Open Systems in TLA

Martín Abadi and Leslie Lamport
Digital Equipment Corporation
Systems Research Center

Abstract

We describe a method for writing assumption/guarantee specifications of concurrent systems. We also provide a proof rule for reasoning about the composition of these systems. Specifications are written in TLA (the Temporal Logic of Actions), and all reasoning is performed within the logic. Our proof rule handles internal variables and both safety and liveness properties.

1 Introduction

An open system is one that interacts with an environment that neither it nor its implementor controls. To deduce useful properties of a system, we must specify its environment. No system will exhibit its intended behavior in the presence of a sufficiently hostile environment. For example, a combinational circuit will not produce an output in the intended range if some input line, instead of having a 0 or a 1, has an improper voltage level of 1/2. The specification of the circuit's environment must rule out such improper inputs. An open system calls for an assumption/guarantee specification, asserting that the system satisfies a guarantee only as long as its environment satisfies an assumption [9].

In this paper, we study how to specify open systems and how to reason about their composition. A companion paper considers the different problems that arise when a given system is decomposed into components [1]. Decomposition and composition are both discussed in more detail in [4].

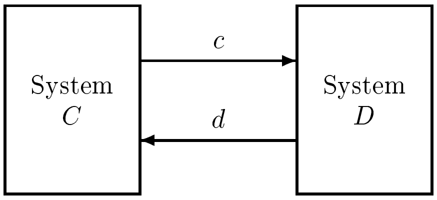

Figure 1: A simple example.

The setting for our work is temporal logic, where specifications are formulas and programs are viewed as lower-level specifications. A specification S^l implements another specification S iff $S^l \Rightarrow S$ is valid. We write specifications in such a way that the parallel composition of specifications S_1 and S_2 is $S_1 \wedge S_2$.

In our approach, a system guarantee M and an environment assumption E are temporal-logic formulas. The corresponding assumption/guarantee specification is also a formula that we write $E \mathrel{\stackrel{+}{-\!\!\triangleright}} M$. Section 3 defines $\mathrel{\stackrel{+}{-\!\!\triangleright}}$ and discusses this form of specification.

To show that a composition of open systems satisfies a specification S, we must prove a formula of the form $(E_1 \mathrel{\stackrel{+}{-\!\!\triangleright}} M_1) \wedge \ldots \wedge (E_n \mathrel{\stackrel{+}{-\!\!\triangleright}} M_n) \Rightarrow S$, where S may again be an assumption/guarantee property. Unfortunately, it is not obvious how to reason about the composition of systems described by assumption/guarantee specifications. The basic problem is illustrated by the simple case of two systems, one guaranteeing M_c assuming M_d, and the other guaranteeing M_d assuming M_c. Since each system guarantees to satisfy the other's environment assumption, we would like to conclude that their composition implements the specification $M_c \wedge M_d$ unconditionally, with no environment assumption. Can we? We attempt to answer this question by considering two simple examples, based on Figure 1.

In the first example, M_c^0 asserts that c always equals 0, and M_d^0 asserts that d always equals 0. We can implement the corresponding assump-

---

0

tion/guarantee specifications with two processes Π_c and Π_d. Process Π_c starts with c equal to 0 and repeatedly sets c to the current value of d. It obviously guarantees M_c^0 assuming M_d^0. Process Π_d is analogous; it guarantees M_d^0 assuming M_c^0. Clearly, the composition of Π_c and Π_d keeps c and d both equal to 0, implementing $M_c^0 \wedge M_d^0$.

In the second example, M_c^1 asserts that c eventually equals 1, and M_d^1 asserts that d eventually equals 1. Process Π_c also guarantees M_c^1 assuming M_d^1, and process Π_d also guarantees M_d^1 assuming M_c^1. However, since the composition of Π_c and Π_d leaves c and d unchanged, it does not implement $M_c^1 \wedge M_d^1$.

Our conclusion in the first example does not depend on the particular choice of processes Π_c and Π_d. We can deduce directly from the assumption/guarantee specifications that the composition must implement $M_c^0 \wedge M_d^0$, because the first process to change its output variable would violate its guarantee before its assumption had been violated. This argument does not apply to the second example, because violating M_c^1 and M_d^1 are sins of omission that do not occur at any particular instant. A property that can be made false only by being violated at some instant is called a safety property [6]. As the examples suggest, reasoning about the composition of assumption/guarantee specifications is easiest when assumptions are safety properties.

Our rules for reasoning about the composition of assumption/guarantee specifications are embodied in the Composition Theorem of Section 5. For such a theorem to be of any value, it must be accompanied by a precise logic for writing specifications and a method for verifying its hypotheses. The logic we use is TLA, the Temporal Logic of Actions [10]. We prove propositions helpful in verifying the hypotheses of the theorem for TLA specifications. Together with these propositions, the Composition Theorem allows us to reason about open systems using well-established, effective methods for reasoning about complete systems.

With the Composition Theorem, it is trivial to prove that the conjunction of the assumption/guarantee specifications $M_c^0 \stackrel{+}{\rhd} M_d^0$ and $M_d^0 \stackrel{+}{\rhd} M_c^0$ implies $M_c^0 \wedge M_d^0$. As a less trivial example, in Section 6 we sketch the proof that the composition of two queues implements a larger queue.

In Section 2, we review TLA and our method of specifying components. Much of this section also appears in [1]. Section 3 defines the operator $\stackrel{+}{\rhd}$, and Section 4 defines some additional operators. The Composition Theorem is given in Section 5. Section 6 illustrates how the concepts and results fit together. We discuss related work and conclude in Section 7.

