Decomposing Specifications of Concurrent Systems*

Martín Abadi and Leslie Lamport

Systems Research Center, Digital Equipment Corporation
130 Lytton Avenue, Palo Alto, CA 94301, U.S.A.

We introduce a simple method for specifying individual components of a concurrent system. The specification of the system is the conjunction of its components' specifications. We show how to prove properties of the system by reasoning about its components. Our approach is useful in substantial verification problems.

Keyword Codes: D.2.4; F.3.1
Keywords: Theory; Verification

1. INTRODUCTION

Large systems are built from smaller parts. We present a method for deducing properties of a system by reasoning about its components. We show how to represent an individual component Π_i by a formula S_i so that the parallel composition usually denoted **cobegin** $\Pi_1 \parallel \ldots \parallel \Pi_n$ **coend** is represented by the formula $S_1 \wedge \ldots \wedge S_n$. Composition is conjunction.

We reduce composition to conjunction not for the sake of elegance, but because it is the best way we know to prove properties of composite systems. Rigorous reasoning requires logic, and hence a language of logical formulas. It does not require a conventional programming language for describing systems. We find it most convenient to regard programs and circuit descriptions as low-level specifications, and to represent them in the same logic used for higher-level specifications. The logic we use is TLA, the Temporal Logic of Actions [14]. We do not discuss here the important problem of translating from a low-level TLA specification to an implementation in a conventional language.

The idea of representing concurrent programs and their specifications as formulas in a temporal logic was first proposed by Pnueli [17]. It was later observed that, if specifications allow "stuttering" steps that leave the state unchanged, then $S_l \Rightarrow S_h$ asserts that S_l implements S_h [12]. Hence, proving that a lower-level specification implements a higher-level one was reduced to proving a formula in the logic. Still later, it was noticed that the formula $\exists x : S$ specifies the same system as S except with the variable x hidden [1,13], and variable hiding became logical quantification. The idea of composition as conjunction has also been suggested [5,6,20], but our method for reducing composition to conjunction is new.

*This article was typeset using a LaTeX document style provided by Elsevier.

Composite specifications arise in two ways: by *composing* given parts to form a larger system, and by *decomposing* a given system into smaller parts. These two situations call for two methods of writing component specifications that differ in their treatment of the component's environment. This difference in turn leads to different proof rules. Here, we consider only decomposition.

When decomposing a specification, the environment of each component is assumed to be the other components, and is usually left implicit. To reason about a component, we must state what we are assuming about its environment, and then prove that this assumption is satisfied by the other components. The Decomposition Theorem of Section 5 provides the needed proof rule. It reduces the verification of a complex, low-level system to proving properties of a higher-level specification and properties of one low-level component at a time. Decomposing proofs in this way allows us to apply decision procedures to verifications that hitherto required completely hand-guided proofs [11].

In the next section, we examine the issues that arise in decomposition. Our discussion is informal, because we wish to show that these issues are fundamental, not artifacts of a particular programming language or formalism. Section 3 covers the formal preliminaries, Section 4 investigates a method of writing specifications of components, and Section 5 gives the Decomposition Theorem. Proofs appear in [4].

2. AN INFORMAL OVERVIEW

A complete system is one that is self-contained; it may be observed, but it does not interact with the observer. A program is a complete system, provided we model inputs as being generated nondeterministically by the program itself.

As a tiny example of a complete system, consider the following program, written in an informal programming-language notation in which statements within angle brackets are executed atomically.

<u>Program GCD</u>
var a **initially** 233344, b **initially** 233577899 ;
cobegin **loop** $\langle$ **if** $a > b$ **then** $a := a - b$ $\rangle$ **endloop**
$\qquad\qquad\|$
loop $\langle$ **if** $b > a$ **then** $b := b - a$ $\rangle$ **endloop** **coend**

Program GCD satisfies the correctness property that eventually a and b become and remain equal to the gcd of 233344 and 233577899. We make no distinction between programs and properties, writing them all as TLA formulas. If formula M_{gcd} represents program GCD and formula P_{gcd} represents the correctness property, then the program implements the property iff (if and only if) M_{gcd} implies P_{gcd}. Thus, correctness of program GCD is verified by proving $M_{gcd} \Rightarrow P_{gcd}$.

In hierarchical development, one decomposes the specification of a system into specifications of its parts. As explained in Section 4, the specification M_{gcd} of program GCD can be written as $M_a \wedge M_b$, where M_a asserts that a initially equals 233344 and is repeatedly decremented by the value of b whenever $a > b$, and where M_b is analogous. The formulas M_a and M_b are the specifications of two processes Π_a and Π_b. We can write Π_a and Π_b as

<u>Process Π_a</u>
output var a **initially** 233344 ;
input var b ;
loop $\langle$ **if** $a > b$ **then** $a := a - b$ $\rangle$
endloop

<u>Process Π_b</u>
output var b **initially** 233577899 ;
input var a ;
loop $\langle$ **if** $b > a$ **then** $b := b - a$ $\rangle$
endloop

One decomposes a specification in order to refine the components separately. We can refine the GCD program, to remove simultaneous atomic accesses to both a and b, by refining process Π_a to

<u>Process Π^l_a</u>
output var a **initially** 233344 ;
internal var ai ;
input var b ;
loop $\langle\, ai := b \,\rangle$; **if** $\langle\, a > ai \,\rangle$ **then** $\langle\, a := a - ai \,\rangle$ **endloop**

and refining Π_b to the analogous process Π^l_b.

