = "s2")
IntActionBH ≜ ∧ Ev′ = Ev
              ∧ (st = "s1") ∧ (st′ = st)
NextH ≜ InActionA ∨ OutActionC ∨ IntActionBH
\* LivenessH ≜ WF_vars(OutActionC ∨ IntActionBH)
LivenessH ≜ WF_vars(OutActionC) ∨ □◇ENABLED (IntActionBH)
SpecT ≜ Init ∧ □[NextH]_vars
SpecFT ≜ SpectT ∧ LivenessH
ReqSpec ≜ ReqModule!Spec
\* CSpec ≜ (in = ReqModule!in) ∧ (out = ReqModule!out) ∧ ReqSpec
├───────────────────────────────────────────────────────────────────┤
THEOREM SpecFT ⇒ □TypeInvariant
THEOREM SpecFT ⇒ SpecT    \* fairtraces(A1) ⊆ traces(A1)
THEOREM SpecT ⇒ SpecFT    \* traces(A1) ⊆ fairtraces(A1)
THEOREM SpecFT ⇒ ReqSpec  \* fairtraces(A1) ⊆ (fair)traces(A1Req)
\* THEOREM SpecFT ⇒ CSpec
THEOREM ReqSpec ⇒ SpecFT  \* (fair)traces(A1Req) ⊆ fairtraces(A1)
└───────────────────────────────────────────────────────────────────┘
```

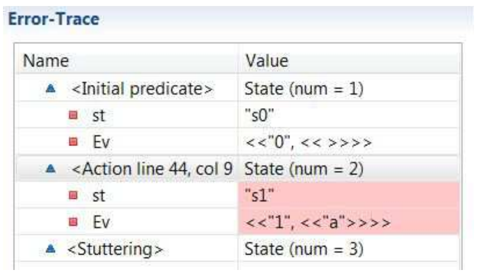
Fig. 3. Module *A1*.

of the antecedent. As the refinement mapping is simply the identity function $\overline{st} = st$, the consequent is the same as the original formula. According to Section 2, if the implications of such formulas are valid, then also the implications of such formulas with *st* hidden are valid, thereby proving the trace inclusion for $A1$ and $A1Req$ as stated in the comments beside the theorems in module $A1$ in Fig. 3, and consequently the equality of the external behaviour of automata $A1$ and $A1Req$, as well as of $fairtraces(A1)$ and $traces(A1)$.

We first deliberately used the wrong *LivenessH* formula (formula (5) from Subsection 4.4) in module $A1$ (the formula is shown as a line comment in Fig. 3) to check the implementation relations between automata $A1$ and $A1Req$. TLC reported an error when verifying the implication $ReqSpec \Rightarrow SpecFT$, i.e. whether $traces(A1Req) \subseteq fairtraces(A1)$.

```
---------------------------- MODULE A1Req ----------------------------
EXTENDS ActOps
VARIABLES Ev, st
-----------------------------------------------------------------------
vars ≜ ⟨Ev, st⟩
in ≜ {⟨“a”⟩}
out ≜ {⟨“c”⟩}
ext ≜ in ∪ out
States ≜ {“s0”, “s1”, “s2”}
StInit ≜ st = “s0”
Init ≜ ActInit(Ev) ∧ StInit
TypeInvariant ≜ ∧ Ev ∈ {“0”, “1”} × (ext ∪ {⟨⟩})
                ∧ st ∈ States
-----------------------------------------------------------------------
InActionA ≜ ∧ Act(Ev, ⟨“a”⟩)
            ∧ st′ = IF st = “s0” THEN “s1”
                                 ELSE st
OutActionC ≜ ∧ Act(Ev, ⟨“c”⟩)
             ∧ (st = “s1”) ∧ (st′ = “s2”)
Next ≜ InActionA ∨ OutActionC
Spec ≜ Init ∧ □[Next]_vars
-----------------------------------------------------------------------
THEOREM Spec ⇒ □TypeInvariant
=======================================================================
```

Fig. 4. Module *A1Req*.



Error-Trace

Name	Value
▲ <Initial predicate>	State (num = 1)
■ st	"s0"
■ Ev	<<"0", << >>>>
▲ <Action line 44, col 9	State (num = 2)
■ st	"s1"
■ Ev	<<"1", <<"a">>>>
▲ <Stuttering>	State (num = 3)

Fig. 5. An error trace in case of wrong fair trace specification of $A1$.

TLC printed the error trace shown in Fig. 5. It represents exactly the trace $\langle a\rangle$ of $A1$ we claim not to be allowed by the wrong specification of $\mathit{fairtraces}(A1)$ in Subsection 4.4. Indeed, in this way TLC reported that the implication is not valid because trace $\langle a\rangle$ is in the set of traces of $A1Req$, but not in the specified set of fair traces of $A1$, meaning $\neg(\mathit{traces}(A1Req) \subseteq \mathit{fairtraces}(A1))$. When we applied the right *LivenessH* formula (the uncommented one in Fig. 3), TLC reported that the implication was valid. As expected, TLC answered that implication $\mathit{SpecFT} \Rightarrow \mathit{ReqSpec}$ was valid for *SpecFT* with both versions of *LivenessH*. Clearly, if using the wrong formula *LivenessH*, theorem $\mathit{SpecT} \Rightarrow \mathit{SpecFT}$, which asserts that $\mathit{traces}(A1) \subseteq \mathit{fairtraces}(A1)$, is also not valid. TLC generates the same error trace as above.

