TLA in Pictures

Leslie Lamport

Abstract— **Predicate-action diagrams, which are similar to standard state-transition diagrams, are precisely defined as formulas of TLA (the Temporal Logic of Actions). We explain how these diagrams can be used to describe aspects of a specification—and those descriptions then proved correct—even when the complete specification cannot be written as a diagram. We also use the diagrams to illustrate proofs.**

Keywords— **Concurrency, specification, state-transition diagrams, temporal logic.**

I. Introduction

Pictures aid understanding. A simple flowchart is easier to understand than the equivalent programming-language text. However, complex pictures are confusing. A large, spaghetti-like flowchart is harder to understand than a properly structured program text.

Pictures are inadequate for specifying complex systems, but they can help us understand particular aspects of a system. For a picture to provide more than an informal comment, there must be a formal connection between the complete specification and the picture. The assertion that the picture is a correct description of (some aspect of) the system must be a precise mathematical statement.

We use TLA (the Temporal Logic of Actions) to specify systems. In TLA, a specification is a logical formula describing all possible correct behaviors of the system. As an aid to understanding TLA formulas, we introduce here a type of picture called a *predicate-action diagram*. These diagrams are similar to the various kinds of state-transition diagrams that have been used for years to describe systems, starting with Mealy and Moore machines [1], [2]. We relate these pictures to TLA specifications by interpreting a predicate-action diagram as a TLA formula. A diagram denoting formula D is a correct description of a system with specification S iff (if and only if) S implies D. We therefore provide a precise statement of what it means for a diagram to describe a specification.

We use predicate-action diagrams in three ways that we believe are new for a precisely defined formal notation:

- To describe aspects of a specification even when it is not feasible to write the complete specification as a diagram.
- To draw different diagrams that provide complementary views of the same system.
- To illustrate formal correctness proofs.

Section II is a brief review of TLA; a more leisurely introduction to TLA appears in [3]. Section III describes predicate-action diagrams, using an n-input Muller C-element as an example. It shows how diagrams are used to describe aspects of a complete specification, and to provide complementary views of a system. Section IV gives another example of how predicate-action diagrams are used to describe a system, and shows how they are used to illustrate a proof.

II. TLA

We now describe the syntax and semantics of TLA. The description is illustrated with the formulas defined in Figure 1. (The symbol $\triangleq$ means *equals by definition.*)

We assume an infinite set of variables (such as x and y) and a class of semantic values. Our variables are the flexible variables of temporal logic, which are analogous to variables in a programming language. TLA also includes the rigid variables of predicate logic, which are analogous to constant parameters of a program, but we ignore them here. The class of values includes numbers, strings, sets, and functions.

A *state* is an assignment of values to variables. A *behavior* is an infinite sequence of states. Semantically, a TLA formula is true or false of a behavior. Syntactically, TLA formulas are built up from state functions using Boolean operators ($\neg$, $\wedge$, $\vee$, $\Rightarrow$ [implication], and $\equiv$ [equivalence]) and the operators $'$ and $\Box$, as described below. TLA also has a hiding operator $\exists\!\!\!\exists$, which we do not use here.

A *state function* is a nonBoolean expression built from variables, constants, and constant operators. Semantically, it assigns a value to each state—for example $x + 1$ assigns to state s one plus the value that s assigns to the variable x. A *state predicate* (often called just a *predicate*) is a Boolean expression built from variables, constants, and constant operators such as $+$. Semantically, it is true or false for a state—for example the predicate $Init_\Phi$ is true of state s iff s assigns the value zero to both x and y.

An *action* is a Boolean expression containing primed and unprimed variables. Semantically, an action is true or false of a pair of states, with primed variables referring to the second state—for example, action $\mathcal{M}_1$ is true for $\langle s, t\rangle$ iff the value that state t assigns to x equals one plus the value that state s assigns to x, and the values assigned to y by states s and t are equal. A pair of states satisfying an action $\mathcal{A}$ is called an $\mathcal{A}$ *step*. Thus, an $\mathcal{M}_1$ step is one that increments x by one and leaves y unchanged.

If f is a state function or state predicate, we write f' for the expression obtained by priming all the variables of f. For example $(x+1)'$ equals $x'+1$, and $Init'_\Phi$ equals $(x' = 0) \wedge (y' = 0)$. For an action $\mathcal{A}$ and a state function v,

$$
\begin{aligned}
Init_\Phi &\triangleq (x = 0) \wedge (y = 0)\\
\mathcal{M}_1 &\triangleq (x' = x + 1) \wedge (y' = y)\\
\mathcal{M}_2 &\triangleq (y' = y + 1) \wedge (x' = x)\\
\mathcal{M} &\triangleq \mathcal{M}_1 \vee \mathcal{M}_2\\
\Phi &\triangleq Init_\Phi \wedge \Box[\mathcal{M}]_{\langle x,\, y\rangle} \wedge \mathrm{WF}_{\langle x,\, y\rangle}(\mathcal{M}_1) \wedge \mathrm{WF}_{\langle x,\, y\rangle}(\mathcal{M}_2)
\end{aligned}
$$

Fig. 1. The TLA formula Φ describing a simple program that repeatedly increments x or y.

we define $[\mathcal{A}]_v$ to equal $\mathcal{A} \vee (v' = v)$, so a $[\mathcal{A}]_v$ step is either an $\mathcal{A}$ step or a step that leaves the value of v unchanged. Thus, a $[\mathcal{M}_1]_{\langle x,y\rangle}$ step is one that increments x by one and leaves y unchanged, or else leaves the ordered pair $\langle x, y\rangle$ unchanged. Since a tuple is unchanged iff each component is unchanged, a $[\mathcal{M}_1]_{\langle x,y\rangle}$ step is one that increments x by one and leaves y unchanged, or else leaves both x and y unchanged. We define $\langle\mathcal{A}\rangle_v$ to equal $\mathcal{A} \wedge (v' \neq v)$, so an $\langle\mathcal{M}_1\rangle_{\langle x,y\rangle}$ step is an $\mathcal{M}_1$ step that changes x or y. Since an $\mathcal{M}_1$ step leaves y unchanged, an $\langle\mathcal{M}_1\rangle_{\langle x,y\rangle}$ step is a step that increments x by 1, changes the value of x, and leaves y unchanged.