2 Preliminaries

2.1 Review of the Syntax and Semantics of TLA

A state is an assignment of values to variables. (Technically, our variables are the "flexible" variables of temporal logic that correspond to the variables of programming languages; they are distinct from the variables of first-order logic.) A behavior is an infinite sequence of states. Semantically, a TLA formula F is true or false of a behavior; we say that F is *valid*, and write $\models F$, iff it is true of every behavior. Syntactically, TLA formulas are built up from state functions using Boolean operators ($\neg$, $\wedge$, $\vee$, $\Rightarrow$ [implication], and $=$ [equivalence]) and the operators $'$, $\Box$, and $\exists\!\!\!\exists$, as described below.

A *state function* is like an expression in a programming language. Semantically, it assigns a value to each state—for example $3 + x$ assigns to state s three plus the value of the variable x in s. A *state predicate* is a Boolean-valued state function. An *action* is a Boolean-valued 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, $x + 1 > y'$ is true for $\langle s, t\rangle$ iff the value of $x+1$ in s is greater than the value of y in t. A pair of states satisfying action $\mathcal{A}$ is called an $\mathcal{A}$ *step*. We say that $\mathcal{A}$ is *enabled* in state s iff there exists a state t such that $\langle s, t\rangle$ is an $\mathcal{A}$ step. We write f' for the expression obtained by priming all the variables of the state function f, and $[\mathcal{A}]_f$ for $\mathcal{A} \vee (f' = f)$, so an $[\mathcal{A}]_f$ step is either an $\mathcal{A}$ step or a step that leaves f unchanged.

As usual in temporal logic, if F is a formula then $\Box F$ is a formula that means that F is always true. 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 either there are infinitely many $\mathcal{A}$ steps that change v, or there are infinitely many states in which such steps are not enabled. The *strong fairness* formula $\mathrm{SF}_v(\mathcal{A})$ asserts that either there are infinitely many $\mathcal{A}$ steps that change v, or there are only finitely many states in which such steps are enabled.

The formula $\exists\!\!\!\exists x : F$ essentially means that there is some way of choosing a sequence of values for x such that the temporal formula F holds. We think of $\exists\!\!\!\exists x : F$ as "F with x hidden" and call x an internal variable of $\exists\!\!\!\exists x : F$. If x is a tuple of variables $\langle x_1, \ldots, x_k\rangle$, we write $\exists\!\!\!\exists x : F$ for $\exists\!\!\!\exists x_1 : \ldots \exists\!\!\!\exists x_k : F$.

Intuitively, a variable represents some part of the universe and a behavior represents a possible complete history of the universe. A system Π is represented by a TLA formula M that is true for precisely those behaviors that represent histories in which Π is running. We make no formal distinction between systems, specifications, and properties; they are all represented by TLA formulas, which we usually call specifications.

2.2 Specifying Components and Complete Systems

A system guarantee should describe what we want the system to do, without saying anything about what the environment does. Similarly, an environment assumption should describe only the environment's behavior, not the system's. We can consider an open system and its environment to be separate components that together form a complete system. We now explain how to specify such a component in TLA.

For simplicity, in this paper we describe only the special case where the visible variables of the component's specification can be partitioned into a tuple m of output variables and a tuple e of input variables. We also consider only interleaving specifications, which assert that inputs and outputs do not change simultaneously. We discuss other specification styles in [4].

The specification M of a component has the "canonical form" $\exists\!\!\exists x : \mathit{Init} \wedge \Box[\mathcal{N}]_v \wedge L$, where:

v is the tuple $\langle m, x\rangle$. Thus, the specification allows any step that does not change the component's output variables m or internal variables x; such a step represents an act of the component's environment.

Init is a predicate that describes the initial values of the component's output variables m and internal variables x.

$\mathcal{N}$ is the "next-state" action that describes the steps performed by the component. In an interleaving representation, the component's inputs and outputs cannot change simultaneously, so $\mathcal{N}$ implies $e' = e$.

L is the conjunction of fairness conditions of the form $\mathrm{WF}_{\langle m, x\rangle}(\mathcal{A})$ and $\mathrm{SF}_{\langle m, x\rangle}(\mathcal{A})$.

This formula asserts that there exists a sequence of values for x such that *Init* is true for the initial state, every step of the behavior is an $\mathcal{N}$ step or leaves v unchanged, and L holds.

A complete system is one with a single component and no input variables, so its specification has this same form with v the tuple of all relevant variables.

2.3 Conditional Implementation

A specification M^l implements a specification M iff every behavior that satisfies M^l also satisfies M, that is, iff $M^l \Rightarrow M$ is valid [10]. Instead of proving that a specification M^l implements a specification M, we sometimes want to prove the weaker condition that M^l implements M assuming a formula G. In other words, we want to prove $G \Rightarrow (M^l \Rightarrow M)$, which is equivalent to $G \wedge M^l \Rightarrow M$. The formula G may express one or more of the following:

- A law of nature. For example, in a real-time specification, G might assert that time increases monotonically.
- An interface refinement, where G expresses the relation between a low-level tuple l of variables and its high-level representation as a tuple h of variables.
- An assumption about how reality is translated into the formalism of behaviors. In particular, G may assert the interleaving assumption $\mathit{Disjoint}(v_1, \ldots, v_n)$, which means that no two of the tuples of variables v_i change simultaneously, and is defined as:

$$\bigwedge_{i \neq j} \Box[(v'_i = v_i) \vee (v'_j = v_j)]_{\langle v_i,\, v_j\rangle}$$

Conditional implementation, with an explicit formula G, is needed only for open systems. For a complete system, the properties expressed by G can easily be made part of the system's specification. For example, the system can include a component that advances time. In contrast, it can be difficult to include G in the specification of an open system.