The composition of processes Π^l_a and Π^l_b correctly implements program GCD. This is expressed in TLA by the assertion that $M^l_a \wedge M^l_b$ implies $M_a \wedge M_b$, where M^l_a and M^l_b are the formulas representing Π^l_a and Π^l_b.

We would like to decompose the proof of $M^l_a \wedge M^l_b \Rightarrow M_a \wedge M_b$ into proofs of $M^l_a \Rightarrow M_a$ and $M^l_b \Rightarrow M_b$. These proofs would show that Π^l_a implements Π_a and Π^l_b implements Π_b.

Unfortunately, Π^l_a does not implement Π_a because, in the absence of assumptions about when its input b can change, Π^l_a can behave in ways that process Π_a cannot. Process Π_a can decrement a only by the current value of b, but Π^l_a can decrement a by a previous value of b if b changes between the assignment to ai and the assignment to a. Similarly, Π^l_b does not implement Π_b.

Process Π^l_a does correctly implement process Π_a in a context in which b does not change when $a > b$. This is expressed in TLA by the formula $E_a \wedge M^l_a \Rightarrow M_a$, where E_a asserts that b does not change when $a > b$. Similarly, $E_b \wedge M^l_b \Rightarrow M_b$ holds, for the analogous E_b. The Decomposition Theorem of Section 5 allows us to deduce $M^l_a \wedge M^l_b \Rightarrow M_a \wedge M_b$ from approximately the following hypotheses:

$$\begin{aligned} &E_a \wedge M^l_a \Rightarrow M_a \\ &E_b \wedge M^l_b \Rightarrow M_b \\ &M_a \wedge M_b \Rightarrow E_a \wedge E_b \end{aligned} \tag{1}$$

The third hypothesis holds because the composition of processes Π_a and Π_b does not allow a to change when $b > a$ or b to change when $a > b$.

Observe that E_a asserts only the property of Π^l_b needed to guarantee that Π^l_a implements Π_a. In a more complicated example, E_a will be significantly simpler than M^l_b, the full specification of Π^l_b. Verifying these hypotheses will therefore be easier than proving $M^l_a \wedge M^l_b \Rightarrow M_a \wedge M_b$ directly, since this proof requires reasoning about the specification $M^l_a \wedge M^l_b$ of the complete low-level program.

One cannot really deduce $M^l_a \wedge M^l_b \Rightarrow M_a \wedge M_b$ from the hypotheses (1). For example, (1) is trivially satisfied if E_a, E_b, M_a, and M_b all equal false; but we cannot deduce

$M_a^l \wedge M_b^l \Rightarrow \mathsf{false}$ for arbitrary M_a^l and M_b^l. The precise hypotheses of the Decomposition Theorem are more complicated, and we must develop a number of formal concepts in order to state them. We also develop results that allow us to discharge these more complicated hypotheses by proving conditions essentially as simple as (1).

3. PRELIMINARIES

3.1. TLA: a brief introduction

3.1.1. Review of the syntax and semantics

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 $\exists\!\!\!\exists x_1 : \ldots \exists\!\!\!\exists x_k : F$.

The standard way of specifying a system in TLA is with a formula in the "canonical form" $\exists\!\!\!\exists x : \mathit{Init} \wedge \Box[\mathcal{N}]_v \wedge L$, where Init is a predicate and L a conjunction of fairness conditions. 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 the state function v unchanged, and L holds. For example, the specification M_{gcd} of the complete high-level GCD program is written in canonical form by taking[1]

---

[1]A list of formulas bulleted with $\wedge$ or $\vee$ denotes the conjunction or disjunction of the formulas, using indentation to eliminate parentheses; $\Rightarrow$ has lower precedence than the other Boolean operators.

$$
\begin{array}{lcl}
\mathit{Init} & \triangleq & (a = 233344) \land (b = 233577899) \\
\mathcal{N} & \triangleq & \lor\ (a > b)\ \land\ (a' = a - b)\ \land\ (b' = b) \\
 & & \lor\ (b > a)\ \land\ (b' = b - a)\ \land\ (a' = a) \\
v & \triangleq & \langle a,\, b \rangle \\
L & \triangleq & \mathrm{WF}_v(\mathcal{N})
\end{array}
\tag{2}
$$

3.1.2. Implementation and composition

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.