We say that an action $\mathcal{A}$ is *enabled* in state s iff there exists a state t such that $\langle s, t\rangle$ is an $\mathcal{A}$ step. For example, $\mathcal{M}_1$ is enabled iff it is possible to take a step that increments x by one, changes x, and leaves y unchanged. Since $x + 1 \neq x$ for any natural number x, action $\langle\mathcal{M}_1\rangle_{\langle x,y\rangle}$ is enabled in any state in which x is a natural number. If $\infty + 1$ equals ∞, then $\langle\mathcal{M}_1\rangle_{\langle x,y\rangle}$ is not enabled in a state in which x equals ∞.

A TLA formula is true or false of a behavior. A predicate is true of a behavior iff it is true of the first state. An action is true of a behavior iff it is true of the first pair of states. As usual in temporal logic, if F is a formula then $\Box F$ is the formula meaning that F is always true. Thus, $\Box \mathit{Init}_\phi$ is true of a behavior iff x and y equal zero for every state in the behavior. The formula $\Box[\mathcal{M}]_{\langle x,y\rangle}$ is true of a behavior iff each step (pair of successive states) of the behavior is a $[\mathcal{M}]_{\langle x,y\rangle}$ step.

Using $\Box$ and "enabled" predicates, we can define fairness operators WF and SF. The *weak fairness* formula $\mathrm{WF}_v(\mathcal{A})$ asserts of a behavior that there are infinitely many $\langle\mathcal{A}\rangle_v$ steps, or there are infinitely many states in which $\langle\mathcal{A}\rangle_v$ is not enabled. In other words, $\mathrm{WF}_v(\mathcal{A})$ asserts that if $\langle\mathcal{A}\rangle_v$ becomes enabled forever, then infinitely many $\langle\mathcal{A}\rangle_v$ steps occur. The *strong fairness* formula $\mathrm{SF}_v(\mathcal{A})$ asserts that either there are infinitely many $\langle\mathcal{A}\rangle_v$ steps, or there are only finitely many states in which $\langle\mathcal{A}\rangle_v$ is enabled. In other words, $\mathrm{SF}_v(\mathcal{A})$ asserts that if $\langle\mathcal{A}\rangle_v$ is enabled infinitely often, then infinitely many $\langle\mathcal{A}\rangle_v$ steps occur.

The standard form of a TLA specification is $\mathit{Init} \wedge \Box[\mathcal{N}]_v \wedge L$, where Init is a predicate, $\mathcal{N}$ is an action, v is a state function, and L is a conjunction of fairness conditions. This formula asserts of a behavior that (i) Init is true for the initial state, (ii) every step of the behavior is an $\mathcal{N}$ step or leaves v unchanged, and (iii) L holds. Formula Φ of Figure 1 is in this form, asserting that (i) initially x and y both equal zero, (ii) every step either increments x by one and leaves y unchanged, increments y by one and leaves x unchanged, or leaves both x and y unchanged, and (iii) the fairness condition $\mathrm{WF}_{\langle x,\,y\rangle}(\mathcal{M}_1) \wedge \mathrm{WF}_{\langle x,\,y\rangle}(\mathcal{M}_2)$ holds. Formula $\mathrm{WF}_{\langle x,\,y\rangle}(\mathcal{M}_1)$ asserts that there are infinitely many $\langle\mathcal{M}_1\rangle_{\langle x,y\rangle}$ steps or $\langle\mathcal{M}_1\rangle_{\langle x,y\rangle}$ is infinitely often not enabled. Since (i) and (ii) imply that x is always a natural number, $\langle\mathcal{M}_1\rangle_{\langle x,y\rangle}$ is always enabled. Hence, $\mathrm{WF}_{\langle x,\,y\rangle}(\mathcal{M}_1)$ implies that there are infinitely many $\langle\mathcal{M}_1\rangle_{\langle x,y\rangle}$ steps, so x is incremented infinitely often. Similarly, $\mathrm{WF}_{\langle x,\,y\rangle}(\mathcal{M}_2)$ implies that y is incremented infinitely often. Putting this all together, we see that Φ is true of a behavior iff (i) x and y are initially zero, (ii) every step increments either x or y by one and leaves the other unchanged or else leaves both x and y unchanged, and (iii) both x and y are incremented infinitely many times.

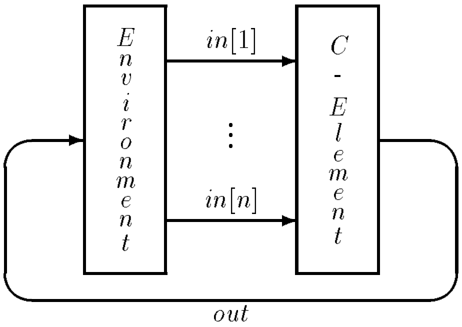


Fig. 2. A Muller C-element.

The formula $\mathit{Init} \wedge \Box[\mathcal{N}]_v$ is a safety property [4]. It describes what steps are allowed, but it does not require anything to happen. (The formula is satisfied by a behavior satisfying the initial condition in which no variables ever change.) Fairness conditions are used to specify that something must happen.

III. Predicate-Action Diagrams

A. An Example

We take as an example a Muller C-element [5]. This is a circuit with n binary inputs $in[1], \ldots, in[n]$ and one binary output out, as shown in Figure 2. As the figure indicates, we are considering the closed system consisting of the C-element together with its environment. Initially, all the inputs and the output are equal. The output becomes 0 when all the inputs are 0, and it becomes 1 when all the inputs are 1. After an input changes, it must remain stable until the output changes.