2.4 Safety and Closure

A finite sequence of states is called a finite behavior. For any formula F and finite behavior ρ, we say that ρ *satisfies* F iff ρ can be extended to an infinite behavior that satisfies F. A *safety property* is a formula that is satisfied by an infinite behavior σ iff it is satisfied by every prefix of σ [6]. It can be shown that, for any TLA formula F, there is a TLA formula $\mathcal{C}(F)$, called the *closure of* F, such that a behavior σ satisfies $\mathcal{C}(F)$ iff every prefix of σ satisfies F. Formula $\mathcal{C}(F)$ is the strongest safety property such that $\models F \Rightarrow \mathcal{C}(F)$.

When writing a specification in the form $\mathit{Init} \wedge \Box[\mathcal{N}]_v \wedge L$, we expect L to constrain infinite behaviors, not finite ones. Formally, this means that the closure of $\mathit{Init} \wedge \Box[\mathcal{N}]_v \wedge L$ should be $\mathit{Init} \wedge \Box[\mathcal{N}]_v$.

Proposition 1 shows that this is the case when L is the conjunction of fairness properties (under reasonable assumptions). It is an immediate consequence of a result proved in [2].

Proposition 1 *If L is the conjunction of countably many formulas of the form $\mathrm{WF}_w(\mathcal{A})$ and/or $\mathrm{SF}_w(\mathcal{A})$ such that $\mathcal{A}$ implies $\mathcal{N}$, then*

$$\models \mathcal{C}(\mathit{Init} \wedge \Box[\mathcal{N}]_v \wedge L) \;=\; \mathit{Init} \wedge \Box[\mathcal{N}]_v$$

Some of our results have hypotheses of the form $\models \mathcal{C}(M_1) \wedge \ldots \wedge \mathcal{C}(M_n) \Rightarrow \mathcal{C}(M)$. The obvious first step in proving such a formula is to compute the closures $\mathcal{C}(M_1)$, $\ldots$, $\mathcal{C}(M_n)$, and $\mathcal{C}(M)$. We can use Proposition 1 to compute the closure of a formula with no internal variables. When there are internal variables, the following proposition allows us to reduce the proof of $\mathcal{C}(M_1) \wedge \ldots \wedge \mathcal{C}(M_n) \Rightarrow \mathcal{C}(M)$ to the proof of a formula in which the closures can be computed with Proposition 1.

Proposition 2 *Let x, x_1, $\ldots$, x_n be tuples of variables such that for each i, no variable in x_i occurs in M or in any M_j with $i \neq j$.*

$$\begin{array}{ll} \textit{If} & \models \displaystyle\bigwedge_{i=1}^{n} \mathcal{C}(M_i) \;\Rightarrow\; \exists\, x : \mathcal{C}(M) \\ \textit{then} & \models \displaystyle\bigwedge_{i=1}^{n} \mathcal{C}(\exists\, x_i : M_i) \;\Rightarrow\; \mathcal{C}(\exists x : M). \end{array}$$

3 Assumption/Guarantee Specifications

An assumption/guarantee specification asserts that a system guarantees M under the assumption that its environment satisfies E, where M and E are component specifications of the type described in Section 2.2. Perhaps the most obvious form for this assumption/guarantee specification is $E \Rightarrow M$. Another appealing form is $E \mathrel{-\!\!\triangleright} M$, which asserts that M holds at least as long as E does [5]. Instead, we take as the specification the formula $E \overset{+}{-\!\!\triangleright} M$, which we define to mean that, for any n, if the environment satisfies E through "time" n, then the system must satisfy M through "time" $n+1$. More precisely, $E \overset{+}{-\!\!\triangleright} M$ is true of a behavior σ iff $E \Rightarrow M$ is true of σ and, for every $n \geq 0$, if E holds for the first n states of σ, then M holds for the first $n+1$ states of σ. (The formula $E \overset{+}{-\!\!\triangleright} M$ can be expressed in terms of the primitives $'$, $\Box$, and $\exists$.)

The formulas $E \Rightarrow M$ and $E \mathrel{-\!\!\triangleright} M$ are both weaker than $E \overset{+}{-\!\!\triangleright} M$, since they allow behaviors in which M is violated before E or at the same time as E, respectively. An implementation could exploit the extra freedom of $E \Rightarrow M$ only by predicting in advance that the environment will violate E. Similarly, the extra freedom of $E \mathrel{-\!\!\triangleright} M$ could only be useful if the system could react instantaneously to its environment. Therefore, the specifications $E \Rightarrow M$, $E \mathrel{-\!\!\triangleright} M$, and $E \overset{+}{-\!\!\triangleright} M$ all allow the same implementations. We take $E \overset{+}{-\!\!\triangleright} M$ to be the form of assumption/guarantee specifications because it leads to the simpler rules for composition.

As suggested by the discussion in Section 1, composition works well only when environment assumptions are safety properties. Because $E \overset{+}{-\!\!\triangleright} M$ is equivalent to $\mathcal{C}(E) \overset{+}{-\!\!\triangleright} (\mathcal{C}(M) \wedge (E \Rightarrow M))$, we can in principle convert any assumption/guarantee specification to one whose assumption is a safety property. (A similar observation appears as Theorem 1 of [3].) However, this equivalence is of intellectual interest only. In practice, we write the environment assumption as a safety property and the system's fairness guarantee as the conjunction of properties $E_L \Rightarrow \mathrm{WF}_v(\mathcal{A})$ and $E_L \Rightarrow \mathrm{SF}_v(\mathcal{A})$, where E_L is an environment fairness assumption. We can still apply Proposition 1 because, if $\mathcal{C}(P \wedge L) = P$ and L implies R, then $\mathcal{C}(P \wedge R) = P$ [2, Proposition 3].

4 Additional Temporal Operators

We now define two additional temporal operators. The first is useful in stating the Composition Principle; the second is an auxiliary operator important in verifying the hypotheses of the Composition Principle. Both can be expressed in terms of the primitives $'$, $\Box$, and $\exists$, but we define them semantically.