According to Definition 3, the matching of input and output actions of the automata should be checked before establishing an implementation relation between them. It is easy to see that $A1$ and $A1Req$ have the same sets of input and output actions. Nevertheless, module $A1$ in Fig. 3 illustrates how it is possible to check this with TLC. This could be useful for automata with large action sets. The TLA$^+$ ASSUME statement expresses the matching of the input and output action sets. Normally, an ASSUME statement is used for stating assumptions about the constant parameters of a module, but it is only important that it does not contain any variables. A verification in TLC is always carried out for a particular module. When TLC is called for a verification, it first checks all the assumptions of this module. It continues only if all of them are valid, i.e. if the action sets match in our case.

Another possibility to check the matching of the action sets would be to specify it as a part of the property to be proved in the theorems expressing the trace inclusion. For example, instead of the assumption and the fourth theorem listed in $A1$, the currently commented *CSpec* formula and the theorem including it could be used in module $A1$.

6. Discussion and Conclusions

The main contribution of this paper is the proposal of how to specify fair traces of fair I/O automata and how to verify the implementation relations, which is the 'standard' approach to the I/O automata verification, by using TLA. In this way, the specification and verification of the external behaviour of fair I/O automata can be carried out strictly formally in a single language—TLA or more precisely, TLA$^+$—and by using its tool support. In contrast to Kapus (2002, 2005), in this paper we specified the recording of action labels of I/O automata in such a way that TLA specifications of the behaviour of finite-state I/O automata cannot give infinitely many reachable states and can, therefore, be automatically verified with the TLC model checker. We defined action Act in a similar way to action $ChanOp$ which is used by Ladkin *et al.* (1999) for the representation of message passing over channels represented by variables.

Funiak (2001) tried to apply TLA$^+$ for ordinary (i.e. not fair) I/O automata. However, he did not use TLA$^+$ directly for that model. He translated I/O automata into a state-based model by ignoring the I/O automata action labels. Müller (1998) and Søgaard-Andersen *et al.* (1993) considered the specification of fair I/O automata and used some kind of linear-time temporal logic. They, however, applied the temporal logic only for the specification of fairness conditions and required temporal properties of executions. The safety part of automata was specified in a semi-formal precondition/effect style. The IOA language (Garland and Lynch, 2000; Garland *et al.*, 2003) enables formal description of ordinary I/O automata, as well as the specification of required implementation relations and state invariants. For finite-state IOA programs, a model checker exists. However, as mentioned in the introduction, verification by theorem proving cannot be carried out directly in IOA.

Systems are often developed by using stepwise refinement. First, a top-level specification is written as a composition of I/O automata (see, e.g., Müller, 1998), and a more detailed system specification is then developed by a refinement of every component of the composition. The refinement step has to be verified by checking whether the detailed system specification implements the top-level one. Generally, compositional verification is possible by establishing an implementation relation between every component in the refined specification and its counterpart in the top-level one separately. Our approach can be used for the specification of the external behaviour of the components and the verification of trace inclusion for each one.

Sometimes, however, a system is given as a composition of I/O automata, but its requirement specification as a single one. To be able to verify trace inclusion by using our approach in such a case, a single I/O automaton should first be obtained from the components of the composition and then its traces specified by using our approach by not recording internal actions of the system. The composition of I/O automata in TLA is out of the scope of this paper, but a TLA specification of its executions could be obtained from the TLA specifications of the executions of its components in a similar way to Kapus (2009). We shall deal with the TLA specification and verification of the composition of fair I/O automata in a forthcoming paper.

TLAPS, the TLA$^+$ Proof System (2014), is being developed for writing and mechanically checking TLA$^+$ proofs entirely on the level of TLA$^+$, without the need for the user to know the mechanisms of the general-purpose theorem provers which are used behind and have also been employed for I/O automata (e.g. Garland *et al.*, 2003; Müller, 1998). We will try to apply TLAPS for the verification of I/O automata in order to test its capabilities.

Acknowledgments. The author acknowledges the support of the Slovenian Research Agency (ARRS) through program P2-0069.

References

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

Armendáriz-Iñigo, J.E., González de Mendívil, J.R., Garitagoitia, J.R., Muñoz-Escoí, F.D. (2009). Correctness proof of a database replication protocol under the perspective of the I/O automaton model. *Acta Informatica*, 46(4), 297–330.

Funiak, S. (2001). *Model checking IOA programs with TLC.* Summer 2001 report. http://theory.lcs.mit.edu/tds/papers/Funiak/report.ps.