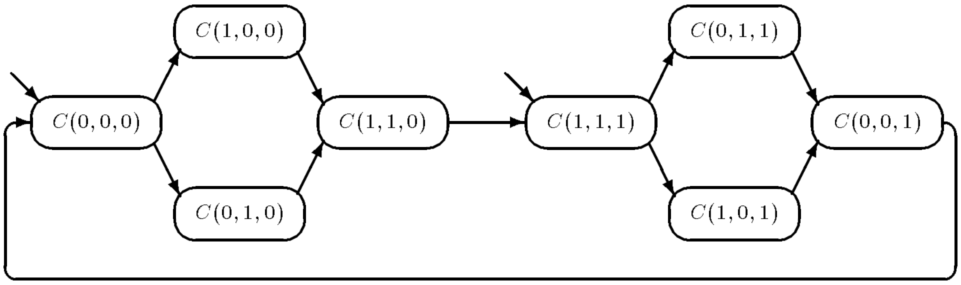
The behavior of a 2-input C-element and its environment is described by the predicate-action diagram of Figure 3(a), where C is defined by

$$C(i, j, k) \triangleq (in[1] = i) \wedge (in[2] = j) \wedge (out = k)$$

The short arrows, with no originating node, identify the nodes labeled $C(0, 0, 0)$ and $C(1, 1, 1)$ as initial nodes. They indicate that the C-element starts in a state satisfying $C(0, 0, 0)$ or $C(1, 1, 1)$. The arrows connecting nodes indicate possible state transitions. For example, from a state satisfying $C(1, 1, 1)$, it is possible for the system to go to a state satisfying either $C(0, 1, 1)$ or $C(1, 0, 1)$. More precisely, these arrows indicate all steps in which the triple $\langle in[1], in[2], out\rangle$ changes—that is, transitions in which at least one of $in[1]$, $in[2]$, and out changes. Steps that change other variables—for example, variables representing circuit elements inside the environment—but leave $\langle in[1], in[2], out\rangle$ unchanged are also possible.

The predicate-action diagram of Figure 3(a) looks like a standard state-transition diagram. However, we interpret it formally not as a conventional state machine, but as the

(a) A predicate-action diagram.



(b) The corresponding TLA formula.

$$
\begin{array}{l}
\wedge\ C(0,0,0) \vee C(1,1,1) \\
\wedge\ \Box[C(0,0,0) \Rightarrow C(1,0,0)' \vee C(0,1,0)']_{\langle in[1], in[2], out\rangle} \\
\wedge\ \Box[C(1,0,0) \Rightarrow C(1,1,0)']_{\langle in[1], in[2], out\rangle} \\
\quad \ldots \\
\wedge\ \Box[C(0,0,1) \Rightarrow C(0,0,0)']_{\langle in[1], in[2], out\rangle}
\end{array}
$$

Fig. 3. Predicate-Action diagram of $\langle in[1], in[2], out\rangle$ for a 2-input C-element, and the corresponding TLA formula.

TLA formula of Figure 3(b).[1] This formula has the form $Init \wedge \bigwedge_o F_o$, where $Init$ is a state predicate and there is one conjunct F_o for each node o. The predicate $Init$ is $C(0,0,0) \vee C(1,1,1)$. Each F_o describes the possible state changes starting from a state described by node o. For example, the formula F_o for the node labeled $C(1,1,0)$ is

$$\Box[C(1,1,0) \Rightarrow C(1,1,1)']_{\langle in[1], in[2], out\rangle}$$

A predicate-action diagram represents a safety property; it does not include any fairness conditions.

Figure 3(a) is a reasonable way to describe a 2-input C-element. However, the corresponding diagram for a 3-input C-element would be quite complicated; and there is no way to draw such a diagram for an n-input circuit. The general specification is written directly as a TLA formula in Figure 4. The array of inputs is represented formally by a variable in whose value is a function with domain $\{1, \ldots, \mathsf{n}\}$, where square brackets denote function application. (Formally, n is a rigid variable—one whose value is constant throughout a behavior.) We introduce two pieces of notation for representing functions:

- $[i \in S \mapsto e(i)]$ denotes the function f with domain S such that $f[i]$ equals $e(i)$ for every i in S.
- $[f$ **except** $![i] = e]$ denotes the function g that is the same as f except that $g[i]$ equals e.

The formulas defined in Figure 4 have the following interpretation.

$Init_C$ A state predicate asserting that out is either 0 or 1, and that in is the function with domain $\{1, \ldots, \mathsf{n}\}$ such that $in[i]$ equals out for all i in its domain.

$Input(i)$ An action that is enabled iff $in[i]$ equals out. It complements $in[i]$, leaves $in[j]$ unchanged for $j \neq i$, and leaves out unchanged. (The symbol i is a parameter.)

$Output$ An action that is enabled iff all the $in[i]$ are different from out. It complements out and leaves in unchanged.

[1] A list of formulas bulleted by $\wedge$ or $\vee$ denotes their conjunction or disjunction; $\wedge$ and $\vee$ are also used as ordinary infix operators.

$$
\begin{array}{lcl}
Init_C & \triangleq & \wedge\ out \in \{0,1\} \\
 & & \wedge\ in = [i \in \{1, \ldots, \mathsf{n}\} \mapsto out] \\
Input(i) & \triangleq & \wedge\ in[i] = out \\
 & & \wedge\ in' = [in \ \textbf{except}\ ![i] = 1 - in[i]] \\
 & & \wedge\ out' = out \\
Output & \triangleq & \wedge\ \forall i \in \{1, \ldots, \mathsf{n}\} : in[i] \neq out \\
 & & \wedge\ out' = 1 - out \\
 & & \wedge\ in' = in \\
Next & \triangleq & Output \vee \exists\, i \in \{1, \ldots, \mathsf{n}\} : Input(i) \\
\Pi_C & \triangleq & Init_C \wedge \Box[Next]_{\langle in, out\rangle} \wedge \mathrm{WF}_{\langle in, out\rangle}(Output)
\end{array}
$$

Fig. 4. A TLA specification of an n-input C-element.

$Next$ An action that is the disjunction of $Output$ and all the $Input(i)$ actions, for $i \in \{1, \ldots, \mathsf{n}\}$. Thus, a $Next$ step is either an $Output$ step or an $Input(i)$ step for some input line i.

Π_C A temporal formula that is the specification of the C-element (together with its environment). It asserts that (i) $Init_C$ holds initially, (ii) every step is either a $Next$ step or else leaves $\langle in, out\rangle$ unchanged, and (iii) $Output$ cannot be enabled forever without an *Output* step occurring. The fairness condition (iii) requires the output to change if all the inputs have; inputs are not required to change. (Since predicate-action diagrams describe only safety properties, the fairness condition is irrelevant to our explanation of the diagrams.)