4.1 +

The formula E_{+v} asserts that, if the temporal formula E ever becomes false, then the state function v stops changing. More precisely, a behavior σ satisfies E_{+v} iff either σ satisfies E, or there is some n such that E holds for the first n states of σ, and v never changes from the $(n+1)$st state on. When E is a safety property in canonical form, it is easy to write E_{+v} explicitly.

We need to reason about the $+$ operator only to check hypotheses of the form $\models \mathcal{C}(E)_{+v} \wedge \mathcal{C}(M^l) \Rightarrow \mathcal{C}(M)$ in the Composition Theorem. We can check such a hypothesis by calculating $\mathcal{C}(E)_{+v}$ explicitly. While this approach is viable, it is required only for noninterleaving specifications. Proposition 3 below provides a better way of proving these hypotheses for interleaving specifications.

4.2 $\perp$

The specification M of a component can be made false only by a step that changes the component's output variables. In an interleaving representation, we do not allow a single step to change output variables of two different components. Hence, if E and M are specifications of separate components, we expect that no step will make both E and M false. More precisely, we expect E and M to be orthogonal ($\perp$), where $E \perp M$ is true of a behavior σ iff there is no $n \geq 0$ such that E and M are both true for the first n states of σ and both false for the first $n+1$ states of σ.

We can use orthogonality to remove $+$ from proof obligations:

Proposition 3 *If E, M, and R are safety properties, and v is a tuple of variables containing all variables that occur free in M, then $\models E \wedge R \Rightarrow M$ and $\models R \Rightarrow E \perp M$ imply $\models E_{+v} \wedge R \Rightarrow M$.*

To apply this proposition, we must prove the orthogonality of component specifications. We do this for interleaving specifications with the following result.

Proposition 4

$$
\begin{array}{ll}
\textit{If} & \models \mathcal{C}(E) = \textit{Init}_E \wedge \Box[\mathcal{N}_E]_{\langle x,\, e\rangle} \ \textit{and} \\
& \models \mathcal{C}(M) = \textit{Init}_M \wedge \Box[\mathcal{N}_M]_{\langle y,\, m\rangle} \\
\textit{then} & \models (\exists x : \textit{Init}_E \vee \exists y : \textit{Init}_M) \wedge \textit{Disjoint}(e, m) \\
& \quad \Rightarrow\ \mathcal{C}(\boldsymbol{\exists} x : E) \perp \mathcal{C}(\boldsymbol{\exists} y : M).
\end{array}
$$

If no step falsifies both E and M, and M remains true as long as E does, then M must remain true at least one step longer than E does. Hence, $E \perp M$ implies the equivalence of $E \mathrel{-\!\!\triangleright} M$ and $E \stackrel{+}{-\!\!\triangleright} M$. In fact, $(E \stackrel{+}{-\!\!\triangleright} M) = (E \mathrel{-\!\!\triangleright} M) \wedge (E \perp M)$ is valid.

5 The Composition Theorem

Suppose we are given n devices, each with an assumption/guarantee specification $E_j \stackrel{+}{-\!\!\triangleright} M_j$. To verify that the composition of these devices implements a higher-level assumption/guarantee specification $E \stackrel{+}{-\!\!\triangleright} M$, we must prove $\bigwedge_{j=1}^{n}(E_j \stackrel{+}{-\!\!\triangleright} M_j) \Rightarrow (E \stackrel{+}{-\!\!\triangleright} M)$. We use the following theorem:

Composition Theorem *If, for $i = 1, \ldots, n$,*

1. $\models \mathcal{C}(E) \wedge \bigwedge_{j=1}^{n} \mathcal{C}(M_j) \ \Rightarrow\ E_i$

2. *(a)* $\models \mathcal{C}(E)_{+v} \wedge \bigwedge_{j=1}^{n} \mathcal{C}(M_j) \ \Rightarrow\ \mathcal{C}(M)$

   (b) $\models E \wedge \bigwedge_{j=1}^{n} M_j \ \Rightarrow\ M$

then $\models \bigwedge_{j=1}^{n} (E_j \stackrel{+}{-\!\!\triangleright} M_j) \ \Rightarrow\ (E \stackrel{+}{-\!\!\triangleright} M)$.

To discharge the hypotheses, we use Propositions 1 and 2 to eliminate $\mathcal{C}$'s, and Propositions 3 and 4 to eliminate the $+v$. The example in Section 6 illustrates how the propositions are applied.

Observe that the hypotheses all have the form $\models P \wedge \bigwedge_{j=1}^{n} Q_j \Rightarrow R$. Each formula $P \wedge \bigwedge_{j=1}^{n} Q_j$ is the conjunction of the specifications of components that together form a complete system. In fact, simple logical manipulation shows that this formula is equivalent to a canonical-form specification of that complete system [1]. Thus, each hypothesis asserts that a complete system satisfies a property R. In other words, the theorem reduces reasoning about assumption/guarantee specifications to the kind of reasoning used for complete-system specifications.

This theorem also allows us to prove conditional implementation results of the form

$$\models G \wedge \bigwedge_{j=1}^{n} (E_j \stackrel{+}{-\!\!\triangleright} M_j) \Rightarrow (E \stackrel{+}{-\!\!\triangleright} M)$$

We just let M_1 equal G and E_1 equal true, since $\mathsf{true} \stackrel{+}{-\!\!\triangleright} G$ equals G. For interleaving specifications, we can in general prove only conditional implementation, where G includes disjointness conditions asserting that the outputs of different components do not change simultaneously.

Among the corollaries of the Composition Theorem are ones that allow us to prove that a lower-level specification implies a higher-level one. The simplest such result has $\models (E \stackrel{+}{-\!\!\triangleright} M^l) \Rightarrow (E \stackrel{+}{-\!\!\triangleright} M)$ as its conclusion. This condition expresses the correctness of the refinement of a system with a fixed environment assumption.

