Composition: A Way to Make Proofs Harder

Leslie Lamport

24 December 1997

Appeared in *Compositionality: The Significant Difference (Proceedings of the COMPOS'97 Symposium)*, Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli editors. Lecture Notes in Computer Science, number 1536, (1998), 402–423.

Table of Contents

Composition: A Way to Make Proofs Harder

Leslie Lamport

Systems Research Center, Digital Equipment Corporation

Abstract. Compositional reasoning about a system means writing its specification as the parallel composition of components and reasoning separately about each component. When distracting language issues are removed and the underlying mathematics is revealed, compositional reasoning is seen to be of little use.

1 Introduction

When an engineer designs a bridge, she makes a mathematical model of it and reasons mathematically about her model. She might talk about *calculating* rather than *reasoning*, but calculating $\sqrt{2}$ to three decimal places is just a way of proving $|\sqrt{2} - 1.414| < 10^{-3}$. The engineer reasons compositionally, using laws of mathematics to decompose her calculations into small steps. She would probably be mystified by the concept of compositional reasoning about bridges, finding it hard to imagine any form of reasoning that was not compositional.

Because computer systems can be built with software rather than girders and rivets, many computer scientists believe these systems should not be modeled with the ordinary mathematics used by engineers and scientists, but with something that looks vaguely like a programming language. We call such a language a *pseudo-programming languages* (PPL). Some PPLs, such as CSP, use constructs of ordinary programming languages. Others, like CCS, use more abstract notation. But, they have two defining properties: they are specially designed to model computer systems, and they are not meant to implement useful, real-world programs.

When using a pseudo-programming language, compositional reasoning means writing a model as the composition of smaller pseudo-programs, and reasoning separately about those smaller pseudo-programs. If one believes in using PPLs to model computer systems, then it is natural to believe that decomposition should be done in terms of the PPL, so compositionality must be a Good Thing.

We adopt the radical approach of modeling computer systems the way engineers model bridges—using mathematics. Compositionality is then a trivial consequence of the compositionality of ordinary mathematics. We will see that the compositional approaches based on pseudo-programming languages are analogous to performing calculations about a bridge design by decomposing it into smaller bridge designs. While this technique may occasionally be useful, it is hardly a good general approach to bridge design.

2 The Mathematical Laws of Composition

Mathematical reasoning is embodied in statements (also called theorems) and their proofs. The reasoning is hierarchical—the proof of a statement consists of a sequence of statements, each with its proof. The decomposition stops at a level at which the proof is sufficiently obvious that it can be written as a short, simple paragraph. How rigorous the proof is depends on what "obvious" means. In the most rigorous proofs, it means simple enough so that even a computer can verify it. Less rigorous proofs assume a reader of greater intelligence (or greater faith). We will use the notation introduced in [10] to write hierarchical proofs.

Two fundamental laws of mathematics are used to decompose proofs:

$$\wedge\text{-Composition} \quad \frac{\begin{array}{c}A \Rightarrow B\\ A \Rightarrow C\end{array}}{A \Rightarrow B \wedge C} \qquad \vee\text{-Composition} \quad \frac{\begin{array}{c}A \Rightarrow C\\ B \Rightarrow C\end{array}}{A \vee B \Rightarrow C}$$

Logicians have other names for these laws, but our subject is compositionality, so we adopt these names. A special case of $\vee$-composition is:

$$\text{Case-Analysis} \quad \frac{\begin{array}{c}A \wedge B \Rightarrow C\\ A \wedge \neg B \Rightarrow C\end{array}}{A \Rightarrow C}$$

The propositional $\wedge$- and $\vee$-composition rules have the following predicate-logic generalizations:

$$\forall\text{-Composition} \quad \frac{(i \in S) \wedge P \Rightarrow Q(i)}{P \Rightarrow (\forall\, i \in S : Q(i))}$$

$$\exists\text{-Composition} \quad \frac{(i \in S) \wedge P(i) \Rightarrow Q}{(\exists\, i \in S : P(i)) \Rightarrow Q}$$

Another rule that is often used (under a very different name) is

$$\text{Act-Stupid} \quad \frac{A \Rightarrow C}{A \wedge B \Rightarrow C}$$

We call it the act-stupid rule because it proves that $A \wedge B$ implies C by ignoring the hypothesis B. This rule is useful when B can't help in the proof, so we need only the hypothesis A. Applying it in a general method, when we don't know what A and B are, is usually a bad idea.

3 Describing a System with Mathematics

We now explain how to use mathematics to describe systems. We take as our example a digital clock that displays the hour and minute. For simplicity, we ignore the fact that a clock is supposed to tell the real time, and we instead just specify the sequence of times that it displays. A more formal explanation of the approach can be found in [9].

3.1 Discrete Dynamic Systems

Our clock is a dynamic system, meaning that it evolves over time. The classic way to model a dynamic system is by describing its state as a continuous function of time. Such a function would describe the continuum of states the display passes through when changing from 12:49 to 12:50. However, we view the clock as a discrete system. Discrete systems are, by definition, ones we consider to exhibit discrete state changes. Viewing the clock as a discrete system means ignoring the continuum of real states and pretending that it changes from 12:49 to 12:50 without passing through any intermediate state. We model the execution of a discrete system as a sequence of states. We call such a sequence a *behavior*. To describe a system, we describe all the behaviors that it can exhibit.

3.2 An Hour Clock

A First Attempt To illustrate how systems are described mathematically, we start with an even simpler example than the hour-minute clock—namely, a clock that displays only the hour. We describe its state by the value of the variable hr. A typical behavior of this system is

$$[hr = 11] \;\rightarrow\; [hr = 12] \;\rightarrow\; [hr = 1] \;\rightarrow\; [hr = 2] \;\rightarrow\; \cdots$$

We describe all possible behaviors by an *initial predicate* that specifies the possible initial values of hr, and a *next-state relation* that specifies how the value of hr can change in any step (pair of successive states).

The initial predicate is just $hr \in \{1, \ldots, 12\}$. The next-state relation is the following formula, in which hr denotes the old value and hr' denotes the new value.

$$((hr = 12) \land (hr' = 1)) \lor ((hr \neq 12) \land (hr' = hr + 1))$$

This kind of formula is easier to read when written with lists of conjuncts or disjuncts, using indentation to eliminate parentheses:

$$\begin{array}{l} \lor \land hr = 12 \\ \land hr' = 1 \\ \lor \land hr \neq 12 \\ \land hr' = hr + 1 \end{array}$$

There are many ways to write the same formula. Borrowing some notation from programming languages, we can write this next-state relation as

$$hr' \;=\; \text{IF } hr = 12 \text{ THEN } 1 \text{ ELSE } hr + 1$$

This kind of formula, a Boolean-valued expression containing primed and unprimed variables, is called an *action*.

Our model is easier to manipulate mathematically if it is written as a single formula. We can write it as

$$\begin{array}{l} \land\; hr \in \{1, \ldots, 12\} \\ \land\; \Box\,(hr' = \text{IF } hr = 12 \text{ THEN } 1 \text{ ELSE } hr + 1) \end{array} \tag{1}$$

This is a temporal formula, meaning that it is true or false of a behavior. A state predicate like $hr \in \{1, \ldots, 12\}$ is true for a behavior iff it is true in the first state. A formula of the form $\Box N$ asserts that the action N holds on all steps of the behavior.

By introducing the operator $\Box$, we have left the realm of everyday mathematics and entered the world of temporal logic. Temporal logic is more complicated than ordinary mathematics. Having a single formula as our mathematical description is worth the extra complication. However, we should use temporal reasoning as little as possible. In any event, temporal logic formulas are still much easier to reason about than programs in a pseudo-programming language.