The specification Π_C is short and precise. However, it is not as reader-friendly as a predicate-action diagram. We therefore use diagrams to help explain the specification, beginning with the predicate-action diagram of Figure 5. It is a diagram *of* the state function $\langle in[i], out\rangle$, meaning that it describes transitions that change $\langle in[i], out\rangle$. It is a diagram *for* the formula Π_C, meaning that it represents a formula that is implied by Π_C. The diagram shows the synchronization between the C-element's ith input and its output.

We can draw many different predicate-action diagrams

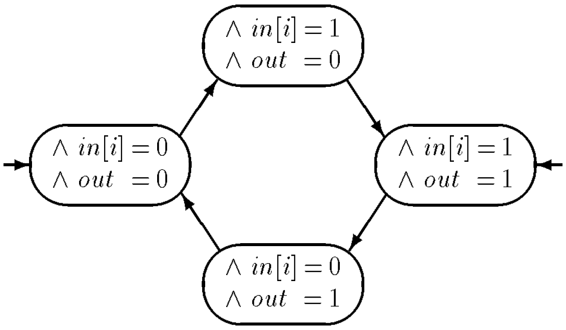


Fig. 5. A predicate-action diagram of $\langle in[i], out\rangle$ for the specification Π_C of an n-input C-element, where $1 \leq i \leq \mathsf{n}$.

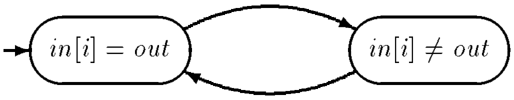


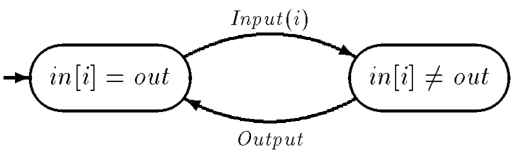
Fig. 6. Another predicate-action diagram of $\langle in[i], out\rangle$ for Π_C, where $1 \leq i \leq \mathsf{n}$.

for the same specification. Figure 6 shows another diagram of $\langle in[i], out\rangle$ for Π_C. It is simpler than the one in Figure 5, but it contains less information. It does not indicate that the values of $in[i]$ and out are always 0 or 1, and it does not show which variable is changed by each transition. The latter information is added in the diagram of Figure 7(a), where each transition is labeled with an action. The label $Input(i)$ on the left-to-right arrow indicates that a transition from a state satisfying $in[i] = out$ to a state satisfying $in[i] \neq out$ is an $Input(i)$ step. This diagram represents the TLA formula of Figure 7(b).

Even more information is conveyed by a predicate-action diagram of $\langle in, out\rangle$, which also shows transitions that leave $in[i]$ and out unchanged but change $in[j]$ for some $j \neq i$. Such a diagram is drawn in Figure 8(a). Figure 8(b) gives the corresponding TLA formula.

There are innumerable predicate-action diagrams that can be drawn for a specification. Figure 9 shows yet another diagram for the C-element specification Π_C. Since we are not relying on these diagrams as our specification, but simply to help explain the specification, we can show as much or as little information in them as we wish. We can draw multiple diagrams to illustrate different aspects of a system. Actual specifications are written as TLA formulas, which are much more expressive than pictures.

(a) A predicate-action diagram of $\langle in[i], out\rangle$.

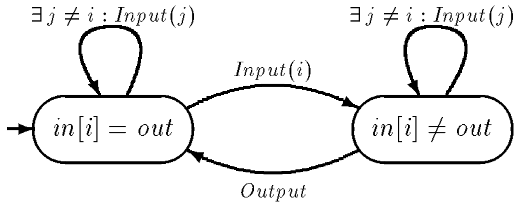


(b) The corresponding TLA formula.

$$\begin{aligned}
&\wedge\ in[i] = out\\
&\wedge\ \Box[(in[i] = out) \Rightarrow Input(i) \wedge (in'[i] \neq out')]_{\langle in[i], out\rangle}\\
&\wedge\ \Box[(in[i] \neq out) \Rightarrow Output \wedge (in'[i] = out')]_{\langle in[i], out\rangle}
\end{aligned}$$

Fig. 7. A more informative predicate-action diagram of $\langle in[i], out\rangle$ for Π_C, and the corresponding TLA formula.

(a) A predicate-action diagram of $\langle in, out\rangle$.



(b) The corresponding TLA formula.

$$\begin{aligned}
&\wedge\ in[i] = out\\
&\wedge\ \Box\left[\begin{array}{l}(in[i] = out) \Rightarrow\\ \quad\begin{pmatrix}\vee\ Input(i) \wedge (in'[i] \neq out')\\ \vee\ (\exists j \neq i : Input(j)) \wedge (in'[i] = out')\end{pmatrix}\end{array}\right]_{\langle in, out\rangle}\\
&\wedge\ \Box\left[\begin{array}{l}(in[i] \neq out) \Rightarrow\\ \quad\begin{pmatrix}\vee\ Output \wedge (in'[i] = out')\\ \vee\ (\exists j \neq i : Input(j)) \wedge (in'[i] \neq out')\end{pmatrix}\end{array}\right]_{\langle in, out\rangle}
\end{aligned}$$

Fig. 8. A predicate-action diagram of $\langle in, out\rangle$ for Π_C, and the corresponding TLA formula, where $1 \leq i \leq \mathsf{n}$.

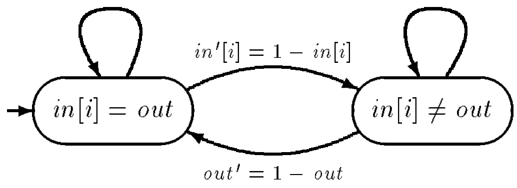


Fig. 9. Yet another predicate-action diagram of $\langle in, out\rangle$ for Π_C.