A specification M^l implies a specification M iff every behavior that satisfies M^l also satisfies M, hence proving $M^l \Rightarrow M$ shows that the system Π^l represented by M^l implements the system or property Π represented by M. The formula $M^l \Rightarrow M$ is proved by applying a handful of simple rules [14]. When M has the form $\exists\, x : \widehat{M}$, a key step in the proof is finding a *refinement mapping*—a tuple of state functions $\overline{x}$ such that M^l implies $\overline{\widehat{M}}$, where $\overline{\widehat{M}}$ is the formula obtained by substituting $\overline{x}$ for x in $\widehat{M}$. Under reasonable assumptions, such a refinement mapping exists when $M^l \Rightarrow \exists\, x : \widehat{M}$ is valid [1].

Composing two systems means constructing a universe in which they are both running. If formulas M_1 and M_2 represent the two systems, then $M_1 \land M_2$ represents their composition, since a behavior represents a possible history of a universe containing both systems iff it satisfies both M_1 and M_2. Thus, in principle, composition is conjunction. We show in Section 4 that composition is conjunction in practice as well.

3.2. Safety and closure

3.2.1. Definition of 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. For convenience, we say that the empty sequence $\langle\,\rangle$ satisfies every formula.

A *safety property* is a formula that is satisfied by an infinite behavior σ iff it is satisfied by every prefix of σ [7]. For any predicate *Init*, action $\mathcal{N}$, and state function v, the formula $\mathit{Init} \land \Box[\mathcal{N}]_v$ is a safety property. 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 $\models F \Rightarrow \mathcal{C}(F)$.

3.2.2. Machine closure

When writing a specification in the form $\mathit{Init} \land \Box[\mathcal{N}]_v \land L$, we expect L to constrain infinite behaviors, not finite ones. Formally, this means that the closure of $\mathit{Init} \land \Box[\mathcal{N}]_v \land L$ should be $\mathit{Init} \land \Box[\mathcal{N}]_v$. A pair of properties $(P,\, L)$ is called *machine closed* iff $\mathcal{C}(P \land L)$ equals P [1]. (We often say informally that $P \land L$ is machine closed.)

Proposition 1 below, which is proved in [2], shows that we can use fairness properties to write machine-closed specifications. The proposition relies on the following definition: an action $\mathcal{A}$ is a *subaction* of a safety property P iff for every finite behavior $\rho = \langle r_0, \ldots, r_n \rangle$, if ρ satisfies P and $\mathcal{A}$ is enabled in state r_n, then there exists a state r_{n+1} such that

$\langle r_0, \ldots, r_{n+1}\rangle$ satisfies P and $\langle r_n, r_{n+1}\rangle$ is an $\mathcal{A}$ step. If $\mathcal{A}$ implies $\mathcal{N}$, then $\mathcal{A}$ is a subaction of $Init \wedge \Box[\mathcal{N}]_v$.

Proposition 1 *If P is a safety property and L is the conjunction of a countable number of formulas of the form $\mathrm{WF}_w(\mathcal{A})$ and/or $\mathrm{SF}_w(\mathcal{A})$ such that $\mathcal{A} \wedge (w' \neq w)$ is a subaction of P, then (P, L) is machine closed.*

3.2.3. Closure and hiding

To apply the Decomposition Theorem, we must prove formulas of the form $\mathcal{C}(M_1) \wedge \ldots \wedge \mathcal{C}(M_n) \Rightarrow \mathcal{C}(M)$. The obvious first step in proving such a formula is to compute the closures $\mathcal{C}(M_1)$, ... , $\mathcal{C}(M_n)$, 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, ..., 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 $i \neq j$.*
If $\models \bigwedge_{i=1}^{n} \mathcal{C}(M_i) \;\Rightarrow\; \exists\, x : \mathcal{C}(M)$, then $\models \bigwedge_{i=1}^{n} \mathcal{C}(\exists\, x_i : M_i) \;\Rightarrow\; \mathcal{C}(\exists x : M)$.

4. DECOMPOSITION

4.1. Interleaving and noninterleaving representations

When representing a history of the universe as a behavior, we can describe concurrent changes to two objects ξ and ψ either by a single simultaneous change to the corresponding variables x and y, or by separate changes to x and y in some order. If the changes to ξ and ψ are directly linked, then it is usually most convenient to describe their concurrent change by a single change to both x and y. However, if the changes are independent, then we are free to choose whether or not to allow simultaneous changes to x and y. An *interleaving* representation is one in which such simultaneous changes are disallowed.

When changes to ξ and ψ are directly linked, we often think of x and y as output variables of a single component. An interleaving representation is then one in which simultaneous changes to output variables of different processes are disallowed. The absence of such simultaneous changes can be expressed as a TLA formula. For a system with n components in which v_i is the tuple of output variables of component i, interleaving is expressed by the formula

$$Disjoint(v_1, \ldots, v_n) \;\triangleq\; \bigwedge_{i \neq j} \Box[(v_i' = v_i) \vee (v_j' = v_j)]_{\langle v_i, v_j\rangle}$$

We have found that, in TLA, interleaving representations are usually easier to write and to reason about. Moreover, an interleaving representation is adequate for reasoning about a system if the system is modeled at a sufficiently fine grain of atomicity. However, TLA also works for noninterleaving representations. TLA does not mandate any particular method for representing systems. Indeed, one can write specifications that are intermediate between interleaving and noninterleaving representations.