Stuttering Before adopting (1) as our mathematical description of the hour clock, we ask the question, what is a state? For a simple clock, the obvious answer is that a state is an assignment of values to the variable hr. What about a railroad station with a clock? To model a railroad station, we would use a number of additional variables, perhaps including a variable sig to record the state of a particular signal in the station. One possible behavior of the system might be

$$\begin{bmatrix} hr \ = 11 \\ sig = \text{“red”} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} hr \ = 12 \\ sig = \text{“red”} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} hr \ = 12 \\ sig = \text{“green”} \\ \vdots \end{bmatrix} \rightarrow$$
$$\begin{bmatrix} hr \ = 12 \\ sig = \text{“red”} \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} hr \ = 1 \\ sig = \text{“red”} \\ \vdots \end{bmatrix} \rightarrow \cdots$$

We would expect our description of a clock to describe the clock in the railroad station. However, formula (1) doesn't do this. It asserts that hr is incremented in every step, but the behavior of the railroad station with clock includes steps like the second and third, which change sig but leave hr unchanged.

To write a single description that applies to any clock, we let a state consist of an assignment of values to all possible variables. In mathematics, the equation $x + y = 1$, doesn't assert that there is no z. It simply says nothing about the value of z. In other words, the formula $x + y = 1$ is not an assertion about some universe containing only x and y. It is an assertion about a universe containing x, y, and all other variables; it constrains the values of only the variables x and y.

Similarly, a mathematical formula that describes a clock should be an assertion not about the variable hr, but about the entire universe of possible variables. It should constrain the value only of hr and should allow arbitrary changes to the other variables—including changes that occur while the value of hr stays the same. We obtain such a formula by modifying (1) to allow "stuttering" steps that leave hr unchanged, obtaining:

$$\begin{aligned} &\wedge\ hr \in \{1, \ldots, 12\} \\ &\wedge\ \Box \begin{pmatrix} \vee\ hr' = \textsc{if}\ hr = 12\ \textsc{then}\ 1\ \textsc{else}\ hr + 1 \\ \vee\ hr' = hr \end{pmatrix} \end{aligned} \tag{2}$$

Clearly, every next-state relation we write is going to have a disjunct that leaves variables unchanged. So, it's convenient to introduce the notation that $[A]_v$ equals $A \vee (v' = v)$, where v' is obtained from the expression v by priming all its free variables. We can then write (2) more compactly as

$$\begin{aligned} &\wedge\ hr \in \{1, \ldots, 12\} \\ &\wedge\ \Box[hr' = \textsc{if}\ hr = 12\ \textsc{then}\ 1\ \textsc{else}\ hr + 1]_{hr} \end{aligned} \tag{3}$$

This formula allows behaviors that stutter forever, such as

$$[hr = 11] \ \rightarrow\ [hr = 12] \ \rightarrow\ [hr = 12] \ \rightarrow\ [hr = 12] \ \rightarrow\ \cdots$$

Such a behavior describes a stopped clock. It illustrates that we can assume all behaviors are infinite, because systems that halt are described by behaviors that end with infinite stuttering. But, we usually want our clocks not to stop.

Fairness To describe a clock that doesn't stop, we must add a conjunct to (3) to rule out infinite stuttering. Experience has shown that the best way to write this conjunct is with *fairness* formulas. There are two types of fairness, weak and strong, expressed with the WF and SF operators that are defined as follows.

$\mathrm{WF}_v(A)$ If $A \wedge (v' \neq v)$ is enabled forever, then infinitely many $A \wedge (v' \neq v)$ steps must occur.

$\mathrm{SF}_v(A)$ If $A \wedge (v' \neq v)$ is enabled infinitely often, then infinitely many $A \wedge (v' \neq v)$ steps must occur.

The $v' \neq v$ conjuncts make it impossible to use WF or SF to write a formula that rules out finite stuttering.

We can now write our description of the hour clock as the formula Π, defined by

$$\begin{aligned} N &\triangleq hr' = \textsc{if}\ hr = 12\ \textsc{then}\ 1\ \textsc{else}\ hr + 1 \\ \Pi &\triangleq (hr \in \{1, \ldots, 12\}) \wedge \Box[N]_{hr} \wedge \mathrm{WF}_{hr}(N) \end{aligned}$$

The first two conjuncts of Π (which equal (3)), express a *safety* property. Intuitively, a safety property is characterized by any of the following equivalent conditions.

– It asserts that the system never does something bad.

– It asserts that the system starts in a good state and never takes a wrong step.

– It is finitely refutable—if it is violated, then it is violated at some particular point in the behavior.

The last conjunct of Π (the WF formula) is an example of a *liveness* property. Intuitively, a liveness property is characterized by any of the following equivalent conditions.

– It asserts that the system eventually does something good.

– It asserts that the system eventually takes a good step.

– It is not finitely refutable—it is possible to satisfy it after any finite portion of the behavior.

Formal definitions of safety and liveness are due to Alpern and Schneider [4].

Safety properties are proved using only ordinary mathematics (plus a couple of lines of temporal reasoning). Liveness properties are proved by combining temporal logic with ordinary mathematics. Here, we will mostly ignore liveness and concentrate on safety properties.

3.3 An Hour-Minute Clock

The Internal Specification It is now straightforward to describe a clock with an hour and minute display. The two displays are represented by the values of the variables *hr* and *min*. To make the specification more interesting, we describe a clock in which the two displays don't change simultaneously when the hour changes. When the display changes from 8:59 to 9:00, it transiently reads 8:00 or 9:59. Since we are ignoring the actual times at which state changes occur, these transient states are no different from the states when the clock displays the "correct" time.

Figure 1 defines a formula Φ that describes the hour-minute clock. It uses an additional variable *chg* that equals TRUE when the display is in a transient state. Action M_m describes the changing of *min*; action M_h describes the changing of hr. The testing and setting of *chg* by these actions is a bit tricky, but a little thought reveals what's going on. Action M_h introduces a gratuitous bit of cleverness to remove the **if/then** construct from the specification of the new value of hr. The next-state relation for the hour-minute clock is $M_m \lor M_h$, because a step of the clock increments either *min* or hr. Since $\langle hr, \mathit{min}\ \mathit{chg} \rangle'$ equals $\langle hr', \mathit{min}', \mathit{chg}' \rangle$, it equals $\langle hr, \mathit{min}, \mathit{chg} \rangle$ iff hr, *min*, and, *chg* are all unchanged.

Existential Quantification Formula Φ of Figure 1 contains the free variables hr, *min*, and *chg*. However, the description of a clock should mention only *hr* and *min*, not *chg*. We need to "hide" *chg*. In mathematics, hiding means existential quantification. The formula $\exists\, x : y = x^2$ asserts that there is some value of x that makes $y = x^2$ true; it says nothing about the actual value of x. The formula describing an hour-minute clock is $\boldsymbol{\exists}\, \mathit{chg} : \Phi$. The quantifier $\boldsymbol{\exists}$ is a temporal operator, asserting that there is a *sequence* of values of *chg* that makes Φ true. The precise definition of $\boldsymbol{\exists}$ is a bit subtle and can be found in [9].

$$\begin{aligned}
Init_{\Phi} \;&\triangleq\; \land\ hr \in \{1, \ldots, 12\} \\
&\;\; \land\ min \in \{0, \ldots, 59\} \\
&\;\; \land\ chg = \text{FALSE} \\
M_m \;&\triangleq\; \land\ \neg((min = 0) \land chg) \\
&\;\; \land\ min' = (min + 1) \bmod 60 \\
&\;\; \land\ chg' = (min = 59) \land \neg chg \\
&\;\; \land\ hr' = hr \\
M_h \;&\triangleq\; \land\ \lor\ (min = 59) \land \neg chg \\
&\;\; \ \lor\ (min = 0) \land chg \\
&\;\; \land\ hr' = (hr \bmod 12) + 1 \\
&\;\; \land\ chg' = \neg chg \\
&\;\; \land\ min' = min \\
\Phi \;&\triangleq\; \land\ Init_{\Phi} \\
&\;\; \land\ \Box[M_m \lor M_h]_{\langle hr, min, chg \rangle} \\
&\;\; \land\ \mathrm{WF}_{\langle hr, min, chg \rangle}(M_m \lor M_h)
\end{aligned}$$

Fig. 1. The internal specification of an hour-minute clock.