Corollary *If E is a safety property and*

(a) $\models E_{+v} \wedge \mathcal{C}(M^l) \Rightarrow \mathcal{C}(M)$

(b) $\models E \wedge M^l \Rightarrow M$

then $\models (E \stackrel{+}{-\!\!\triangleright} M^l) \Rightarrow (E \stackrel{+}{-\!\!\triangleright} M)$.

6 The Queue Example

6.1 Informal Description of the Queue

In our example, we consider systems that communicate by using a standard two-phase handshake protocol [11] to send values over channels. The state of a channel c is described by three components: the value $c.val$ that is being sent, and two bits $c.sig$ and $c.ack$ used for synchronization. We let $c.snd$ denote the pair $\langle c.sig, c.val \rangle$. We also write c for the triple $\langle c.sig, c.ack, c.val \rangle$.

Figure 2 shows the sequence of states assumed in sending the sequence of values 37, 4, ... The channel is ready to send when $c.sig = c.ack$. A value v is sent by setting $c.val$ to v and complementing $c.sig$. Receipt of the value is acknowledged by complementing $c.ack$.

	start state	*37 sent*	*37 acked*	*4 sent*	*4 acked*	
$c.ack$:	0	0	1	1	0	...
$c.sig$:	0	1	1	0	0	...
$c.val$:	–	37	37	4	4	...

Figure 2: The two-phase handshake protocol for a channel c.

We consider an N-element queue with input channel i and output channel o. It is depicted in Figure 3. To describe the queue, we introduce the following notation for finite sequences: $|\rho|$ denotes the length of sequence ρ, which equals 0 if ρ is empty; $Head(\rho)$ and $Tail(\rho)$ as usual denote the head (first element) and the tail of sequence ρ, if ρ is nonempty; and $\rho \circ \tau$ denotes the concatenation of sequences ρ and τ. Angle brackets are used to form sequences, so $\langle\, \rangle$ denotes the empty sequence and $\langle e \rangle$ denotes the sequence with e as its only element. With this notation, the queue can be written as in Figure 4, where large angle brackets enclose atomic operations.

We will define QM to be the TLA formula that represents this queue process. It will be the queue's guarantee—a component specification, of the sort described in Section 2.2. It is impossible to implement this guarantee if the environment does not obey the communication protocol. For example, in a lower-level implementation, reading the input $o.ack$ and setting the outputs $o.sig$ and $o.val$ would be separate actions. If the environment changed $o.ack$ between these actions, the implementation could violate the requirement that it change $o.val$ only when $o.ack = o.sig$. This problem is not an artifact of our particular representation of the queue; actual hardware implementations of a queue can enter metastable states, consequently producing bizarre, unpredictable behavior, if their inputs are changed when they are not supposed to be [11]. An implementable specification of the queue must include an assumption QE asserting that the environment obeys the communication protocol.

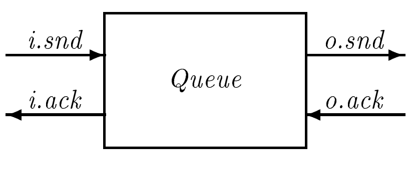

Figure 3: A queue.

Process *Queue*:
output var $i.ack$, $o.sig$ **initially** 0,
$o.val$;
internal var q **initially** $\langle\,\rangle$;
input var $i.sig$, $i.val$, $o.ack$;
cobegin
loop $\Big\langle$ **if** $(i.ack \neq i.sig) \wedge (|q| < N)$
then $q := q \circ \langle i.val \rangle$;
$i.ack := 1 - i.ack$ $\Big\rangle$ **endloop**
$\|$
loop $\Big\langle$ **if** $(o.ack = o.sig) \wedge (|q| > 0)$
then $o.val := head(q)$;
$q := tail(q)$;
$o.sig := 1 - o.sig$ $\Big\rangle$ **endloop**
coend

Figure 4: A queue process.

6.2 The Queue as a Complete System

Before defining QM and QE, we write a TLA specification of the complete system comprising the queue and its environment, shown in Figure 5.

A channel is initially ready for sending, so the initial condition on wire c is the predicate $CInit(c)$

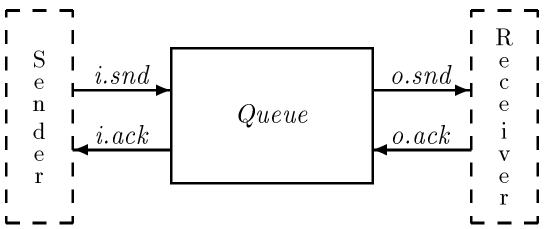

Figure 5: The complete system of queue plus environment.

defined by

$$CInit(c) \triangleq (c.sig = c.ack = 0)$$

The operations of sending a value v and acknowledging receipt of a value on channel c are represented by the following $Send(\mathsf{v}, c)$ and $Ack(c)$ actions.

$$\begin{array}{rcl} Send(\mathsf{v}, c) & \triangleq & \land\ c.sig = c.ack \\ & & \land\ c.snd' = \langle \mathsf{v},\ 1 - c.sig \rangle \\ & & \land\ c.ack' = c.ack \\ Ack(c) & \triangleq & \land\ c.sig \neq c.ack \\ & & \land\ c.ack' = 1 - c.ack \\ & & \land\ c.snd' = c.snd \end{array}$$

To represent the queue as a complete system, we add an environment that sends arbitrary natural numbers over channel i and acknowledges values on channel o.

The TLA formula CQ specifying the queue is defined in Figure 6. It has the canonical form $\exists\!\!\exists x : Init \land \Box[\mathcal{N}]_v \land L$, where:

x is the internal variable q, which represents the sequence of values received on the input channel i but not yet sent on the output channel o.