B. A Formal Treatment

B.1 Definition

We first define precisely the TLA formula represented by a diagram. Formally, a predicate-action diagram consists of a directed graph, with a subset of the nodes identified as initial nodes, where each node is labeled by a state predicate and each edge is labeled by an action. We assume a given diagram of a state function v and introduce the following notation.

N	The set of nodes.
I	The set of initial nodes.
$E(n)$	The set of edges originating at node n.
$d(e)$	The destination node of edge e.
P_n	The predicate labeling node n.
$\mathcal{E}_e$	The action labeling edge e.

The formula Δ represented by the diagram is defined as follows.

$$\begin{aligned}
Init_\Delta &\triangleq \exists\, n \in I : P_n\\
\mathcal{A}_n &\triangleq \exists\, e \in E(n) : \mathcal{E}_e \wedge P'_{d(e)}\\
\Delta &\triangleq Init_\Delta \wedge \forall n \in N : \Box[P_n \Rightarrow \mathcal{A}_n]_v
\end{aligned}$$

When no explicit label is attached to an edge e, we take $\mathcal{E}_e$ to be *true*. When no set of initial nodes is explicitly indicated, we take I to be N. With the usual convention for quantification over an empty set, $\mathcal{A}_n$ is defined to equal *false* if there are no edges originating at node n.

B.2 Another Interpretation

Another possible interpretation of the predicate-action diagram is the formula $\widehat{\Delta}$, defined by

$$\widehat{\Delta} \ \triangleq\ \mathit{Init}_\Delta \wedge \Box[\exists\, n \in N : P_n \wedge \mathcal{A}_n]_v$$

This is perhaps a more obvious interpretation—especially if the diagram is viewed as a description of a next-state relation. We now show that Δ always implies $\widehat{\Delta}$, and that the converse implication holds if the predicates labeling the nodes are disjoint.

(A) Δ implies $\widehat{\Delta}$.

Proof: A simple invariance proof, using rule INV1 of [3, Figure 5, page 888], shows that Δ implies $\Box(\exists\, n \in N : P_n)$. We then have:

$$\begin{array}{lll}
\Delta & \triangleq & \mathit{Init}_\Delta \ \wedge\ \forall\, n \in N : \Box[P_n \Rightarrow \mathcal{A}_n]_v \\
& \equiv & \mathit{Init}_\Delta \ \wedge\ \Box([\exists\, n \in N : P_n]_v) \\
& & \quad \wedge\ \forall\, n \in N : \Box[P_n \Rightarrow \mathcal{A}_n]_v \\
& & \quad [\text{because } \Delta \text{ implies } \Box(\exists\, n \in N : P_n)\,] \\
& \equiv & \mathit{Init}_\Delta \ \wedge\ \Box[(\exists\, n \in N : P_n) \\
& & \qquad\qquad \wedge\ \forall\, n \in N : (P_n \Rightarrow \mathcal{A}_n)]_v \\
& & \quad [\text{because } \Box \text{ distributes over conjunction and } \forall, \text{ and} \\
& & \quad\ [X]_v \wedge \forall n \in N : [Y_n]_v \text{ is equivalent to} \\
& & \quad\ [X \wedge \forall n \in N : Y_n]_v\,] \\
& \Rightarrow & \mathit{Init}_\Delta \ \wedge\ \Box[\exists\, n \in N : P_n \wedge \mathcal{A}_n]_v \\
& & \quad [\text{by predicate logic, since } \mathcal{B} \Rightarrow \mathcal{C} \text{ implies } \Box[\mathcal{B}]_v \Rightarrow \Box[\mathcal{C}]_v\,] \\
& \triangleq & \widehat{\Delta}
\end{array}$$

(B) If $\neg(P_m \wedge P_n)$ holds for all m, n in N with $m \neq n$, then $\widehat{\Delta}$ implies Δ.

Proof: By propositional logic, the hypothesis implies

$$(\exists\, n \in N : P_n \wedge \mathcal{A}_n) \ \Rightarrow\ (\forall\, n \in N : P_n \Rightarrow \mathcal{A}_n)$$

The result then follows from simple temporal reasoning, essentially by the reverse of the string of equivalences and implication used to prove (A). □

We usually label the nodes of a predicate-action diagram with disjoint predicates, in which case (A) and (B) imply that the interpretations Δ and $\widehat{\Delta}$ are equivalent. Diagrams with nondisjoint node labels may occasionally be useful; Δ is the more convenient interpretation of such diagrams.

C. Proving a Predicate-Action Diagram

Saying that a diagram is a predicate-action diagram for a specification Π asserts that Π implies the formula Δ represented by the diagram. Formula Π will usually have the form $\mathit{Init}_\Pi \wedge \Box[\mathcal{M}]_u \wedge L$, where L is a fairness condition. Formula Δ equals $\mathit{Init}_\Delta \wedge \forall n \in N : \Box[P_n \Rightarrow \mathcal{A}_n]_v$. To prove $\Pi \Rightarrow \Delta$, we prove:

1. $\mathit{Init}_\Pi \ \Rightarrow\ \mathit{Init}_\Delta$
2. $\mathit{Init}_\Pi \wedge \Box[\mathcal{M}]_u \ \Rightarrow\ \Box[P_n \Rightarrow \mathcal{A}_n]_v$, for each node n.

The first condition is an assertion about predicates; it is generally easy to prove. To prove the second condition, one usually finds an invariant Inv such that $\mathit{Init}_\Pi \wedge \Box[\mathcal{M}]_u$ implies $\Box \mathit{Inv}$, so Π implies $\Box[\mathcal{M} \wedge \mathit{Inv}]_u$. The second condition is then proved by showing that $[\mathcal{M} \wedge \mathit{Inv}]_u$ implies $[P_n \Rightarrow \mathcal{A}_n]_v$, for each node n. Usually, u and v are tuples and every component of v is a component of u, so $u' = u$ implies $v' = v$. In this case, one need show only that $\mathcal{M} \wedge \mathit{Inv}$ implies $[P_n \Rightarrow \mathcal{A}_n]_v$, for each n. By definition of $\mathcal{A}_n$, this means proving

$$P_n \wedge \mathcal{M} \wedge \mathit{Inv} \ \Rightarrow\ (\exists\, m \in E(n) : \mathcal{E}_m \wedge P'_{d(m)}) \vee (v' = v)$$

for each node n. This formula asserts that an $\mathcal{M}$ step that starts with P_n and Inv true and changes v is an $\mathcal{E}_m$ step that ends in a state satisfying $P_{d(m)}$, for some edge m originating at node n.