3.4 Implementation and Implication

An hour-minute clock implements an hour clock. (If we ask someone to build a device that displays the hour, we can't complain if the device also displays the minute.) Every behavior that satisfies the description of an hour-minute clock also satisfies the description of an hour clock. Formally, this means that the formula $(\exists\, chg : \Phi) \Rightarrow \Pi$ is true. In mathematics, if something is true, we should be able to prove it. The rules of mathematics allow us to decompose the proof hierarchically. Here is the statement of the theorem, and the first two levels of its proof. (See [10] for an explanation of the proof style.)

Theorem 1. $(\exists\, chg : \Phi) \Rightarrow \Pi$

$\langle 1 \rangle 1$. $\Phi \Rightarrow \Pi$
  $\langle 2 \rangle 1$. $Init_{\Phi} \Rightarrow hr \in \{1, \ldots, 12\}$
  $\langle 2 \rangle 2$. $\Box[M_m \lor M_h]_{\langle hr, min, chg \rangle} \Rightarrow \Box[N]_{hr}$
  $\langle 2 \rangle 3$. $\Phi \Rightarrow \mathrm{WF}_{hr}(N)$
  $\langle 2 \rangle 4$. Q.E.D.
    PROOF: By $\langle 2 \rangle 1$–$\langle 2 \rangle 3$ and the $\land$-composition and act-stupid rules.
$\langle 1 \rangle 2$. Q.E.D.
  PROOF: By $\langle 1 \rangle 1$, the definition of Φ, and predicate logic[1], since chg does not occur free in Π.

[1] We are actually reasoning about the temporal operator $\exists$ rather than ordinary existential quantification, but it obeys the usual rules of predicate logic.

Let's now go deeper into the hierarchical proof. The proof of $\langle 2\rangle 1$ is trivial, since $Init_{\Phi}$ contains the conjunct $hr \in \{1, \ldots, 12\}$. Proving liveness requires more temporal logic than we want to delve into here, so we will not show the proof of $\langle 2\rangle 3$ or of any other liveness properties. We expand the proof of $\langle 2\rangle 2$ two more levels as follows.

$\langle 2\rangle 2.$ $\Box[M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow \Box[N]_{hr}$

$\langle 3\rangle 1.$ $[M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow [N]_{hr}$

$\langle 4\rangle 1.$ $M_m \Rightarrow [N]_{hr}$

$\langle 4\rangle 2.$ $M_h \Rightarrow [N]_{hr}$

$\langle 4\rangle 3.$ $(\langle hr, min, chg\rangle' = \langle hr, min, chg\rangle) \Rightarrow [N]_{hr}$

$\langle 4\rangle 4.$ Q.E.D.

PROOF: By $\langle 4\rangle 1$–$\langle 4\rangle 3$ and the $\vee$-composition rule.

$\langle 3\rangle 2.$ Q.E.D.

PROOF: By $\langle 3\rangle 1$ and the rule $\dfrac{A \Rightarrow B}{\Box A \Rightarrow \Box B}$.

The proof of $\langle 4\rangle 1$ is easy, since M_m implies $hr' = hr$. The proof of $\langle 4\rangle 3$ is equally easy. The proof of $\langle 4\rangle 2$ looks easy enough.

$\langle 4\rangle 2.$ $M_h \Rightarrow [N]_{hr}$

PROOF:
$$\begin{array}{lll} M_h & \Rightarrow & hr' = (hr \bmod 12) + 1 \\ & \Rightarrow & hr' = \text{IF } hr = 12 \text{ THEN } 1 \text{ ELSE } hr + 1 \\ & \triangleq & N \end{array}$$

However, this proof is wrong! The second implication is not valid. For example, if hr equals 25, then the first equation asserts $hr' = 2$, while the second asserts $hr' = 26$. The implication is valid only under the additional assumption $hr \in \{1, \ldots, 12\}$.

Define Inv to equal the predicate $hr \in \{1, \ldots, 12\}$. We must show that Inv is true throughout the execution, and use that fact in the proof of step $\langle 4\rangle 2$. Here are the top levels of the corrected proof.

$\langle 1\rangle 1.$ $\Phi \Rightarrow \Pi$

$\langle 2\rangle 1.$ $Init_{\Phi} \Rightarrow hr \in \{1, \ldots, 12\}$

$\langle 2\rangle 2.$ $Init_{\Phi} \wedge \Box[M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow \Box Inv$

$\langle 2\rangle 3.$ $\Box Inv \wedge \Box[M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow \Box[N]_{hr}$

$\langle 2\rangle 4.$ $\Box Inv \wedge \Phi \Rightarrow \mathrm{WF}_{hr}(N)$

$\langle 2\rangle 5.$ Q.E.D.

PROOF: By $\langle 2\rangle 1$–$\langle 2\rangle 4$, and the $\wedge$-composition and act-stupid rules.

$\langle 1\rangle 2.$ Q.E.D.

PROOF: By $\langle 1\rangle 1$, the definition of Φ, and predicate logic, since chg does not occur free in Π.

The high-level proofs of $\langle 2\rangle 2$ and $\langle 2\rangle 3$ are

$\langle 2\rangle 2.$ $Init_{\Phi} \wedge \Box[M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow \Box Inv$

$\langle 3\rangle 1.$ $Init_{\Phi} \Rightarrow Inv$

$\langle 3\rangle 2.$ $Inv \wedge [M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow Inv'$

$\langle 3\rangle 3.$ Q.E.D.

PROOF: By $\langle 3\rangle 1$, $\langle 3\rangle 2$ and the rule $\dfrac{P \wedge [A]_v \Rightarrow P'}{P \wedge \Box[A]_v \Rightarrow \Box P}$.

$\langle 2\rangle 3.$ $\Box Inv \wedge \Box[M_m \vee M_h]_{\langle hr, min, chg\rangle} \Rightarrow \Box[N]_{hr}$

$\langle 3 \rangle 1.$ $Inv \wedge [M_m \vee M_h]_{\langle hr, min, chg \rangle} \Rightarrow [N]_{hr}$
$\langle 3 \rangle 2.$ Q.E.D.
PROOF: By $\langle 3 \rangle 1$ and the rules $\dfrac{A \Rightarrow B}{\Box A \Rightarrow \Box B}$ and $\Box(A \wedge B) \equiv \Box A \wedge \Box B$.

The further expansion of the proofs is straightforward and is left as an exercise for the diligent reader.

3.5 Invariance and Step Simulation

The part of the proof shown above is completely standard. It contains all the temporal-logic reasoning used in proving safety properties. The formula *Inv* satisfying $\langle 2 \rangle 2$ is called an *invariant*. Substep $\langle 3 \rangle 2$ of step $\langle 2 \rangle 3$ is called proving *step simulation*. The invariant is crucial in this step and in step $\langle 2 \rangle 4$ (the proof of liveness). In general, the hard parts of the proof are discovering the invariant, substep $\langle 3 \rangle 2$ of step $\langle 2 \rangle 2$ (the crucial step in the proof of invariance), step simulation, and liveness.