$Init$ is written as the conjunction $Init_E \land Init_M$ of initial predicates for the environment and the queue. (We arbitrarily consider the initial conditions on a channel to be part of the sender's initial predicate.)

$\mathcal{N}$ is the disjunction of two actions: $\mathcal{Q}_M$, describing the steps taken by the queue, and $\mathcal{Q}_E \land (q' = q)$, describing steps taken by the environment (which leave q unchanged). Action $\mathcal{Q}_M$ is the disjunction of actions Enq and Deq. An Enq step acknowledges receipt of a value on i and appends the value to q; it is enabled only when q has fewer than N elements. A Deq step removes the first element of q and sends it on o. Action $\mathcal{Q}_E$ is the disjunction of Put, which sends an arbitrary number on channel i, and Get, which acknowledges receipt of a number on channel o.

v is the tuple $\langle i, o, q \rangle$ of all relevant variables. (Informally, we write $\langle i, o, q \rangle$ for the concatenation of the tuples i, o, and $\langle q \rangle$.)

L is the weak-fairness formula $\mathrm{WF}_{\langle i, o, q\rangle}(\mathcal{Q}_M)$, which asserts that a queue step cannot remain forever possible without occurring. It can be shown that a logically equivalent specification is obtained if this condition is replaced with $\mathrm{WF}_{\langle i, o, q\rangle}(Enq) \land \mathrm{WF}_{\langle i, o, q\rangle}(Deq)$.

Environment actions

$$\begin{array}{lcl} Init_E & \triangleq & CInit(i) \\ Put & \triangleq & (\exists v \in \mathsf{Nat} : Send(v, i)) \ \land\ (o' = o) \\ Get & \triangleq & Ack(o) \ \land\ (i' = i) \\ \mathcal{Q}_E & \triangleq & Get \ \lor\ Put \end{array}$$

Queue actions

$$\begin{array}{lcl} Init_M & \triangleq & CInit(o) \ \land\ (q = \langle\,\rangle) \\ Enq & \triangleq & \land\ |q| < N \\ & & \land\ Ack(i) \ \land\ (q' = q \circ \langle i.val \rangle) \\ & & \land\ o' = o \\ Deq & \triangleq & \land\ |q| > 0 \\ & & \land\ Send(Head(q), o) \ \land\ (q' = Tail(q)) \\ & & \land\ i' = i \\ \mathcal{Q}_M & \triangleq & Enq \ \lor\ Deq \end{array}$$

Complete system specification

$$\begin{array}{lcl} ICL & \triangleq & \mathrm{WF}_{\langle i, o, q\rangle}(\mathcal{Q}_M) \\ ICQ & \triangleq & \land\ Init_E \ \land\ Init_M \\ & & \land\ \Box \begin{bmatrix} \lor\ \mathcal{Q}_E \ \land\ (q' = q) \\ \lor\ \mathcal{Q}_M \end{bmatrix}_{\langle i, o, q\rangle} \\ & & \land\ ICL \\ CQ & \triangleq & \exists\!\!\exists\, q : ICQ \end{array}$$

Figure 6: The specification CQ of the complete queue.

Formula CQ gives an interleaving representation of a queue; simultaneous steps by the queue and its environment are not allowed. Moreover, simultaneous changes to the two inputs $i.snd$ and $o.ack$ are disallowed, as are simultaneous changes to the two outputs $i.ack$ and $o.snd$.

6.3 The Component Specifications

We now describe the queue and its environment as separate components, specified in canonical form. For the queue component, the tuple m of output variables is $\langle i.ack,\ o.snd \rangle$, the tuple e of input variables is $\langle i.snd,\ o.ack \rangle$, and the specification is

$$\begin{array}{lcl} IQM & \triangleq & Init_M \ \land\ \Box[\mathcal{Q}_M]_{\langle i.ack,\, o.snd,\, q\rangle} \ \land\ ICL \\ QM & \triangleq & \exists\!\!\exists\, q : IQM \end{array} \qquad (1)$$

The specification of the environment as a separate component is

$$QE \triangleq Init_E \ \land\ \Box[\mathcal{Q}_E]_{\langle i.snd,\, o.ack\rangle} \qquad (2)$$

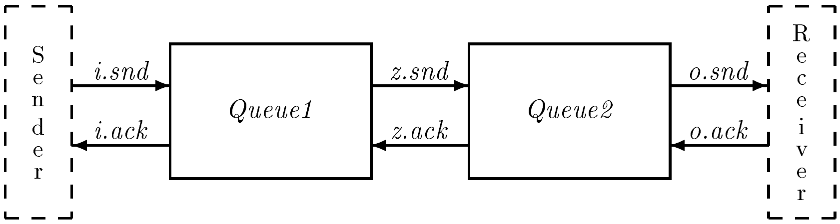

Figure 7: A complete system containing two queues in series.

$$
\begin{array}{l}
ICDQ \;\triangleq \\
\quad \land\ Init_E \ \land\ Init_M^{[1]} \ \land\ Init_M^{[2]} \\
\quad \land\ \Box \begin{bmatrix} \lor\ \mathcal{Q}_E \ \land \langle q_1, q_2, z\rangle' = \langle q_1, q_2, z\rangle \\ \lor\ \mathcal{Q}_M^{[1]} \land \langle q_2, o\rangle' = \langle q_2, o\rangle \\ \lor\ \mathcal{Q}_M^{[2]} \land \langle q_1, i\rangle' = \langle q_1, i\rangle \end{bmatrix}_{\langle i,\, o,\, z,\, q_1,\, q_2\rangle} \\
\quad \land\ ICL^{[1]} \ \land\ ICL^{[2]} \\
CDQ \;\triangleq\; \exists\, q_1, q_2 : ICDQ
\end{array}
$$

Figure 8: Specification of the complete double-queue system of Figure 7.

6.4 Implementing a Queue

The complete system composed of two queues in series and their environment, shown in Figure 7, implements a single larger queue and its environment.