4.2. Specifying a component

Let us consider how to write the specification M of one component of a larger system. We assume that the free variables of the specification can be partitioned into tuples m of output variables and e of input variables, where the component changes the values of the variables of m only. (A more general situation is discussed below.) The specification of a component has the same form $\exists x : Init \wedge \Box[\mathcal{N}]_v \wedge L$ as that of a complete system. For a component specification:

v is the tuple $\langle x, m, e\rangle$.

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

$\mathcal{N}$ should allow two kinds of steps—ones that the component performs, and ones that its environment performs. Steps performed by the component, which change its output variables m, are described by an action $\mathcal{N}_m$. In an interleaving representation, the component's inputs and outputs cannot change simultaneously, so $\mathcal{N}_m$ implies $e' = e$. In a noninterleaving representation, $\mathcal{N}_m$ does not constrain the value of e', so the variables of e do not appear primed in $\mathcal{N}_m$. In either case, we are specifying the component but not its environment, so we let the environment do anything except change the component's output variables or internal variables. In other words, the environment is allowed to perform any step in which $\langle m, x\rangle'$ equals $\langle m, x\rangle$. (Below, we describe more general specifications in which an environment action can change x.) Therefore, $\mathcal{N}$ should equal $\mathcal{N}_m \vee (\langle m, x\rangle' = \langle m, x\rangle)$.

L is the conjunction of fairness conditions of the form $\mathrm{WF}_{\langle m, x\rangle}(\mathcal{A})$ and $\mathrm{SF}_{\langle m, x\rangle}(\mathcal{A})$. For an interleaving representation, which by definition does not allow steps that change both e and m, the subscripts $\langle m, x\rangle$ and $\langle e, m, x\rangle$ yield equivalent fairness conditions.

This leads us to write M in the form

$$M \triangleq \exists x : Init \wedge \Box[\mathcal{N}_m \vee (\langle m, x\rangle' = \langle m, x\rangle)]_{\langle e, m, x\rangle} \wedge L \tag{3}$$

By simple logic, (3) is equivalent to

$$M \triangleq \exists x : Init \wedge \Box[\mathcal{N}_m]_{\langle m, x\rangle} \wedge L \tag{4}$$

For the specification M_a of process Π_a in the GCD example, x is the empty tuple (there is no internal variable), the input variable e is b, the output variable m is a, and

$$\begin{aligned} Init_a &\triangleq a = 233344 \\ \mathcal{N}_a &\triangleq (a > b) \wedge (a' = a - b) \wedge (b' = b) \\ M_a &\triangleq Init_a \wedge \Box[\mathcal{N}_a]_a \wedge \mathrm{WF}_a(\mathcal{N}_a) \end{aligned} \tag{5}$$

For the specification M_a^l of the low-level process Π_a^l, the tuple x is $\langle ai, pca\rangle$, where *pca* is an internal variable that tells whether control is at the beginning of the loop or after the assignment to *ai*. The specification has the form

$$M_a^l \triangleq \exists\, ai, pca : Init_a^l \wedge \Box[\mathcal{N}_a^l]_{\langle a, ai, pca\rangle} \wedge \mathrm{WF}_{\langle a, ai, pca\rangle}(\mathcal{N}_a^l) \tag{6}$$

for appropriate initial condition $Init^l_a$ and next-state action $\mathcal{N}^l_a$. The specifications M_b and M^l_b are similar.

In describing the component's next-state action $\mathcal{N}$, we required that an environment action not change the component's internal variables. One can also write a specification in which the component records environment actions by changing its own internal variables. In this case, $\mathcal{N}$ will not equal $\mathcal{N}_m \;\vee\; (\langle m, x\rangle' = \langle m, x\rangle)$, but may just imply $(e' = e) \vee (m' = m)$. The resulting formula will not be a pure interleaving specification because environment actions can change the component's variables, but no action can change both the component's and the environment's output variables. We have not explored this style of specification.

We have been assuming that the visible variables of the component's specification can be partitioned into tuples m of output variables and e of input variables. To see how to handle a more general case, let μ_M be the action $m' \neq m$, let v equal $\langle e, m\rangle$, and observe that $[\mathcal{N}_M]_{\langle m, x\rangle}$ equals $[\mathcal{N}_M \vee (\neg\mu_M \wedge (x' = x))]_{\langle v, x\rangle}$. A μ_M step is one that is attributed to the component, since it changes the component's output variables. When the tuple v of variables is not partitioned into input and output variables, we define an action μ_M that specifies what steps are attributed to the component, and we write the component's next-state action in the form $\mathcal{N}_M \vee (\neg\mu_M \wedge (x' = x))$. All our results for separate input and output variables can be generalized by writing the next-state action in this form. However, for simplicity, we consider only the special case.

4.3. Conjoining components to form a complete system

A complete system is the composition of its components. For composition really to be conjunction, the conjunction of the specifications of all components should equal the expected specification of the complete system. The following proposition shows that this is so for interleaving representations.