In our example, *Inv* asserts that the value of *hr* always lies in the correct set. Computer scientists call this assertion *type correctness*, and call the set of correct values the *type* of *hr*. Hence, *Inv* is called a type-correctness invariant. This is the simplest form of invariant. Computer scientists usually add a type system just to handle this particular kind of invariant, since they tend to prefer formalisms that are more complicated and less powerful than simple mathematics.

Most invariants express more interesting properties than just type correctness. The invariant captures the essence of what makes an implementation correct. Finding the right invariant, and proving its invariance, suffices to prove the desired safety properties of many concurrent algorithms. This is the basis of the first practical method for reasoning about concurrent algorithms, which is due to Ashcroft [5].

3.6 A Formula by any Other Name

We have been calling formulas like Φ and Π "descriptions" or "models" of a system. It is customary to call them *specifications*. This term is sometimes reserved for high-level description of systems, with low-level descriptions being called *implementations*. We make no distinction between specifications and implementations. They are all descriptions of a system at various levels of detail. We use the terms algorithm, description, model, and specification as different names for the same thing: a mathematical formula.

4 Invariance in a Pseudo-Programming Language

Invariance is a simple concept. We now show how a popular method for proving invariance in terms of a pseudo-programming language is a straightforward consequence of the rules of mathematics.

4.1 The Owicki-Gries Method

In the Owicki-Gries method [8, 11], the invariant is written as a program annotation. For simplicity, let's assume a multiprocess program in which each process i in a set $\mathbf{P}$ of processes repeatedly executes a sequence of atomic instructions $S_0^{(i)}$, $\ldots$, $S_{n-1}^{(i)}$. The invariant is written as an annotation, in which each statement $S_j^{(i)}$ is preceded by an assertion $A_j^{(i)}$, as shown in Figure 2.

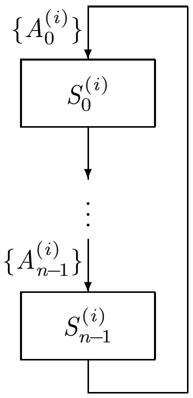

Fig. 2. An Owicki-Gries style annotation of a process.

To make sense of this picture, we must translate it into mathematics. We first rewrite each operation $S_j^{(i)}$ as an action, which we also call $S_j^{(i)}$. This rewriting is easy. For example, an assignment statement $x := x + 1$ is written as the action $(x' = x + 1) \wedge (\langle \ldots \rangle' = \langle \ldots \rangle)$, where "…" is the list of other variables. We represent the program's control state with a variable pc, where $pc[i] = j$ means that control in process i is immediately before statement $S_j^{(i)}$. The program and its invariant are then described by the formulas Π and Inv of Figure 3.

We can derive the Owicki-Gries rules for proving invariance by applying the proof rules we used before. The top-level proof is:

Theorem 2. (Owicki-Gries) $\Pi \Rightarrow \Box I$

$\langle 1 \rangle 1.$ $Init \Rightarrow Inv$
$\langle 1 \rangle 2.$ $Inv \wedge [N]_{\langle vbl,\, pc \rangle} \Rightarrow Inv'$
  $\langle 2 \rangle 1.$ $Inv \wedge N \Rightarrow Inv'$
  $\langle 2 \rangle 2.$ $Inv \wedge (\langle vbl,\, pc \rangle' = \langle vbl,\, pc \rangle) \Rightarrow Inv'$
  $\langle 2 \rangle 3.$ Q.E.D.
    PROOF: By $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, and the $\vee$-composition rule.
$\langle 1 \rangle 3.$ Q.E.D.
  PROOF: By $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, and the rule $\dfrac{P \wedge [A]_v \Rightarrow P'}{P \wedge \Box[A]_v \Rightarrow \Box P}$.

$$
\begin{array}{lll}
\mathit{Init} & \triangleq & \land\ \forall\, i \in \mathbf{P} \,:\, pc[i] = 0 \\
 & & \land\ \ldots \quad \text{[The initial conditions on program variables.]} \\
Go_j^{(i)} & \triangleq & \land\ pc[i] = j \\
 & & \land\ pc[i]' = (j+1) \bmod n \\
 & & \land\ \forall\, k \in \mathbf{P} \,:\, (k \neq i) \Rightarrow (pc[k]' = pc[k]) \\
N & \triangleq & \exists\, i \in \mathbf{P},\ j \in \{0, \ldots, n-1\} \,:\, Go_j^{(i)} \land S_j^{(i)} \\
\mathit{vbl} & \triangleq & \langle \ldots \rangle \quad \text{[The tuple of all program variables.]} \\
\mathit{\Pi} & \triangleq & \mathit{Init} \land \Box[N]_{\langle \mathit{vbl},\, pc \rangle} \\
\mathit{Inv} & \triangleq & \forall\, i \in \mathbf{P},\ j \in \{0, \ldots, n-1\} \,:\, (pc[i] = j) \Rightarrow A_j^{(i)}
\end{array}
$$

Fig. 3. The formulas describing the program and annotation of Figure 2.

The hard part is the proof of $\langle 2 \rangle 1$. We first decompose it using the $\forall$- and $\exists$-composition rules.

$\langle 2 \rangle 1.$ $\mathit{Inv} \land N \Rightarrow \mathit{Inv}'$

$\langle 3 \rangle 1.$ $\begin{pmatrix} \land\ i \in \mathbf{P} \\ \land\ j \in \{0, \ldots, n-1\} \\ \land\ \mathit{Inv} \land Go_j^{(i)} \land S_j^{(i)} \end{pmatrix} \Rightarrow \mathit{Inv}'$

$\langle 4 \rangle 1.$ $\begin{pmatrix} \land\ i \in \mathbf{P} \\ \land\ j \in \{0, \ldots, n-1\} \\ \land\ k \in \mathbf{P} \\ \land\ l \in \{0, \ldots, n-1\} \\ \land\ \mathit{Inv} \land Go_j^{(i)} \land S_j^{(i)} \end{pmatrix} \Rightarrow ((pc[k]' = l) \Rightarrow (A_l^{(k)})')$

$\langle 4 \rangle 2.$ Q.E.D.

PROOF: By $\langle 4 \rangle 1$, the definition of *Inv*, and the $\forall$-composition rule.

$\langle 3 \rangle 2.$ Q.E.D.

PROOF: By $\langle 3 \rangle 1$, the definition of N, and the $\exists$-composition rule.

We prove $\langle 4 \rangle 1$ by cases, after first using propositional logic to simplify its statement. We let $j \oplus 1$ equal $(j+1) \bmod n$.

$\langle 4 \rangle 1.$ $\begin{pmatrix} \land\ i, k \in \mathbf{P} \\ \land\ j, l \in \{0, \ldots, n-1\} \\ \land\ pc[k]' = l \\ \land\ \mathit{Inv} \land Go_j^{(i)} \land S_j^{(i)} \end{pmatrix} \Rightarrow (A_l^{(k)})'$

$\langle 5 \rangle 1.$ CASE: $i = k$

$\langle 6 \rangle 1.$ $\begin{pmatrix} \land\ i \in \mathbf{P} \\ \land\ j \in \{0, \ldots, n-1\} \\ \land\ A_j^{(i)} \land S_j^{(i)} \end{pmatrix} \Rightarrow (A_{j \oplus 1}^{(i)})'$

$\langle 6 \rangle 2.$ Q.E.D.