We let $F[e_1/v_1, \ldots, e_n/v_n]$ denote the result of substituting each expression e_i for v_i in a formula F, and let

$$
\begin{array}{rcl}
F^{[1]} & \triangleq & F[z/o,\, q_1/q] \\
F^{[2]} & \triangleq & F[z/i,\, q_2/q] \\
F^{[\mathrm{dbl}]} & \triangleq & F[(2N+1)/N]
\end{array}
$$

Using this notation, the specification CDQ of the composite system is defined in Figure 8 in terms of the formulas from Figure 6.

The composite system implements a $(2N+1)$-element queue; formally, $CDQ \Rightarrow CQ^{[\mathrm{dbl}]}$ is valid. This result is proved by standard TLA reasoning using a simple refinement mapping [10].

6.5 Composing Open Queues

In Section 6.4, we specified the composition of two queues with their environment directly as a complete system, and stated that it implements a larger queue with its environment. We now consider open queues, described by assumption/guarantee specifications. Using the Composition Theorem, we show that the composition of two open queues implements a larger open queue.

The assumption/guarantee specification of the simple queue of Figure 3 is $QE \stackrel{+}{-\!\!\triangleright} QM$, where QM and QE are defined in (1) and (2) of Section 6.3. The assumption/guarantee specifications of the two queues in Figure 7 are obtained from $QE \stackrel{+}{-\!\!\triangleright} QM$ by substitution; they are $QE^{[1]} \stackrel{+}{-\!\!\triangleright} QM^{[1]}$ and $QE^{[2]} \stackrel{+}{-\!\!\triangleright} QM^{[2]}$. We want to show that their composition implements the $(2N+1)$-element queue specified by $QE^{[\mathrm{dbl}]} \stackrel{+}{-\!\!\triangleright} QM^{[\mathrm{dbl}]}$.

The obvious thing to try to prove is

$$
\begin{array}{l}
(QE^{[1]} \stackrel{+}{-\!\!\triangleright} QM^{[1]}) \ \land\ (QE^{[2]} \stackrel{+}{-\!\!\triangleright} QM^{[2]}) \\
\qquad \Rightarrow\ (QE^{[\mathrm{dbl}]} \stackrel{+}{-\!\!\triangleright} QM^{[\mathrm{dbl}]})
\end{array} \tag{3}
$$

We could prove this had we used a noninterleaving representation of the queue. However, (3) is not valid for an interleaving representation, for the following reason. The specification of the first queue does not mention o, and that of the second queue does not mention i. The conjunction of the two specifications allows an enqueue action of the first queue and a dequeue action of the second queue to happen simultaneously, a step that changes $i.ack$ and $o.snd$ simultaneously. But, in an interleaving representation, the $(2N+1)$-element queue's guarantee does not allow such a step, so (3) must be invalid. Another problem with (3) is that the conjunction of the component queues' specifications allows a step that changes $z.snd$ and $o.ack$ simultaneously. Such a step satisfies the $(2N+1)$-element queue's environment assumption $QE^{[\mathrm{dbl}]}$, which does not mention z, so (3) asserts that the next step must satisfy its guarantee $QM^{[\mathrm{dbl}]}$. However, a step that changes both $z.snd$ and $o.ack$ violates the second component queue's environment assumption $QE^{[2]}$, permitting the component queue to make arbitrary changes to $o.snd$ in the next step. A similar problem is caused by simultaneous changes

1. $\mathcal{C}(QE^{[\mathrm{dbl}]}) \wedge \mathcal{C}(G) \wedge \mathcal{C}(QM^{[1]}) \wedge \mathcal{C}(QM^{[2]}) \;\Rightarrow\; QE^{[1]} \wedge QE^{[2]}$

   PROOF: We use Propositions 2 and 1 to remove the quantifiers and closure operators from the left-hand side of the implication. The resulting formula then asserts that a complete system, consisting of the safety parts of the two queues (with their internal state visible) together with the environment, implements $QE^{[1]}$ and $QE^{[2]}$. The proof of this formula is straightforward.

2. $\mathcal{C}(QE^{[\mathrm{dbl}]})_{+\langle i,\,o,\,z\rangle} \wedge \mathcal{C}(QM^{[1]}) \wedge \mathcal{C}(G) \wedge \mathcal{C}(QM^{[2]}) \;\Rightarrow\; \mathcal{C}(QM^{[\mathrm{dbl}]})$

   2.1. $\mathcal{C}(G) \wedge \mathcal{C}(QM^{[1]}) \wedge \mathcal{C}(QM^{[2]}) \;\Rightarrow\; \mathcal{C}(QE^{[\mathrm{dbl}]}) \perp \mathcal{C}(QM^{[\mathrm{dbl}]})$

   2.1.1. $\mathcal{C}(IQM^{[1]}) \wedge \mathcal{C}(IQM^{[2]}) \;\Rightarrow\; \exists\, q_1,\, q_2 : Init_M^{[1]} \wedge Init_M^{[2]}$

   PROOF: Follows easily from Proposition 1 and the definitions.

   2.1.2. $\mathcal{C}(QM^{[1]}) \wedge \mathcal{C}(QM^{[2]}) \;\Rightarrow\; \exists\, q_1,\, q_2 : Init_M^{[1]} \wedge Init_M^{[2]}$

   PROOF: 2.1.1 and Proposition 2 (since any predicate is a safety property).

   2.1.3. Q.E.D.

   PROOF: 2.1.2, the definition of G, and Proposition 4 (since disjointness is a safety property).

   2.2. $\mathcal{C}(QE^{[\mathrm{dbl}]}) \wedge \mathcal{C}(G) \wedge \mathcal{C}(QM^{[1]}) \wedge \mathcal{C}(QM^{[2]}) \;\Rightarrow\; \mathcal{C}(QM^{[\mathrm{dbl}]})$

   PROOF: We use Propositions 2 and 1 to remove the quantifiers and closures from the formula. The resulting formula is proved when proving the safety part of step 3.