IV. Illustrating Proofs

In TLA, there is no distinction between a specification and a property; they are both formulas. Verification means proving that one formula implies another. A practical, relatively complete set of rules for proving such implications is described in [3]. We show here how predicate-action diagrams can be used to illustrate these proofs. We take as our example the same one treated in [3], that the specification Ψ defined in Section IV-A below implies the specification Φ defined in Section II above.

A. Another Specification

We define a TLA formula Ψ describing a program with two processes, each of which repeatedly loops through the sequence of operations $P(\mathit{sem})$; *increment*; $V(\mathit{sem})$, where one process increments x by one and the other increments y by one. Here, $P(\mathit{sem})$ and $V(\mathit{sem})$ denote the usual operations on a semaphore *sem*. To describe this program formally, we introduce a variable *pc* that indicates the control state. Each process has three control points, which we call "a", "b", and "g". (Quotes indicate string values.)

We motivate the definition of Ψ with the three predicate-action diagrams for Ψ in Figure 10. In these diagrams, the predicate $PC(p, q)$ asserts that control is at p in process 1 and at q in process 2. Figure 10(a) shows how the control state changes when the $P(\mathit{sem})$, $V(\mathit{sem})$, and *increment* actions are performed. Variables other than *pc* not mentioned in an edge label are left unchanged by the indicated steps—for example, steps described by the edge labeled $x' = x + 1$ leave y and *sem* unchanged—but this is not asserted by the diagram. The next-state action $\mathcal{N}$ is written as the disjunction $\mathcal{N}_1 \vee \mathcal{N}_2$ of the next-state actions of each process; and each $\mathcal{N}_i$ is written as the disjunction $\alpha_i \vee \beta_i \vee \gamma_i$. Figure 10(b) illustrates this decomposition. Finally, the predicate-action diagram of Figure 10(c) describes how the semaphore variable *sem* changes.

To write the specification Ψ, we let *pc* be a function with domain $\{1, 2\}$, with $pc[i]$ indicating where control resides in process i. The formula $PC(p, q)$ can then be defined by

$$PC(p, q) \ \triangleq\ (pc[1] = p) \wedge (pc[2] = q)$$

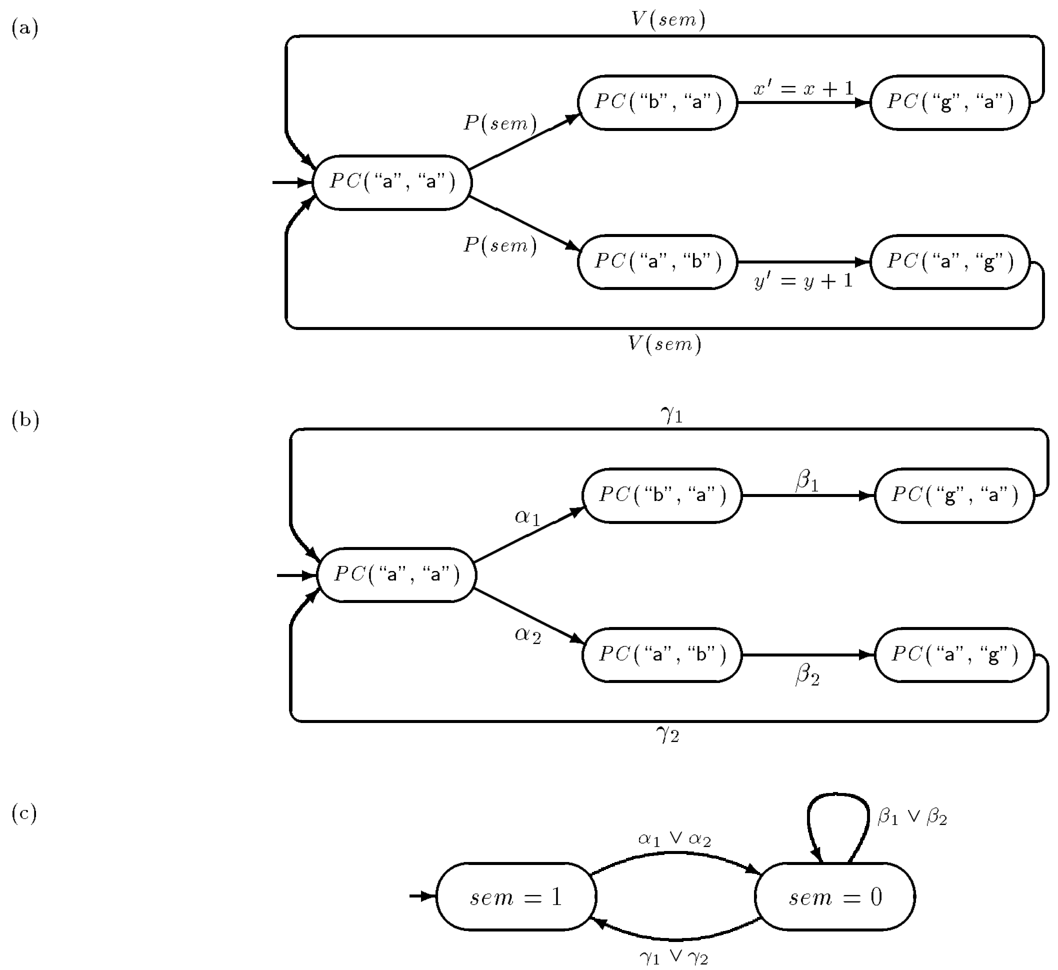


Fig. 10. Three predicate-action diagrams of $\langle x, y, pc, sem\rangle$ for Ψ.