PROOF: By $\langle 6 \rangle 1$, the level-$\langle 5 \rangle$ assumption, the definition of *Inv*, and the act-stupid rule, since $(pc[i]' = l) \wedge Go_j^{(i)}$ implies $(l = j \oplus 1)$.

⟨5⟩2. Case: $i \neq k$

⟨6⟩1. $\begin{pmatrix} \wedge\ i, k \in \mathbf{P} \\ \wedge\ j, l \in \{0, \ldots, n-1\} \\ \wedge\ A_j^{(i)} \wedge A_l^{(k)} \wedge S_j^{(i)} \end{pmatrix} \;\Rightarrow\; (A_l^{(k)})'$

⟨6⟩2. Q.E.D.

Proof: By ⟨6⟩1, the level-⟨5⟩ assumption, the definition of *Inv*, and the act-stupid rule, since $(pc[k]' = l) \wedge Go_j^{(i)}$ implies $(pc[k] = l)$, for $k \neq i$, and $(pc[k] = l) \wedge Inv$ implies $A_l^{(k)}$.

We are finally left with the two subgoals numbered ⟨6⟩1. Summarizing, we see that to prove $Init \Rightarrow \Box Inv$, it suffices to prove the two conditions

$$A_j^{(i)} \wedge S_j^{(i)} \;\Rightarrow\; (A_{j \oplus 1}^{(i)})'$$
$$A_j^{(i)} \wedge A_l^{(k)} \wedge S_j^{(i)} \;\Rightarrow\; (A_l^{(k)})'$$

for all i, k in $\mathbf{P}$ with $i \neq k$, and all j, l in $\{0, \ldots, n-1\}$. These conditions are called *Sequential Correctness* and *Interference Freedom*, respectively.

4.2 Why Bother?

We now consider just what have has been accomplished by describing by proving invariance in terms of a pseudo-programming language instead of directly in mathematics.

Computer scientists are quick to point out that using "$:=$" instead of "$=$" avoids the need to state explicitly what variables are left unchanged. In practice, this reduces the length of a specification by anywhere from about 10% (for a very simple algorithm) to 4% (for a more complicated system). For this minor gain, it introduces the vexing problem of figuring out exactly what variables can and cannot be changed by executing $x := x + 1$. The obvious requirement that no other variable is changed would not allow us to implement x as the sum $lh * 2^{32} + rh$ of two 32-bit values, since it forbids *lh* and *rh* to change when x is incremented. The difficulty of deciding what can and cannot be changed by an assignment statement is one of the things that makes the semantics of programming languages (both real and pseudo) complicated. By using mathematics, we avoid this problem completely.

A major achievement of the Owicki-Gries method is eliminating the explicit mention of the variable *pc*. By writing the invariant as an annotation, one can write $A_j^{(i)}$ instead of $(pc[i] = j) \Rightarrow A_j^{(i)}$. At the time, computer scientists seemed to think that mentioning *pc* was a sin. However, when reasoning about a concurrent algorithm, we must refer to the control state in the invariant. Owicki and Gries therefore had to introduce dummy variables to serve as euphemisms for *pc*. When using mathematics, any valid formula of the form $Init \wedge \Box[N]_v \Rightarrow \Box P$, for a state predicate P, can be proved without adding dummy variables.

One major drawback of the Owicki-Gries method arises from the use of the act-stupid rule in the proofs of the two steps numbered ⟨6⟩2. The rule was applied without regard for whether the hypotheses being ignored are useful. This means that there are annotations for which step $\langle 2 \rangle 1$ (which asserts $N \wedge Inv \Rightarrow Inv'$) is valid but cannot be proved with the Owicki-Gries method. Such invariants must be rewritten as different, more complicated annotations.

Perhaps the thing about the Owicki-Gries method is that it obscures the underlying concept of invariance. We refer the reader to [6] for an example of how complicated this simple concept becomes when expressed in terms of a pseudo-programming language. In 1976, the Owicki-Gries method seemed like a major advance over Ashcroft's simple notion of invariance. We have since learned better.

5 Refinement

5.1 Refinement in General

We showed above that an hour-minute clock implements an hour clock by proving $(\exists\, chg : \Phi) \Rightarrow \Pi$. That proof does not illustrate the general case of proving that one specification implements another because the higher-level specification Π has no internal (bound) variable. The general case is covered by the following proof outline, where x, y, and z denote arbitrary tuples of variables, and the internal variables y and z of the two specifications are distinct from the free variables x. The proof involves finding a function f, which is called a *refinement mapping* [1].

Theorem 3. (Refinement) $(\exists\, y : \Phi(x, y)) \;\Rightarrow\; (\exists\, z : \Pi(x, z))$

LET: $\overline{z} \;\triangleq\; f(x, y)$
$\langle 1 \rangle 1.$ $\Phi(x, y) \;\Rightarrow\; \Pi(x, \overline{z})$
$\langle 1 \rangle 2.$ $\Phi(x, y) \;\Rightarrow\; (\exists\, z : \Pi(x, z))$
PROOF: By $\langle 1 \rangle 1$ and predicate logic, since the variables of z are distinct from those of x.

The proof of step $\langle 1 \rangle 1$ has the same structure as in our clock example.

5.2 Hierarchical Refinement

In mathematics, it is common to prove a theorem of the form $P \Rightarrow Q$ by introducing a new formula R and proving $P \Rightarrow R$ and $R \Rightarrow Q$. We can prove that a lower-level specification $\exists\, y : \Phi(x, y)$ implies a higher-level specification $\exists\, z : \Pi(x, z)$ by introducing an intermediate-level specification $\exists\, w : \Psi(x, w)$ and using the following proof outline.

LET: $\Psi(x, w) \;\triangleq\; \ldots$
$\langle 1 \rangle 1.$ $(\exists\, y : \Phi(x, y)) \;\Rightarrow\; (\exists\, w : \Psi(x, w))$
LET: $\overline{w} \;\triangleq\; g(x, y)$
$\ldots$
$\langle 1 \rangle 2.$ $(\exists\, w : \Psi(x, w)) \;\Rightarrow\; (\exists\, z : \Pi(x, z))$

Let: $\overline{z} \;\triangleq\; h(x, w)$
...

$\langle 1 \rangle 3.$ Q.E.D.
Proof: By $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$.

This proof method is called *hierarchical decomposition*. It's a good way to explain a proof. By using a sequence multiple intermediate specifications, each differing from the next in only one aspect, we can decompose the proof into conceptually simple steps.

Although it is a useful pedagogical tool, hierarchical decomposition does not simplify the total proof. In fact, it usually adds extra work. Hierarchical decomposition adds the task of writing the extra intermediate-level specification. It also restricts how the proof is decomposed. The single refinement mapping f in the outline of the direct proof can be defined in terms of the two mappings g and h of the hierarchical proof by $f(x, y) \triangleq h(x, g(x, y))$. The steps of a hierarchical proof can then be reshuffled to form a particular way of decomposing the lower levels of the direct proof. However, there could be better ways to decompose those levels.

5.3 Interface Refinement

We have said that implementation is implication. For this to be true, the two specifications must have the same free variables. If the high-level specification describes the sending of messages on a network whose state is represented by the variable *net*, then the low-level specification must also describe the sending of messages on *net*.

We often implement a specification $\Sigma(net)$ of sending messages on *net* by a specification $\Lambda(tran)$ of sending packets on a "transport layer" whose state is represented by a variable *tran*. A single message could be broken into multiple packets. Correctness of the implementation cannot mean validity of $\Lambda(tran) \Rightarrow \Sigma(net)$, since $\Lambda(tran)$ and $\Sigma(net)$ have different free variables.