   2.3. Q.E.D.

   PROOF: 2.1, 2.2, and Proposition 3.

3. $QE^{[\mathrm{dbl}]} \wedge G \wedge QM^{[1]} \wedge QM^{[2]} \;\Rightarrow\; QM^{[\mathrm{dbl}]}$

   PROOF: A direct calculation shows that the left-hand side of the implication implies CDQ, the complete-system specification of the double queue. We already observed in Section 6.4 that CDQ implements $CQ^{[\mathrm{dbl}]}$, which equals $QE^{[\mathrm{dbl}]} \wedge QM^{[\mathrm{dbl}]}$.

4. Q.E.D.

   PROOF: 1–3 and the Composition Theorem, substituting

$$\begin{array}{llll} M_1 \leftarrow G & M_2 \leftarrow QM^{[1]} & M_3 \leftarrow QM^{[2]} & M \leftarrow QM^{[\mathrm{dbl}]} \\ E_1 \leftarrow \mathsf{true} & E_2 \leftarrow QE^{[1]} & E_3 \leftarrow QE^{[2]} & E \leftarrow QE^{[\mathrm{dbl}]} \end{array}$$

Figure 9: Proof sketch of (4).

to $i.snd$ and $z.ack$.

We prove that the composition implements the larger queue under the assumption that the outputs of two different components do not change simultaneously. Thus, we prove

$$\begin{array}{l} G \;\wedge\; (QE^{[1]} \stackrel{+}{\Rightarrow} QM^{[1]}) \;\wedge\; (QE^{[2]} \stackrel{+}{\Rightarrow} QM^{[2]}) \\ \quad \Rightarrow\; (QE^{[\mathrm{dbl}]} \stackrel{+}{\Rightarrow} QM^{[\mathrm{dbl}]}) \end{array} \tag{4}$$

where G is the formula

$$Disjoint(\langle i.snd, o.ack\rangle, \langle z.snd, i.ack\rangle, \langle o.snd, z.ack\rangle)$$

The proof is outlined in Figure 9.

7 Conclusion

We have shown how to write assumption/guarantee specifications in TLA: we simply specify the environment and the system as separate components, and combine the specifications with the operator $\stackrel{+}{\Rightarrow}$. We compose assumption/guarantee specifications with the familiar operator $\wedge$. The Composition Theorem then gives a rule for proving properties of large systems by reasoning about their components; it deals with both fairness properties and hiding.

Our work was preceded by results in a long list of publications, described next. Like ours, most previous composition theorems were strong, in the sense that they could handle circularities in reasoning about safety properties. Our approach differs from previous ones in its general treatment of fairness properties and hiding.

The first strong composition theorem we know is that of Misra and Chandy [12], who considered safety properties of processes communicating by means of CSP primitives. They wrote assumption/guarantee specifications as Hoare triples containing assertions about history variables. Pandya and Joseph [13] extended this approach to handle some liveness properties. Pnueli [14] was the first to use temporal logic to write assumption/guarantee specifications. He had a strong composition theorem for safety properties with no hiding. To handle liveness, he wrote assumption/guarantee specifications with implication instead of $\stackrel{+}{-\!\!\triangleright}$, so he did not obtain a strong composition theorem. Stark [15] also wrote assumption/guarantee specifications as implications of temporal formulas and required that circularity be avoided. Our earlier work [3] was semantic, in a more complicated model with agents. It lacked practical proof rules for handling fairness and hiding. Collette [7] adapted this work to Unity. Abadi and Plotkin [5] used a propositional logic with agents, and considered only safety properties.

So far, we have applied our Composition Theorem only to toy examples. Formal reasoning about systems is still rare, and it generally occurs on a case-by-case basis. When the specification of a component is used only to verify a specific system, there is no need for a general assumption/guarantee specification. For most practical applications, decomposition suffices. When decomposition does not suffice, the Composition Theorem makes reasoning about open systems almost as easy as reasoning about complete ones.

References

[1] Martín Abadi and Leslie Lamport. Decomposing specifications of concurrent systems. To appear in *PROCOMET '94*, Proceedings of the IFIP Working Conference on Programming Concepts, Methods and Calculi.

[2] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. Research Report 91, Digital Equipment Corporation, Systems Research Center, 1992. An earlier version, without proofs, appeared in [8, pages 1–27].

[3] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[4] Martín Abadi and Leslie Lamport. Conjoining specifications. Research Report 118, Digital Equipment Corporation, Systems Research Center, 1993.

[5] Martín Abadi and Gordon Plotkin. A logical view of composition and refinement. *Theoretical Computer Science*, 114(1):3–30, June 1993.

[6] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[7] Pierre Collette. Application of the composition principle to Unity-like specifications. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 230–242, Berlin, 1993. Springer-Verlag.

[8] J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992. Proceedings of a REX Real-Time Workshop, held in The Netherlands in June, 1991.

[9] Cliff B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP, North-Holland, September 1983.

[10] Leslie Lamport. The temporal logic of actions. Research Report 79, Digital Equipment Corporation, Systems Research Center, December 1991. To appear in *Transactions on Programming Languages and Systems*.

[11] Carver Mead and Lynn Conway. *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, Reading, Massachusetts, 1980.

[12] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, July 1981.

[13] Paritosh K. Pandya and Mathai Joseph. P-A logic—a compositional proof system for distributed programs. *Distributed Computing*, 5(1):37–54, 1991.

[14] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, NATO ASI Series, pages 123–144. Springer-Verlag, October 1984.

[15] Eugene W. Stark. A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 369–391, Berlin, 1985. Springer-Verlag.