The semaphore actions P and V are defined by

$$\begin{array}{rcl} P(sem) & \triangleq & \wedge\ 0 < sem \\ & & \wedge\ sem' = sem - 1 \\ V(sem) & \triangleq & sem' = sem + 1 \end{array}$$

Missing from Figure 10 are a specification of the initial values of x and y, which we take to be zero, and a fairness condition. One could augment predicate-action diagrams with some notation for indicating fairness conditions. However, the conditions that are easy to represent with a diagram are not expressive enough to describe the variety of fairness requirements that arise in practice. The WF and SF formulas, which are expressive enough, are not easy to represent graphically. So, we have not attempted to represent fairness in our diagrams. We take as the fairness condition for our specification Ψ strong fairness on the next-state action $\mathcal{N}_i$ of each process. The complete definition of Ψ appears in Figure 11.

B. An Illustrated Proof

The proof of $\Psi \Rightarrow \Phi$ is broken into the proof of three conditions:

1. $Init_\Psi \Rightarrow Init_\Phi$
2. $Init_\Psi \wedge \Box[\mathcal{N}]_w \Rightarrow \Box[\mathcal{M}]_{\langle x,y\rangle}$
3. $\Psi \Rightarrow \mathrm{WF}_{\langle x,y\rangle}(\mathcal{M}_i)$, for $i = 1, 2$

We illustrate the proofs of conditions 2 and 3 with the predicate-action diagram of $\langle x, y, sem, pc\rangle$ for Ψ in Figure 12, where Q is defined by

$$\begin{array}{rcl} Q_i(p,q) & \triangleq & \wedge\ PC(p,q) \\ & & \wedge\ sem\ =\ i \\ & & \wedge\ (x \in Nat) \wedge (y \in Nat) \end{array}$$

and Nat is the set of natural numbers.

First, we must show that the diagram in Figure 12 is a predicate-action diagram for Ψ. This is easy using the definition in Section III-B.1; no invariant is needed. For example, the condition to be proved for the node labeled $Q_0($"b", "a"$)$ is that an $\mathcal{N}$ step that starts with $Q_0($"b", "a"$)$ true is an $\mathcal{M}_1$ step (one that increments x and leaves y unchanged) that makes $Q_0($"g", "a"$)$ true. This follows easily from the definitions of Q and $\mathcal{N}$, since an $\mathcal{N}$ step starting with $PC($"b", "a"$)$ true must be a β_1 step.

To prove condition 2, it suffices to prove that every step allowed by the diagram of Figure 12 is a $[\mathcal{M}]_{\langle x,y\rangle}$ step. The steps not shown explicitly by the diagram are ones that leave w unchanged. Such steps leave $\langle x, y\rangle$ unchanged, so they are $[\mathcal{M}]_{\langle x,y\rangle}$ steps. The actions labeling all the edges of the diagram imply $[\mathcal{M}]_{\langle x,y\rangle}$, so all the steps shown ex-

$$
\begin{aligned}
Init_{\Psi} &\triangleq \land\ pc = [i \in \{1,2\} \mapsto \text{"a"}] \\
&\quad\ \land\ (x = 0) \land (y = 0) \\
&\quad\ \land\ sem = 1 \\
\alpha_i &\triangleq \land\ (pc[i] = \text{"a"}) \land (0 < sem) \\
&\quad\ \land\ pc' = [pc\ \mathbf{except}\ ![i] = \text{"b"}] \\
&\quad\ \land\ sem' = sem - 1 \\
&\quad\ \land\ \langle x, y\rangle' = \langle x, y\rangle \\
\gamma_i &\triangleq \land\ pc[i] = \text{"g"} \\
&\quad\ \land\ pc' = [pc\ \mathbf{except}\ ![i] = \text{"a"}] \\
&\quad\ \land\ sem' = sem + 1 \\
&\quad\ \land\ \langle x, y\rangle' = \langle x, y\rangle
\end{aligned}
\qquad
\begin{aligned}
\beta_1 &\triangleq \land\ pc[1] = \text{"b"} \\
&\quad\ \land\ pc' = [pc\ \mathbf{except}\ ![1] = \text{"g"}] \\
&\quad\ \land\ x' = x + 1 \\
&\quad\ \land\ \langle y, sem\rangle' = \langle y, sem\rangle \\
\beta_2 &\triangleq \land\ pc[2] = \text{"b"} \\
&\quad\ \land\ pc' = [pc\ \mathbf{except}\ ![2] = \text{"g"}] \\
&\quad\ \land\ y' = y + 1 \\
&\quad\ \land\ \langle x, sem\rangle' = \langle x, sem\rangle \\
\mathcal{N}_i &\triangleq \alpha_i \lor \beta_i \lor \gamma_i \\
\mathcal{N} &\triangleq \mathcal{N}_1 \lor \mathcal{N}_2 \\
w &\triangleq \langle x, y, sem, pc\rangle
\end{aligned}
$$

$$\Psi \triangleq Init_{\Psi} \land \Box[\mathcal{N}]_w \land \mathrm{SF}_w(\mathcal{N}_1) \land \mathrm{SF}_w(\mathcal{N}_2)$$

Fig. 11. The specification Ψ.

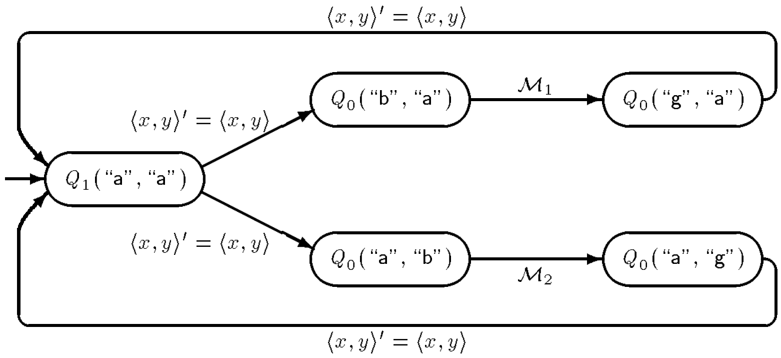


Fig. 12. Another predicate-action diagram of $\langle x, y, sem, pc\rangle$ for Ψ.

plicitly by the diagram are also $[\mathcal{M}]_{\langle x,y\rangle}$ steps. This proves condition 2.