To define what it means for $\Lambda(tran)$ to implement $\Sigma(net)$, we must first define what it means for sending a set of packets to represent the sending of a message. This definition is written as a temporal formula $R(net, trans)$, which is true of a behavior iff the sequence of values of *trans* represents the sending of packets that correspond to the sending of messages represented by the sequence of values of *net*. We call R an *interface refinement*. For R to be a sensible interface refinement, the formula $\Lambda(trans) \Rightarrow \boldsymbol{\exists}\, net : R(net, trans)$ must be valid, meaning that every set of packet transmissions allowed by $\Lambda(trans)$ represents some set of message transmissions. We say that $\Lambda(tran)$ implements $\Sigma(net)$ under the interface refinement $R(net, trans)$ iff $\Lambda(tran) \wedge R(net, trans)$ implies $\Sigma(net)$.

6 Decomposing Specifications

Pseudo-programming languages usually have some parallel composition operator $\|$, where $S_1 \| S_2$ is the parallel composition of specifications S_1 and S_2. We observed in our hour-clock example that a mathematical specification S_1 does not describe only a particular system; rather, it describes a universe containing (the variables that represent) the system. Composing two systems means ensuring that the universe satisfies both of their specifications. Hence, when the specifications S_1 and S_2 are mathematical formulas, their composition is just $S_1 \wedge S_2$.

6.1 Decomposing a Clock into its Hour and Minute Displays

We illustrate how composition becomes conjunction by specifying the hour-minute clock as the conjunction of the specifications of an hour process and a minute process. It is simpler to do this if each variable is modified by only one process. So, we rewrite the specification of the hour-minute clock by replacing the variable *chg* with the expression $chg_h \neq chg_m$, where chg_h and chg_m are two new variables, chg_h being modified by the hour process and chg_m by the minute process. The new specification is $\exists\, chg_h, chg_m : \Psi$, where Ψ is defined in Figure 4. Proving that this specification is equivalent to $\exists\, chg : \Phi$,

$$
\begin{aligned}
Init_\Psi \;&\triangleq\; \wedge\; hr \in \{1, \ldots, 12\}\\
&\;\; \wedge\; min \in \{0, \ldots, 59\}\\
&\;\; \wedge\; chg_m = chg_h = \text{TRUE}\\
N_m \;&\triangleq\; \wedge\; \neg((min = 0) \wedge (chg_m \neq chg_h))\\
&\;\; \wedge\; min' = (min + 1) \bmod 60\\
&\;\; \wedge\; chg'_m = \text{IF } min = 59 \text{ THEN } \neg chg_m \text{ ELSE } chg_h\\
&\;\; \wedge\; \langle hr, chg_h \rangle' = \langle hr, chg_h \rangle\\
N_h \;&\triangleq\; \wedge\; \vee\; (min = 59) \wedge (chg_m = chg_h)\\
&\;\;\;\; \vee\; (min = 0) \wedge (chg_m \neq chg_h)\\
&\;\; \wedge\; hr' = (hr \bmod 12) + 1\\
&\;\; \wedge\; chg'_h = \neg chg_h\\
&\;\; \wedge\; \langle min, chg_m \rangle' = \langle min, chg_m \rangle\\
\Psi \;&\triangleq\; \wedge\; Init_\Psi\\
&\;\; \wedge\; \Box[N_m \vee N_h]_{\langle hr,\, min,\, chg_m,\, chg_h \rangle}\\
&\;\; \wedge\; \mathrm{WF}_{\langle hr,\, min,\, chg_m,\, chg_h \rangle}(N_m \vee N_h)
\end{aligned}
$$

Fig. 4. Another internal specification of the hour-minute clock.

where Φ is defined in Figure 1, is left as a nice exercise for the reader. The proof that $\exists\, chg_h, chg_m : \Psi$ implies $\exists\, chg : \Phi$ uses the refinement mapping $\overline{chg} \triangleq (chg_h \neq chg_m)$. The proof of the converse implication uses the refinement mapping

$$\overline{chg_h} \;\triangleq\; chg \wedge (min = 59) \qquad\qquad \overline{chg_m} \;\triangleq\; chg \wedge (min = 0)$$

The specifications Ψ_h and Ψ_m of the hour and minute processes appear in Figure 5. We now sketch the proof that Ψ is the composition of those two specifications.

$$
\begin{aligned}
Init_m &\triangleq \land\ min \in \{0, \ldots, 59\} \\
&\ \land\ chg_m = \text{TRUE} \\
Init_h &\triangleq \land\ hr \in \{1, \ldots, 12\} \\
&\ \land\ chg_h = \text{TRUE} \\
\Psi_h &\triangleq Init_h \land \Box[N_h]_{\langle hr,\, chg_h \rangle} \quad \land \mathrm{WF}_{\langle hr,\, chg_h \rangle}(N_h) \\
\Psi_m &\triangleq Init_m \land \Box[N_m]_{\langle min,\, chg_m \rangle} \land \mathrm{WF}_{\langle min,\, chg_m \rangle}(N_m)
\end{aligned}
$$

Fig. 5. Definition of the specifications Ψ_h and Ψ_m.

Theorem 4. $\Psi \ \equiv\ \Psi_m \land \Psi_h$

$\langle 1 \rangle 1.$ $Init_\Psi \ \equiv\ Init_m \land Init_h$

$\langle 1 \rangle 2.$ $\Box[N_m \lor N_h]_{\langle hr,\, min,\, chg_m,\, chg_h \rangle} \ \equiv\ \Box[N_m]_{\langle min,\, chg_m \rangle} \ \land\ \Box[N_h]_{\langle hr,\, chg_h \rangle}$

$\langle 2 \rangle 1.$ $[N_m \lor N_h]_{\langle hr,\, min,\, chg_m,\, chg_h \rangle} \ \equiv\ [N_m]_{\langle min,\, chg_m \rangle} \ \land\ [N_h]_{\langle hr,\, chg_h \rangle}$

$\langle 2 \rangle 2.$ Q.E.D.

PROOF: By $\langle 2 \rangle 1$ and the rules $\dfrac{A \Rightarrow B}{\Box A \Rightarrow \Box B}$ and $\Box(A \land B) \equiv \Box A \land \Box B$.

$\langle 1 \rangle 3.$ $\land\ \Psi \ \Rightarrow\ \mathrm{WF}_{\langle min,\, chg_m \rangle}(N_m) \land \mathrm{WF}_{\langle hr,\, chg_h \rangle}(N_h)$

$\land\ \Psi_m \land \Psi_h \ \Rightarrow\ \mathrm{WF}_{\langle hr,\, min,\, chg_m,\, chg_h \rangle}(N_m \lor N_h)$

$\langle 1 \rangle 4.$ Q.E.D.

PROOF: By $\langle 1 \rangle 1$–$\langle 1 \rangle 3$.

Ignoring liveness (step $\langle 1 \rangle 3$), the hard part is proving $\langle 2 \rangle 1$. This step is an immediate consequence of the following propositional logic tautology, which we call the $\lor \leftrightarrow \land$ rule.

$$
\frac{N_i \land (j \neq i) \ \Rightarrow\ (v_j' = v_j) \quad \text{for } 1 \leq i, j \leq n}{[N_1 \lor \ldots \lor N_n]_{\langle v_1, \ldots, v_n \rangle} \ =\ [N_1]_{v_1} \ \land\ \ldots\ \land\ [N_n]_{v_n}}
$$

Its proof is left as an exercise for the reader.