Proposition 3 *Let $m_1, \ldots, m_n$, $x_1, \ldots, x_n$ be tuples of variables, and let*

$$\begin{aligned} m &\triangleq \langle m_1, \ldots, m_n\rangle \qquad x \triangleq \langle x_1, \ldots, x_n\rangle \\ \widehat{x}_i &\triangleq \langle x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\rangle \\ M_i &\triangleq \exists\, x_i : Init_i \;\wedge\; \Box[\mathcal{N}_i]_{\langle m_i, x_i\rangle} \;\wedge\; L_i \end{aligned}$$

If, for all i, $j = 1, \ldots, n$ with $i \neq j$:

1. *no variable of x_j occurs free in x_i or M_i.*
2. *m includes all free variables of M_i.*
3. $\models \mathcal{N}_i \Rightarrow (m'_j = m_j)$

then

$$\models \bigwedge_{i=1}^{n} M_i \;=\; \exists\, x : \bigwedge_{i=1}^{n} Init_i \;\wedge\; \Box[\bigvee_{i=1}^{n} \mathcal{N}_i \wedge (\widehat{x}'_i = \widehat{x}_i)]_{\langle m, x\rangle} \;\wedge\; \bigwedge_{i=1}^{n} L_i$$

In this proposition, hypothesis 3 asserts that component i leaves the variables of other components unchanged, so M_i is an interleaving representation of component i. Hence,

M_i implies $\mathit{Disjoint}(m_i, m_j)$, for each $j \neq i$, and $\bigwedge_{i=1}^n M_i$ implies $\mathit{Disjoint}(m_1, \ldots, m_n)$, as expected for an interleaving representation of the complete system.

In the GCD example, we apply this proposition to the formula M_a of (5) and the analogous formula M_b. We immediately get that $M_a \wedge M_b$ is equivalent to a formula that is the same as M_{gcd}, defined by (2), except with $\mathrm{WF}_{\langle a,\, b\rangle}(\mathcal{N}_a) \wedge \mathrm{WF}_{\langle a,\, b\rangle}(\mathcal{N}_b)$ instead of $\mathrm{WF}_{\langle a,\, b\rangle}(\mathcal{N})$. It can be shown that these two fairness conditions are equivalent; hence, $M_a \wedge M_b$ is equivalent to M_{gcd}.

Hypothesis 3 of Proposition 3 is satisfied only by interleaving representations. For arbitrary representations, a straightforward calculation shows

$$\models \bigwedge_{i=1}^{n} M_i \;=\; \exists\, x : \begin{array}[t]{l} \wedge\ \bigwedge_{i=1}^n \mathit{Init}_i \\ \wedge\ \Box[\bigwedge_{i=1}^n (\mathcal{N}_i \vee \langle m_i,\, x_i\rangle' = \langle m_i,\, x_i\rangle)]_{\langle m,\, x\rangle} \\ \wedge\ \bigwedge_{i=1}^n L_i \end{array} \tag{7}$$

assuming only the first hypothesis of the proposition. The right-hand side has the expected form for a noninterleaving specification, since it allows $\mathcal{N}_i \wedge \mathcal{N}_j$ steps for $i \neq j$. Hence, composition is conjunction for noninterleaving representations too.

5. THE DECOMPOSITION THEOREM

5.1. An additional temporal operator

The simplest statement of our decomposition theorem requires the introduction of one more temporal construct: 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.

Although we have defined it semantically, E_{+v} can be expressed in terms of the primitive TLA operations $'$, $\Box$, and $\exists$. When E is a safety property in canonical form, it is easy to write E_{+v} explicitly:

Proposition 4 *If x is a tuple of variables none of which occurs in v, and s is a variable that does not occur in Init, $\mathcal{N}$, w, v, or x, and*

$$\begin{array}{lcl} \widehat{\mathit{Init}} & \triangleq & (\mathit{Init} \wedge (s = 0)) \vee (\neg \mathit{Init} \wedge (s = 1)) \\ \widehat{\mathcal{N}} & \triangleq & \vee\ (s = 0)\ \wedge \begin{array}[t]{l} \vee\ (s' = 0) \wedge (\mathcal{N} \vee (w' = w)) \\ \vee\ (s' = 1) \wedge \neg(\mathcal{N} \vee (w' = w)) \end{array} \\ & & \vee\ (s = 1)\ \wedge\ (s' = 1) \wedge (v' = v) \end{array}$$

then $\models (\exists\, x : \mathit{Init} \wedge \Box[\mathcal{N}]_w)_{+v} \;=\; \exists\, x, s : \widehat{\mathit{Init}} \wedge \Box[\widehat{\mathcal{N}}]_{\langle w,\, v,\, s\rangle}$.

