A Formal Basis for the Specification of Concurrent Systems

Notes for the NATO Advanced Study Institute, Izmir, Turkey

Leslie Lamport

June 26, 2000

Contents

1 Describing Complete Systems

1.1 Systems as Sets of Behaviors

We are ultimately interested in specifying systems—usually systems that are to be implemented as concurrent programs. We cannot claim to have written a formal specification of such a system unless we can formally state what it means for a program to satisfy the specification. Such a statement requires a formal *semantics* for programs—that is, the assignment to every program of some mathematical object that denotes the "meaning" of the program. We therefore begin with an informal sketch of a formal semantics for concurrent programs.

We will ignore many interesting aspects of program semantics, including the specification of most language constructs and the issue of compositionality. These questions are addressed in [6].

1.1.1 Behaviors

While sequential programs can often be described in terms of their input and output, concurrent programs must be described in terms of their behavior. A *behavior*, also called an *execution sequence*, is a sequence (finite or infinite)

$$s_0 \stackrel{\alpha_1}{\rightarrow} s_1 \stackrel{\alpha_2}{\rightarrow} s_2 \stackrel{\alpha_3}{\rightarrow} s_3 \ldots$$

where the s_i are states and the α_i are atomic operations. We claim, but will not attempt to justify this claim here, that the behavior of every discrete system, be it hardware or software, can be formally represented as such a sequence.

A triple $s \stackrel{\alpha}{\rightarrow} t$ is called an α *transition*; s is called the initial state and t is called the final state of the transition. The above behavior can be viewed as a sequence of transitions $s_{i-1} \stackrel{\alpha_i}{\rightarrow} s_i$ such that the final state of each transition is the initial state of the next transition.

As an example, we consider the simple program of Figure 1, where x and y are assumed to be integer-valued variables. The angle brackets indicate that each assignment statement is a single atomic operation. The statements are labeled α and β, and the control point following statement β is labeled γ. The state of the program consists of an assignment of values to the variables x and y and an assignment of one of the three values α, β, or γ to the "program counter" pc that defines the current locus of control. Let $(x = 4, y = -15, pc = \beta)$ denote the state in which the value of x is 4, the

$$\begin{array}{l}\alpha\text{: } \langle y := 7\ \rangle;\\ \beta\text{: } \langle x := x^2 + y\rangle\\ \gamma\text{:}\end{array}$$

Figure 1: A simple program.

value of y is -15, and control is right at the beginning of statement β. One possible behavior of this program is

$$(x = -2, y = 128, pc = \alpha) \xrightarrow{\alpha} \\ (x = -2, y = 7, pc = \beta) \xrightarrow{\beta} (x = 11, y = 7, pc = \gamma)$$

In fact, the set of all sequences of the form

$$(x = x_0, y = y_0, pc = \alpha) \xrightarrow{\alpha} \\ (x = x_0, y = 7, pc = \beta) \xrightarrow{\beta} (x = x_0^2 + 7, y = 7, pc = \gamma)$$

for arbitrary integers x_0 and y_0, are possible execution sequences of the program. (The value of an integer variable is assumed to be a mathematical integer, which can be arbitrarily large.) One might expect these to be the only possible behaviors of this program, but it is convenient to allow certain others that are described below.

In this example, the atomic-operation labels on the arrows in the execution sequence are redundant—just looking at the sequence of states allows us to fill in the arrow labels. In fact, this is true for any reasonable sequential or concurrent program. Labeling the transition with the action makes it easy to formalize the notion of who is performing an action. For now, these labels can be regarded as a harmless bit of redundancy.

No significance should be attached to the use of the same letters (α and β) to denote both atomic operations (arrow labels) and control point values (values of pc). We simply find it more convenient to overload these symbols than to introduce an extra set of program labels.

1.1.2 Behavioral Semantics

We define a semantics in which the meaning of a program is a set of behaviors—the set of all possible behaviors that the program is allowed to exhibit. There are usually considered to be two general approaches to describing a set of behaviors: constructive (or operational) and axiomatic. In the constructive approach, one gives rules for generating sequences. One can view the program of Figure 1 as a constructive specification by considering it to be a method for generating a behavior given initial values for x and y. The semantics of the program is the set of all sequences generated by this method starting from arbitrary integer value for x and y. In the axiomatic approach, one describes the set of behaviors by a collection of axioms. The meaning of the program is the set of all sequences that satisfy the axioms.