6.2 Decomposing Proofs

In pseudo-programming language terminology, a compositional proof of refinement (implementation) is one performed by breaking a specification into the parallel composition of processes and separately proving the refinement of each process.

The most naive translation of this into mathematics is that we want to prove $\Lambda \Rightarrow \Sigma$ by writing Σ as $\Sigma_1 \land \Sigma_2$ and proving $\Lambda \Rightarrow \Sigma_1$ and $\Lambda \Rightarrow \Sigma_2$ separately. Such a decomposition accomplishes little. The lower-level specification

Λ is usually much more complicated than the higher-level specification Σ, so decomposing Σ is of no interest.

A slightly less naive translation of compositional reasoning into mathematics involves writing both Λ and Σ as compositions. This leads to the following proof of $\Lambda \Rightarrow \Sigma$.

$\langle 1 \rangle 1.$ $\wedge\ \Lambda \ \equiv\ \Lambda_1 \wedge \Lambda_2$
$\quad\ \wedge\ \Sigma \ \equiv\ \Sigma_1 \wedge \Sigma_2$
  PROOF: Use the $\vee \leftrightarrow \wedge$ rule.
$\langle 1 \rangle 2.$ $\Lambda_1 \ \Rightarrow\ \Sigma_1$
$\langle 1 \rangle 3.$ $\Lambda_2 \ \Rightarrow\ \Sigma_2$
$\langle 1 \rangle 4.$ Q.E.D.
  PROOF: By $\langle 1 \rangle 1$–$\langle 1 \rangle 3$ and the $\wedge$-composition and act-stupid rules.

The use of the act-stupid rule in the final step tells us that we have a problem. Indeed, this method works only in the most trivial case. Proving each of the implications $\Lambda_i \Rightarrow \Sigma_i$ requires proving $\Lambda_i \Rightarrow \mathit{Inv}_i$ for some invariant Inv_i. Except when each process accesses only its own variables, so there is no communication between the two processes, Inv_i will have to mention the variables of both processes. As our clock example illustrates, the next-state relation of each process's specification allows arbitrary changes to the other process's variables. Hence, Λ_i can't imply any nontrivial invariant that mentions the other process's variables. So, this proof method doesn't work.

Think of each process Λ_i as the other process's *environment*. We can't prove $\Lambda_i \Rightarrow \Sigma_i$ because it asserts that Λ_i implements Σ_i in the presence of arbitrary behavior by its environment—that is, arbitrary changes to the environment variables. No real process works in the face of completely arbitrary environment behavior.

Our next attempt at compositional reasoning is to write a specification E_i of the assumptions that process i requires of its environment and prove $\Lambda_i \wedge E_i \Rightarrow \Sigma_i$. We hope that one process doesn't depend on all the details of the other process's specification, so E_i will be much simpler than the other process's specification Λ_{2-i}. We can then prove $\Lambda \Rightarrow \Sigma$ using the following propositional logic tautology.

$$\frac{\begin{array}{ll} \Lambda_1 \wedge \Lambda_2 \ \Rightarrow\ E_1 & \quad \Lambda_1 \wedge \Lambda_2 \ \Rightarrow\ E_2 \\ \Lambda_1 \wedge E_1 \ \Rightarrow\ \Sigma_1 & \quad \Lambda_2 \wedge E_2 \ \Rightarrow\ \Sigma_2 \end{array}}{\Lambda_1 \wedge \Lambda_2 \ \Rightarrow\ \Sigma_1 \wedge \Sigma_2}$$

However, this requires proving $\Lambda \Rightarrow E_i$, so we still have to reason about the complete lower-level specification Λ. What we need is a proof rule of the following form

$$\frac{\begin{array}{ll} \Sigma_1 \wedge \Sigma_2 \ \Rightarrow\ E_1 & \quad \Sigma_1 \wedge \Sigma_2 \ \Rightarrow\ E_2 \\ \Lambda_1 \wedge E_1 \ \Rightarrow\ \Sigma_1 & \quad \Lambda_2 \wedge E_2 \ \Rightarrow\ \Sigma_2 \end{array}}{\Lambda_1 \wedge \Lambda_2 \ \Rightarrow\ \Sigma_1 \wedge \Sigma_2} \tag{4}$$

In this rule, the hypotheses $\Lambda \Rightarrow E_i$ of the previous rule are replaced by $\Sigma \Rightarrow E_i$. This is a great improvement because Σ is usually much simpler than Λ. A rule like (4) is called a *decomposition theorem*.

Unfortunately, (4) is not valid for arbitrary formulas. (For example, let the Λ_i equal TRUE and all the other formulas equal FALSE.) Roughly speaking, (4) is valid if all the properties are safety properties, and if Σ_i and E_i modify disjoint sets of variables, for each i. A more complicated version of the rule allows the Λ_i and Σ_i to include liveness properties; and the condition that Σ_i and E_i modify disjoint sets of variables can be replaced by a weaker, more complicated requirement. Moreover, everything generalizes from two conjuncts to n in a straightforward way. All the details can be found in [2].

6.3 Why Bother?

What have we accomplished by using a decomposition theorem of the form (4)? As our clock example shows, writing a specification as the conjunction of n processes rests on an equivalence of the form

$$\Box[N_1 \lor \ldots \lor N_n]_{\langle v_1,\ldots v_n\rangle} \;\equiv\; \Box[N_1]_{v_1} \land \ldots \land \Box[N_n]_{v_n}$$

Replacing the left-hand side by the right-hand side essentially means changing from disjunctive normal form to conjunctive normal form. In a proof, this replaces $\lor$-composition with $\land$-composition. Such a trivial transformation is not going to simplify a proof. It just changes the high-level structure of the proof and rearranges the lower-level steps.

Not only does this transformation not simplify the final proof, it may add extra work. We have to invent the environment specifications E_i, and we have to check the hypotheses of the decomposition theorem. Moreover, handling liveness can be problematic. In the best of all possible cases, the specifications E_i will provide useful abstractions, the extra hypotheses will follow directly from existing theorems, and the decomposition theorem will handle the liveness properties. In this best of all possible scenarios, we still wind up only doing exactly the same proof steps as we would in proving the implementation directly without decomposing it.

This form of decomposition is popular among computer scientists because it can be done in a pseudo-programming language. A conjunction of complete specifications like $\Lambda_1 \land \Lambda_2$ corresponds to parallel composition, which can be written in a PPL as $\Lambda_1 \| \Lambda_2$. The PPL is often sufficiently inexpressive that all the specifications one can write trivially satisfy the hypotheses of the decomposition theorem. For example, the complications introduced by liveness are avoided if the PPL provides no way to express liveness.

Many computer scientists prefer to do as much of a proof as possible in the pseudo-programming language, using its special-purpose rules, before being forced to enter the realm of mathematics with its simple, powerful laws. They denigrate the use of ordinary mathematics as mere “semantic reasoning”. Because mathematics can so easily express the underlying semantics of a pseudo-programming language, any proof in the PPL can be translated to a semantic proof. Any law for manipulating language constructs will have a counterpart that is a theorem of ordinary mathematics for manipulating a particular class of formulas. Mathematics can also provide methods of reasoning that have no counterpart in the PPL because of the PPL's limited expressiveness. For example, because it can directly mention the control state, an invariance proof based on ordinary mathematics is often simpler than one using the Owicki-Gries method.