We need to reason about $+$ only to verify hypotheses of the form $\models \mathcal{C}(E)_{+v} \wedge \mathcal{C}(M^l) \Rightarrow \mathcal{C}(M)$ in our Decomposition Theorem. We can verify such a hypothesis by first applying the observation that $\mathcal{C}(E)_{+v}$ equals $\mathcal{C}(E_{+v})$ and using Proposition 4 to calculate E_{+v}. However, this approach is necessary only for noninterleaving specifications. The next proposition provides a way of proving these hypotheses for interleaving specifications without having to calculate E_{+v}. The crucial third formula of the hypothesis is easy to check when M^l is an interleaving specification.

Proposition 5 *Let v be a tuple of variables that includes all variables in $\widehat{M}$,*

$\models \mathcal{C}(\widehat{E}) \ = \ Init_E \ \wedge \ \Box[\mathcal{N}_E]_{\langle x,\, e\rangle}$,

$\models \mathcal{C}(\widehat{M}) \ = \ Init_M \ \wedge \ \Box[\mathcal{N}_M]_{\langle y,\, m\rangle}$, *and*

$\models \mathcal{C}(M^l) \ \Rightarrow \ (\exists x : Init_E \vee \exists y : Init_M) \ \wedge \ Disjoint(e,\, m)$.

If $\quad \models \mathcal{C}(\boldsymbol{\exists}\, x : \widehat{E}) \ \wedge \ \mathcal{C}(M^l) \ \Rightarrow \ \mathcal{C}(\boldsymbol{\exists}\, y : \widehat{M})$

then $\ \models \mathcal{C}(\boldsymbol{\exists}\, x : \widehat{E})_{+v} \ \wedge \ \mathcal{C}(M^l) \ \Rightarrow \ \mathcal{C}(\boldsymbol{\exists}\, y : \widehat{M})$.

5.2. The basic theorem

Consider a complete system decomposed into components Π_i. We would like to prove that this system is implemented by a lower-level one, consisting of components Π_i^l, by proving that each Π_i^l implements Π_i. Let M_i be the specification of Π_i and M_i^l be the specification of Π_i^l. We must prove that $\bigwedge_{i=1}^n M_i^l$ implies $\bigwedge_{i=1}^n M_i$. This implication is trivially true if M_i^l implies M_i, for all i. However, as we saw in the GCD example, M_i^l need not imply M_i.

Even when $M_i^l \Rightarrow M_i$ does not hold, we need not reason about all the lower-level components together. Instead, we prove $E_i \wedge M_i^l \Rightarrow M_i$, where E_i includes just the properties of the other components assumed by component i, and is usually much simpler than $\bigwedge_{k \neq i} M_k^l$. Proving $E_i \wedge M_i^l \Rightarrow M_i$ involves reasoning only about component i, not about the entire lower-level system.

In propositional logic, to deduce that $\bigwedge_{i=1}^n M_i^l$ implies $\bigwedge_{i=1}^n M_i$ from $\bigwedge_{i=1}^n (E_i \wedge M_i^l \Rightarrow M_i)$, we may prove that $\bigwedge_{k=1}^n M_k^l$ implies E_i for each i. However, proving this still requires reasoning about $\bigwedge_{k=1}^n M_k^l$, the specification of the entire lower-level system. The following theorem shows that we need only prove that E_i is implied by $\bigwedge_{k=1}^n M_k$, the specification of the higher-level system—a formula usually much simpler than $\bigwedge_{k=1}^n M_k^l$.

Proving $E_i \wedge M_i^l \Rightarrow M_i$ and $(\bigwedge_{k=1}^n M_k) \Rightarrow E_i$ for each i and deducing $(\bigwedge_{i=1}^n M_i^l) \Rightarrow (\bigwedge_{i=1}^n M_i)$ is circular reasoning, and is not sound in general. Such reasoning would allow us to deduce $(\bigwedge_{i=1}^n M_i^l) \Rightarrow (\bigwedge_{i=1}^n M_i)$ for any M_i^l and M_i—simply let E_i equal M_i. To break the circularity, we need to add some $\mathcal{C}$'s and one hypothesis: if E_i is ever violated then, for at least one additional step, M_i^l implies M_i. This hypothesis is expressed formally as $\models \mathcal{C}(E_i)_{+v} \wedge \mathcal{C}(M_i^l) \ \Rightarrow \ \mathcal{C}(M_i)$, for some v; the hypothesis is weakest when v is taken to be the tuple of all relevant variables. Our proof rule is:

Theorem 1 (Decomposition Theorem) *If, for $i = 1, \ldots, n$,*

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

2. (a) $\models \mathcal{C}(E_i)_{+v} \wedge \mathcal{C}(M_i^l) \ \Rightarrow \ \mathcal{C}(M_i)$

   (b) $\models E_i \wedge M_i^l \ \Rightarrow \ M_i$

then $\models \displaystyle\bigwedge_{i=1}^{n} M_i^l \ \Rightarrow \ \bigwedge_{i=1}^{n} M_i$.

In the GCD example, we can use the theorem to prove $M_a^l \wedge M_b^l \Rightarrow M_a \wedge M_b$. (The component specifications are described in Section 4.2.) The abstract environment specification

E_a asserts that b can change only when $a < b$, and that a is not changed by steps that change b. Thus,

$$E_a \;\triangleq\; \Box[(a < b) \wedge (a' = a)]_b$$

The definition of E_b is analogous. We let v be $\langle a, b\rangle$.