Formal mathematics is ultimately reducible to axiomatic reasoning, and a constructive method rests upon axioms. The distinction between constructive and axiomatic specifications is therefore illusory. When one formally describes a constructive method, it becomes axiomatic. However, there are axiomatic methods that are nonconstructive—that is, for which there is no clearly operational way to describe the set of behaviors that they define. Thus, constructive methods are really a special class of axiomatic ones.

Classifying constructive methods as a special case of axiomatic ones may seem like a dubious bit of reductionism, obscuring an essential distinction. We hope that the semantics described below will serve to counter that objection. It is certainly axiomatic, since every specification can be written as a formula in a formal logical system. However, most of the axioms will be written in a distinctly operational way.

1.1.3 Specifying Behaviors with Axioms—A Simple Approach

To provide an axiomatic specification of the above set of behaviors for the program of Figure 1, we first observe that this set of behaviors is determined by the following four separate rules:

- The initial state has $pc = \alpha$.
- When $pc = \alpha$, the next transition is an α transition that sets y to 7 and sets pc to β.
- When $pc = \beta$, the next transition is a β transition that sets x to $x^2 + y$ and sets pc to γ.
- When $pc = \gamma$, no more transitions can occur.

The four rules are informal axioms that specify the set of behaviors for the program. To turn them into a true axiomatic specification, they must be expressed in some formal system. Temporal logic is an excellent formal system for this purpose. We will not give a formal description of temporal logic here, and will write axioms using somewhat stilted English. All the properties in our specifications that are expressed informally in English can be translated into a temporal logic formula by anyone well versed in the temporal logic described in the appendix of [7].

The above four rules can be rewritten more precisely as the following temporal assertions.

> *Initial Axiom*: In the starting state, $pc = \alpha$ and x and y have integer values.
>
> α *Transition Axiom*: It is always the case that if $pc = \alpha$ and $x = x_0$, then the next action is labeled α and, in the next state, $pc = \beta$, $x = x_0$, and $y = 7$.
>
> β *Transition Axiom*: It is always the case that if $pc = \beta$, $x = x_0$, and $y = y_0$, then the next action is labeled β and, in the next state, $pc = \gamma$, $x = x_0^2 + y_0$, and $y = y_0$.
>
> *Termination Axiom*: It is always the case that if $pc = \gamma$, then there is no next state.

If we add the assumption that the state is specified by the values of x, y, and pc, then the set of execution sequences that satisfy these four axioms is precisely the set of sequences described above.

This is the obvious method of writing a temporal logic specification of the set of execution sequences described above. However, it turns out that this is the wrong way to do it. The use of the "in the next state" temporal operator raises serious difficulties in defining what it means to implement the program of Figure 1 by a lower-level program. Instead, we now describe a less obvious method that does not use this temporal operator.

Note that the "next action" operator causes no problem and will be used. The temporal logic of [7] can express the concept of the next action but not the concept of the next state.

1.1.4 Specifying Behaviors with Axioms—the Right Way

In the axiomatic approach, one usually specifies a set of behaviors by writing a list of properties, which specifies the set of all behaviors that satisfy all the properties. It is convenient to distinguish two types of properties: safety and liveness. Intuitively, a safety property asserts that something (presumably bad) will not happen, while a liveness property asserts that something (presumably good) will eventually happen. (A formal characterization of these properties can be found in [1].)

In specifying the set of behaviors for the program of Figure 1, safety properties assert that the program does not perform an incorrect action—for example, a safety axiom would rule out execution sequences whose first atomic action sets the value of y to 13, or changes the value of x. Liveness properties assert that the program does eventually make progress, unless it has reached its halting state. For example, a liveness axiom would assert that if control is at α, then eventually control will be at β.

In place of the "in the next state" temporal operator, we use the "until" operator, where "A until B" means that A remains true at least until the next time that B becomes true.

In our next attempt at specifying the program of Figure 1, we use the same initial and termination axiom as before, but, for each atomic operation, we have a pair of axioms, one for safety and one for liveness.