Many computer scientists believe that their favorite pseudo-programming language is better than mathematics because it provides wonderful abstractions such as message passing, or synchronous communication, or objects, or some other popular fad. For centuries, bridge builders, rocket scientists, nuclear physicists, and number theorists have used their own abstractions. They have all expressed those abstractions directly in mathematics, and have reasoned "at the semantic level". Only computer scientists have felt the need to invent new languages for reasoning about the objects they study.

Two empirical laws seem to govern the difficulty of proving the correctness of an implementation, and no pseudo-programming language is likely to circumvent them: (1) the length of a proof is proportional to the product of the length of the low-level specification and the length of the invariant, and (2) the length of the invariant is proportional to the length of the low-level specification. Thus, the length of the proof is quadratic in the length of the low-level specification. To appreciate what this means, consider two examples. The specification of the lazy caching algorithm of Afek, Brown, Merritt [3], a typical high-level algorithm, is 50 lines long. The specification of the cache coherence protocol for a new computer that we worked on is 1900 lines long. We expect the lengths of the two corresponding correctness proofs to differ by a factor of 1500.

The most effective way to reduce the length of an implementation proof is to reduce the length of the low-level specification. A specification is a mathematical abstraction of a real system. When writing the specification, we must choose the level of abstraction. A higher-level abstraction yields a shorter specification. But a higher-level abstraction leaves out details of the real system, and a proof cannot detect errors in omitted details. Verifying a real system involves a tradeoff between the level of detail and the size (and hence difficulty) of the proof.

A quadratic relation between one length and another implies the existence of a constant factor. Reducing this constant factor will shorten the proof. There are several ways to do this. One is to use better abstractions. The right abstraction can make a big difference in the difficulty of a proof. However, unless one has been really stupid, inventing a clever new abstraction is unlikely to help by more than a factor of five. Another way to shorten a proof is to be less rigorous, which means stopping a hierarchical proof one or more levels sooner. (For real systems, proofs reach a depth of perhaps 12 to 20 levels.) Choosing the depth of a proof provides a tradeoff between its length and its reliability. There are also silly ways to reduce the size of a proof, such as using small print or writing unstructured, hand-waving proofs (which are known to be completely unreliable).

Reducing the constant factor still does not alter the essential quadratic nature of the problem. With systems getting ever more complicated, people who try to verify them must run very hard to stay in the same place. Philosophically motivated theories of compositionality will not help.

6.4 When a Decomposition Theorem is Worth the Bother

As we have observed, using a decomposition theorem can only increase the total amount of work involved in proving that one specification implements another. There is one case in which it's worth doing the extra work: when the computer does a lot of it for you. If we decompose the specifications Λ and Σ into n conjuncts Λ_i and Σ_i, the hypotheses of the decomposition theorem become $\Sigma \Rightarrow E_i$ and $\Lambda_i \wedge E_i \Rightarrow \Sigma_i$, for $i = 1, \ldots, n$. The specification Λ is broken into the smaller components Λ_i. Sometimes, these components will be small enough that the proof of $\Lambda_i \wedge E_i \Rightarrow \Sigma_i$ can be done by model checking—using a computer to examine all possible equivalence classes of behaviors. In that case, the extra work introduced by decomposition will be more than offset by the enormous benefit of using model checking instead of human reasoning. An example of such a decomposition is described in [7].

7 Composing Specifications

There is one situation in which compositional reasoning cannot be avoided: when one wants to reason about a component that may be used in several different systems.

The specifications we have described thus far have been *complete-system* specifications. Such specifications describe all behaviors in which both the system and its environment behave correctly. They can be written in the form $S \wedge E$, where S describes the system and E the environment. For example, if we take the component to be our clock example's hour process, then S is the formula Ψ_h and E is Ψ_m. (The hour process's environment consists of the minute process.)

If a component may be used in multiple systems, we need to write an *open-system* specification—one that specifies the component itself, not the complete system containing it. Intuitively, the component's specification asserts that it satisfies S if the environment satisfies E. This suggests that the component's open-system specification should be the formula $E \Rightarrow S$. This specification allows behaviors in which the system misbehaves, if the environment also misbehaves. It turns out to be convenient to rule out behaviors in which the system misbehaves first. (Such behaviors could never be allowed by a real implementation, which cannot know in advance that the environment will misbehave.) We therefore take as the specification the formula $E \stackrel{+}{\rhd} S$, which is satisfied by a behavior in which S holds as long as E does. The precise definition of $\stackrel{+}{\rhd}$ and the precise statement of the results about open-system specifications can be found in [2].

The basic problem of compositional reasoning is showing that the composition of component specifications satisfies a higher-level specification. This means proving that the conjunction of specifications of the form $E \stackrel{+}{\rhd} S$ implies another specification of that form. For two components, the proof rule we want is:

$$\frac{E \wedge S_1 \wedge S_2 \;\Rightarrow\; E_1 \wedge E_2 \wedge S}{(E_1 \stackrel{+}{\rhd} S_1) \;\wedge\; (E_2 \stackrel{+}{\rhd} S_2) \;\Rightarrow\; (E \stackrel{+}{\rhd} S)}$$

Such a rule is called a *composition theorem.* As with the decomposition theorem (4), it is valid only for safety properties under certain disjointness assumptions; a more complicated version is required if S and the S_i include liveness properties.

Composition of open-system specifications is an attractive problem, having obvious application to reusable software and other trendy concerns. But in 1997, the unfortunate reality is that engineers rarely specify and reason formally about the systems they build. It is naive to expect them to go to the extra effort of proving properties of open-system component specifications because they might re-use those components in other systems. It seems unlikely that reasoning about the composition of open-system specifications will be a practical concern within the next 15 years. Formal specifications of systems, with no accompanying verification, may become common sooner. However, the difference between the open-system specification $E \stackrel{+}{\rhd} M$ and the complete-system specification $E \wedge M$ is one symbol—hardly a major concern in a specification that may be 50 or 200 pages long.

8 Conclusion

What should we do if faced with the problem of finding errors in the design of a real system? The complete design will almost always be too complicated to handle by formal methods. We must reason about an abstraction that represents as much of the design as possible, given the limited time and manpower available.

The ideal approach is to let a computer do the verification, which means model checking. Model checkers can handle only a limited class of specifications. These specifications are generally small and simple enough that it makes little difference in what language they are written—conventional mathematics or pseudo-programming languages should work fine. For many systems, abstractions that are amenable to model checking omit too many important aspects of the design. Human reasoning—that is, mathematical proof—is then needed. Occasionally, this reasoning can be restricted to rewriting the specification as the composition of multiple processes, decomposing the problem into subproblems suitable for model checking. In many cases, such a decomposition is not feasible, and mathematical reasoning is the only option.

Any proof in mathematics is compositional—a hierarchical decomposition of the desired result into simpler subgoals. A sensible method of writing proofs will make that hierarchical decomposition explicit, permitting a tradeoff between the length of the proof and its rigor. Mathematics provides more general and more powerful ways of decomposing a proof than just writing a specification as the parallel composition of separate components. That particular form of decomposition is popular only because it can be expressed in terms of the pseudo-programming languages favored by computer scientists.

Mathematics has been developed over two millennia as the best approach to rigorous human reasoning. A couple of decades of pseudo-programming language design poses no threat to its pre-eminence. The best way to reason mathematically is to use mathematics, not a pseudo-programming language.

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. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.
3. Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.
4. Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.
5. E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.
6. Edsger W. Dijkstra. A personal summary of the Gries-Owicki theory. In Edsger W. Dijkstra, editor, *Selected Writings on Computing: A Personal Perspective*, chapter EWD554, pages 188–199. Springer-Verlag, New York, Heidelberg, Berlin, 1982.
7. 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.
8. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.
9. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
10. Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August-September 1995.
11. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.