Garland, S.J., Lynch, N. (2000). Using I/O automata for developing distributed systems. In: Leavens, G.T., Sitaraman, M. (Eds.), *Foundations of Component-Based Systems.* Cambridge University Press, Cambridge, pp. 285–311.

Garland, S.J., Lynch, N.A., Tauber, J.A., Vaziri, M. (2003). *IOA User Guide and Reference Manual.* Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Kapus, T. (2002). Modelling of agent computations using the temporal logic of actions. In: Rožić, N., Begušić, D. (Eds.), *Proceedings of the 10th International Conference on Software, Telecommunications and Computer Networks (SoftCOM 2002).* Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture, Split, pp. 345–349.

Kapus, T. (2005). Mobile agent system specification using the temporal logic of actions. In: Kokol, P. (Ed.), *Proceedings of the IASTED International Conference on Software Engineering (SE 2005).* ACTA Press, Zurich, pp. 319–324.

Kapus, T. (2009). Using mobile TLA as a logic for dynamic I/O automata. *IEICE Transactions on Information and Systems*, E92-D, 1515–1522.

Krukow, K., Nielsen, M. (2007). Trust structures: Denotational and operational semantics. *International Journal of Information Security*, 6(2–3) 153–181.

Ladkin, P., Lamport, L., Olivier, B., Roegel, D. (1999). Lazy caching in TLA. *Distributed Computing*, 12(2–3) 151–174.

Lamport, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), 872–943.

Lamport, L. (2002). *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison–Wesley Professional. http://research.microsoft.com/en-us/um/people/lamport/tla/book.html.

Lamport, L. (2014). *TLA^{+2}: a preliminary guide*. http://research.microsoft.com/en-us/um/people/lamport/tla/tla2-guide.pdf.

Lynch, N.A., Tuttle, M.R. (1989). An introduction to input/output automata. *CWI Quarterly*, 2(3), 219–246.

Merz, S. (2003). On the logic of TLA$^+$. *Computing and Informatics*, 22(3–4), 351–379.

Mitra, S., Kumar, R., Basu, S. (2007). Automated choreographer synthesis for Web services composition using I/O automata. In: *Proceedings of the IEEE International Conference on Web Services (ICWS 2007)*. IEEE Computer Society Press, Los Alamitos, CA, pp. 364–371.

Müller, O. (1998). *A verification environment for I/O automata based on formalized meta-theory*. Technical report, TUM-I9822, Technische Universität München.

Reynolds, M. (1998). *Changing nothing is sometimes doing something: fairness in extensional semantics*. Technical report tr-98-02, King's College, London.

Romijn, J.M.T., Vaandrager, F.W. (1996). A note on fairness in I/O automata. *Information Processing Letters*, 59(5), 245–250.

Segala, R., Gawlick, R., Søgaard-Andersen, J., Lynch, N. (1998). Liveness in timed and untimed systems. *Information and Computation*, 141(2), 119–171.

Søgaard-Andersen, J.F., Lynch, N.A., Lampson, B.W. (1993). *Correctness of communication protocols—a case study*. Technical report MIT/LCS/TR-589, Laboratory for Computer Science, MIT, Cambridge, MA, and Technical report ID-TR: 1993-129, Department of Computer Science, Technical University of Denmark, Lingby.

TLA$^+$ proof system. http://tla.msr-inria.inria.fr/tlaps/content/Home.html.

The TLA Toolbox. http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html.

Trček, D. (2014). Computational trust management, QAD, and its applications. *Informatica*, 25(1), 139–154.

Tuttle, M.R. (1987). *Hierarchical correctness proofs for distributed algorithms*. Master's thesis, Technical report MIT/LCS/TR-387, Massachusetts Institute of Technology, Cambridge, MA.

T. Kapus is a professor at the Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia. She teaches courses on communications networks, formal methods, and programming. Her primary research interest lies in formal methods for the specification and verification of reactive systems.

Temporalinės veiksmų logikos naudojimas teisingos įvesties/išvesties automato išorinei elgsenai specifikuoti ir tikrinti

Tatjana KAPUS

Teisingos įvesties/išvesties automatas – tai būsenų mašina grindžiamas modelis, skirtas specifikuoti ir tikrinti reaktyviasias ir besivaržančiasias sistemas. Tikrinant, paprastai domimasi tik sąveikų sekomis, pateikiamomis teisingos įvesties/išvesties automato savo aplinkai. Tos trasos vadinamos teisingomis trasomis. Tipinė tikrinimo procedūra – tai įrodymas, kad tikrinamos sistemos vykdymo trasos gali būti įterptos į teisingos įvesties/išvesties automato generuojamas teisingas sekas. Straipsnyje pateiktas paprastas teisingų trasų specifikavimo būdas ir parodyta kaip, panaudojant temporalinę veiksmų logiką, porai teisingos įvesties/išvesties automatų sukurti įterpties į teisingas trasas ryšį.