In general, the environment and component specifications can have internal variables. The theorem also allows them to contain fairness conditions. However, hypothesis 1 asserts that the E_i are implied by safety properties. In practice, this means that the theorem can be applied only when the E_i are safety properties. Examples indicate that, in general, compositional reasoning is possible only when the environment conditions are safety properties.

5.3. Verifying the hypotheses

We now discuss how one verifies the hypotheses of the Decomposition Theorem, illustrating the method with the GCD example.

To prove the first hypothesis, one first uses Propositions 1 and 2 to eliminate the closure operators and existential quantifiers, reducing the hypothesis to a condition of the form

$$\models \bigwedge_{i=1}^{n} (Init_i \wedge \Box[\mathcal{N}_i]_{v_i}) \Rightarrow E_i \tag{8}$$

For interleaving representations, we can then use Proposition 3 to write $\bigwedge_{i=1}^{n}(Init_i \wedge \Box[\mathcal{N}_i]_{v_i})$ in canonical form. For noninterleaving representations, we apply (7). In either case, the proof of (8) is an implementation proof of the kind discussed in Section 3.1.2.

For the GCD example, the first hypothesis asserts that $\mathcal{C}(M_a) \wedge \mathcal{C}(M_b)$ implies E_a and E_b. This differs from the third hypothesis of (1) in Section 2 because of the $\mathcal{C}$'s. To verify the hypothesis, we can apply Proposition 1 to show that $\mathcal{C}(M_a)$ and $\mathcal{C}(M_b)$ are obtained by simply deleting the fairness conditions from M_a and M_b. Since $\mathcal{N}_b$ implies $(a < b) \wedge (a' = a)$, it is easy to see that $\mathcal{C}(M_b)$ implies E_a. It is equally easy to see that $\mathcal{C}(M_a)$ implies E_b. (In more complicated examples, E_i will not follow from $\mathcal{C}(M_j)$ for any single j.)

To prove part (a) of the second hypothesis, we first eliminate the $+$. For noninterleaving representations, this must be done with Proposition 4. For interleaving representations, we can apply Proposition 5. In either case, we can prove the resulting formula by first using Proposition 2 to eliminate quantifiers, using Proposition 1 to compute closures, and then performing a standard implementation proof with a refinement mapping.

Part (b) of the hypothesis also calls for a standard implementation proof, for which we use the same refinement mapping as in the proof of (a). Since E_i implies $\mathcal{C}(E_i)_{+v}$ and M_i^l implies $\mathcal{C}(M_i^l)$, we can infer from part (a) that $E_i \wedge M_i^l$ implies $\mathcal{C}(M_i)$. Thus proving part (b) requires verifying only the liveness part of M_i.

For the GCD example, we verify the two parts of the second hypothesis by proving $\mathcal{C}(E_a)_{+\langle a, b\rangle} \wedge \mathcal{C}(M_a^l) \Rightarrow \mathcal{C}(M_a)$ and $E_a \wedge M_a^l \Rightarrow M_a$; the proofs of the corresponding conditions for M_b are similar. We first observe that the initial condition of E_a is true, and that, since M_a^l is an interleaving representation, its next-state action $\mathcal{N}_a^l$ implies that no step changes both a and b, so $\mathcal{C}(M_a^l)$ implies $Disjoint(a, b)$. Hence, applying Proposition 5, we reduce our task to proving $\mathcal{C}(E_a) \wedge \mathcal{C}(M_a^l) \Rightarrow \mathcal{C}(M_a)$ and $E_a \wedge M_a^l \Rightarrow M_a$. Applying

Proposition 2 to remove the quantifier from $\mathcal{C}(M_a^l)$ and Proposition 1 to remove the $\mathcal{C}$'s, we reduce proving $\mathcal{C}(E_a) \wedge \mathcal{C}(M_a^l) \Rightarrow \mathcal{C}(M_a)$ to proving

$$E_a \wedge \mathit{Init}_a^l \wedge \Box[\mathcal{N}_a^l]_{\langle a,\, ai,\, pca\rangle} \Rightarrow \mathit{Init}_a \wedge \Box[\mathcal{N}_a]_a \tag{9}$$

Using simple logic and (9), we reduce proving $E_a \wedge M_a^l \;\Rightarrow\; M_a$ to proving

$$E_a \;\wedge\; \mathit{Init}_a^l \wedge \Box[\mathcal{N}_a^l]_{\langle a,\, ai,\, pca\rangle} \wedge \mathrm{WF}_{\langle a,\, ai,\, pca\rangle}(\mathcal{N}_a^l) \;\Rightarrow\; \mathrm{WF}_a(\mathcal{N}_a) \tag{10}$$

We can use Proposition 3 to rewrite the left-hand sides of (9) and (10) in canonical form. The resulting conditions are in the usual form for a TLA implementation proof.