We now sketch the proof of condition 3. To prove $\mathrm{WF}_{\langle x,y\rangle}(\mathcal{M}_i)$, it suffices to show that infinitely many $\langle \mathcal{M}_i\rangle_{\langle x,y\rangle}$ steps occur. We first observe that each of the predicates labeling a node in the diagram implies that either $\langle \mathcal{N}_1\rangle_w$ or $\langle \mathcal{N}_2\rangle_w$ is enabled. The fairness condition of Ψ then implies that a behavior cannot remain forever at any node, but must keep moving through the diagram. Hence, the behavior must infinitely often pass through the $Q_1(\text{"a"}, \text{"a"})$ node. The predicate $Q_1(\text{"a"}, \text{"a"})$ implies that both $\langle \mathcal{N}_1\rangle_w$ and $\langle \mathcal{N}_2\rangle_w$ are enabled. Hence, the fairness condition $\mathrm{SF}_w(\mathcal{N}_1) \land \mathrm{SF}_w(\mathcal{N}_2)$ implies that infinitely many $\langle \mathcal{N}_1\rangle_w$ steps and infinitely many $\langle \mathcal{N}_2\rangle_w$ steps must occur. Action $\langle \mathcal{N}_1\rangle_w$ is enabled only in the three nodes of the top loop. Taking infinitely many $\langle \mathcal{N}_1\rangle_w$ steps is therefore possible only by going around the top loop infinitely many times, which implies that infinitely many $\mathcal{M}_1$ steps occur, each starting in a state with $Q_0(\text{"b"}, \text{"a"})$ true. Since $Q_0(\text{"b"}, \text{"a"})$ implies $x \in Nat$, an $\mathcal{M}_1$ step starting with $Q_0(\text{"b"}, \text{"a"})$ true changes x, so it is an $\langle \mathcal{M}_1\rangle_{\langle x,y\rangle}$ step. Hence, infinitely many $\langle \mathcal{M}_1\rangle_{\langle x,y\rangle}$ steps occur. Similarly, taking infinitely many $\langle \mathcal{N}_2\rangle_w$ steps implies that infinitely many $\langle \mathcal{M}_2\rangle_{\langle x,y\rangle}$ steps occur. This completes the proof of condition 3.

Using the predicate-action diagram does not simplify the proof. If we were to make the argument given above rigorous, we would go through precisely the same steps as in the proof described in [3]. However, the diagram does allow us to visualize the proof, which can help us to understand it.

V. Conclusion

We have described three uses of diagrams that we believe are new for diagrams with a precise formal semantics:

- To describe particular aspects of a complex specification with a simple diagram. An n-input C-element cannot be specified with a simple picture. However, we explained the specification with diagrams describing the synchronization between the output and each individual input.
- To provide complementary views of the same system. Diagrams (b) and (c) of Figure 10 look quite different, but they are diagrams for the same specification.
- To illustrate proofs. The disjunction of the predicates labeling the nodes in Figure 12 equals the invariant I of the proof in Section 7.2 of [3]. The diagram provides a graphical representation of the invariance proof.

TLA differs from traditional specification methods in two important ways. First, all TLA specifications are interpreted over the same set of states. Instead of assigning values just to the variables that appear in the specification, a state assigns values to all of the infinite number of variables that can appear in any specification. Second, TLA specifications are invariant under stuttering. A formula can neither require nor rule out finite sequences of steps that do not change any variables mentioned in the formula. (The state-function subscripts in TLA formulas are there to guarantee invariance under stuttering.)

These two differences lead to two major differences between traditional state-transition diagrams and predicate-action diagrams. In traditional diagrams, each node represents a single state. Because states in TLA assign values to an infinite number of variables, it is impossible to describe a single state with a formula. Any formula can specify the values of only a finite number of variables. To draw diagrams of TLA formulas, we let each node represent a predicate, which describes a set of states. In traditional diagrams, every possible state change is indicated by an edge. Because TLA formulas are invariant under stuttering, we draw diagrams of particular state functions—usually tuples of variables.

TLA differs from most specification methods because it is a logic. It uses simple logical operations like implication and conjunction instead of more complicated automata-based notions of simulation and composition [6]. Everything we have done with predicate-action diagrams can be done with state-transition diagrams in any purely state-based formalism. However, conventional formalisms must use some notion of homomorphism between diagrams to describe what is expressed in TLA as logical implication.

Most formalisms employing state-transition diagrams are not purely state-based, but use both states and events. Nodes represent states, and edges describe input and output events. The meaning of a diagram is the sequence of events it allows; the states are effectively hidden. In TLA, there are only states, not events. Systems are described in terms of changes to interface variables rather than in terms of interface events. Variables describing the internal state are hidden with the existential quantifier $\boldsymbol{\exists}$ described in [3]. Changes to any variable, whether internal or interface, can be indicated by node labels or edge labels. Hence, a purely state-based approach like TLA allows more flexibility in how diagrams are drawn than a method based on states and events.

References

[1] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 34, pp. 1045–1079, Sept. 1955.

[2] E. F. Moore, "Gedanken-experiments on sequential machines," in *Automata Studies* (C. E. Shannon and J. McCarthy, eds.), pp. 129–153, Princeton, New Jersey: Princeton University Press, 1956.

[3] L. Lamport, "The temporal logic of actions," *ACM Trans. Programming Languages and Systems*, vol. 16, pp. 872–923, May 1994.

[4] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, pp. 181–185, Oct. 1985.

[5] C. Mead and L. Conway, *Introduction to VLSI Systems*, ch. 7. Reading, Massachusetts: Addison-Wesley, 1980.

[6] M. Abadi and L. Lamport, "Conjoining specifications," Research Report 118, Digital Equipment Corporation, Systems Research Center, 1993. To appear in *ACM Transactions on Programming Languages and Systems*.

Leslie Lamport Leslie Lamport attended the Bronx High School of Science, where he took a course in mechanical drawing. He later received a Ph.D. in mathematics from Brandeis University, where he studied the propagation of singularities in the Cauchy problem for analytic partial differential equations. Since 1985, he has been a member of Digital Equipment Corporation's Systems Research Laboratory, where he has written several biographical sketches.