In summary, by applying our propositions in a standard sequence, we can use the Decomposition Theorem to reduce decompositional reasoning to ordinary TLA reasoning. This reduction may seem complicated for so trivial an example as the GCD program, but it will be an insignificant part of the proof for any realistic example.

5.4. The general theorem

We sometimes need to prove the correctness of systems defined inductively. At induction stage N+1, the low- and high-level specifications are defined as the conjunctions of k copies of low- and high-level specifications of stage N, respectively. For example, a 2^{N+1}-bit multiplier is sometimes implemented by combining four 2^N-bit multipliers. We want to prove by induction on N that the stage N low-level specification implements the stage N high-level specification. For such a proof, we need a more general decomposition theorem whose conclusion at stage N can be used in proving the hypotheses at state $N+1$. The appropriate theorem is:

Theorem 2 (General Decomposition 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) $\models \mathcal{C}(E_i)_{+v} \wedge \mathcal{C}(M_i^l) \;\Rightarrow\; \mathcal{C}(M_i)$

   (b) $\models E_i \wedge M_i^l \;\Rightarrow\; M_i$

3. *v is a tuple of variables including all the free variables of M_i.*

then (a) $\models \mathcal{C}(E)_{+v} \wedge \bigwedge_{j=1}^{n} \mathcal{C}(M_j^l) \;\Rightarrow\; \bigwedge_{j=1}^{n} \mathcal{C}(M_j)$, *and*

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

Conclusion (b) of this theorem has the same form as hypothesis 2(b), with M_i^l and M_i replaced with conjunctions. To make the corresponding hypothesis 2(a) follow from conclusion (a), it suffices to prove $\bigwedge_{j=1}^{n} \mathcal{C}(M_j) \Rightarrow \mathcal{C}(\bigwedge_{j=1}^{n} M_j)$, since $\mathcal{C}(\bigwedge_{j=1}^{n} M_j^l) \Rightarrow \bigwedge_{j=1}^{n} \mathcal{C}(M_j^l)$ is always true.

The General Decomposition Theorem has been applied to the verification of an inductively defined multiplier circuit [11].

It can be shown that both versions of our decomposition theorem provide complete rules for verifying that one composition implies another. However, this result is of no significance. Decomposition can simplify a proof only if the proof can be decomposed, in the sense that each M_i^l implements the corresponding M_i under a simple environment assumption E_i. Our theorems are designed to handle those proofs that can be decomposed.

6. COMPARISON WITH RELATED WORK AND CONCLUSIONS

We have developed a method for describing components of concurrent systems as TLA formulas. Although the idea of reducing programming concepts to logic is old, our method is new. Our style of writing specifications is direct and, we believe, practical.

We have also provided rules for proving properties of large systems by reasoning about their components. The Decomposition Theorem is rather simple, yet it allows fairness properties and hiding. The general treatment of fairness and hiding distinguishes our approach from earlier ones for modular reasoning [3,5,9,15,16,18,19]. Moreover, this previous work is mainly concerned with composition of assumption/guarantee specifications, while our rules are crafted to facilitate decomposition of complete systems. An exception is the work of Berthet and Cerny [8], who used decomposition in proving safety properties for finite-state automata.

We have used our Decomposition Theorem with no difficulty on a few toy examples. However, we believe that its biggest payoff will be for systems that are too complex to verify easily by hand. The theorem makes it possible for decision procedures to do most of the work in verifying a system, even when these procedures cannot be applied to the whole system because its state space is very large or unbounded. This approach is currently being pursued in one substantial example: the mechanical verification of a multiplier circuit using a combination of TLA reasoning and mechanical verification with COSPAN [11]. Because it eliminates reasoning about the complete low-level system, the Decomposition Theorem is the key to this division of labor.

REFERENCES

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.
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 [10, 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. To appear as an SRC Research Report, 1993.
5. Martín Abadi and Gordon Plotkin. A logical view of composition and refinement. *Theoretical Computer Science*, 114(1):3–30, June 1993.
6. S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. Technical Report DoC 92/24, Department of Computing, Imperial College of Science, Technology, and Medicine, 1992.
7. Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

8. Christian Berthet and Eduard Cerny. An algebraic model for asynchronous circuits verification. *IEEE Transactions On Computers*, 37(7):835–847, July 1988.
9. 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.
10. 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.
11. R. P. Kurshan and Leslie Lamport. Verification of a multiplier: 64 bits and beyond. In Costas Courcoubetis, editor, *Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 166–179, Berlin, June 1993. Springer-Verlag. Proceedings of the Fifth International Conference, CAV'93.
12. Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, Paris, September 1983. IFIP, North-Holland.
13. Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.
14. 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*.
15. Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, July 1981.
16. Paritosh K. Pandya and Mathai Joseph. P-A logic—a compositional proof system for distributed programs. *Distributed Computing*, 5(1):37–54, 1991.
17. Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–80, 1981.
18. 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.
19. 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.
20. Pamela Zave and Michael Jackson. Conjunction as composition. Submitted for publication, June 1992.