> α *Transition Axiom*: (Safety) It is always the case that if $pc = \alpha$, $x = x_0$, and $y = y_0$, then the next action is labeled α and $pc = \alpha$, $x = x_0$, and $y = y_0$ remain true until $pc = \beta$, $x = x_0$, and $y = 7$.
>
> (Liveness) It is always the case that if $pc = \alpha$ then eventually $pc \neq \alpha$.
>
> β *Transition Axiom*: (Safety) It is always the case that if $pc = \beta$, $x = x_0$, and $y = y_0$, then the next action is labeled β and $pc = \beta$, $x = x_0$, and $y = y_0$ remain true until $pc = \gamma$, $x = x_0^2 + y_0$, and $y = y_0$.
>
> (Liveness) It is always the case that if $pc = \beta$ then eventually $pc \neq \beta$.

Note the general pattern: instead of saying that if A holds then B holds in the next state, we say that A holds until B does (a safety property) and that if A holds then eventually A will cease to hold (a liveness property).

All the execution sequences described above for the program of Figure 1 satisfy these axioms. However, there are additional sequences that also satisfy the axioms. For example, the axiom for statement α states that, starting in a state $(x = x_0, y = y_0, pc = \alpha)$, the next arrow must be labeled α, and the only way that the state can change is for the next state to become $(x = x_0, y = 7, pc = \beta)$. However, it does not rule out "stuttering" actions labeled α that leave the state unchanged.

Assuming that the state is completely determined by the values of x, y, and pc, the above axioms specify the set of all execution sequences starting in state $(x = x_0, y = y_0, pc = \alpha)$, for arbitrary integers x_0 and y_0, followed by a finite number (possibly zero) of actions $\xrightarrow{\alpha}$ $(x = x_0, y = y_0, pc = \alpha)$, followed by an action $\xrightarrow{\beta}$ $(x = x_0, y = 7, pc = \beta)$, followed by a finite number of actions $\xrightarrow{\beta}$ $(x = x_0, y = 7, pc = \beta)$, followed by an action $\xrightarrow{\beta}$ $(x = x_0^2 + 7, y = 7, pc = \gamma)$.

The extra stuttering actions allowed by this specification may seem burdensome. We shall see later that, on the contrary, they are the key to the proper definition of what it means to implement a program with a lower-level program. In fact, the "in the next state" temporal operator is bad precisely because it allows one to specify that there is no stuttering.

Although they may seem strange, there is no reason not to allow stuttering actions. The state includes all visible information about what the program is doing. An atomic action that does not change the state has no visible effect, and an action with no visible effect must be harmless.

Since stuttering actions are harmless, there is no reason not to allow behaviors of the program of Figure 1 to have extra stuttering actions labeled γ at the end. In fact, it turns out to be useful to assume that, instead of the execution sequences being finite, they all end with an infinite sequence of γ actions that do not change the state. Again, this creates no problems because there is no way to distinguish a program that has halted from one that is continually doing nothing. (Indeed, a *halt* instruction does not turn off a computer; it causes the computer to cycle endlessly, doing nothing.)

We now rewrite the above specification to require that execution sequences end with an infinite sequence of stuttering γ actions. This requires adding an axiom for the γ action. We also rewrite the axioms for the α and β actions in an even more baroque form that leads to the proper generalization for concurrent programs. To make the axioms easier to understand, we express them in a form that mentions the next state, although explicitly allowing stuttering actions. These axioms can be expressed in terms of the "until" operator without the "next state" operator, but doing so results in rather convoluted assertions. (In fact, the major problem with the "until" operator is that it leads to formulas that are hard to understand.) The same initialization axiom is used, but the Termination Axiom is replaced by the Completion Axiom.

> α *Transition Axiom*: (Safety) It is always the case that, if the next action is labeled α, then $pc = \alpha$ (in the current state) and the next state is either unchanged or else has pc set equal to β, y set equal to 7, and all other variables unchanged.

(Liveness) It is always the case that, if there are infinitely many α actions, then eventually $pc \neq \alpha$.

β Transition Axiom: (Safety) It is always the case that, if $x = x_0$, $y = y_0$ (in the current state) and the next action is labeled β, then $pc = \beta$ and the next state is either unchanged or else has pc changed to γ, x set to $x_0^2 + y_0$, and all other variables unchanged.

(Liveness) It is always the case that, if there are infinitely many β actions, then eventually $pc \neq \beta$.

γ Transition Axiom: (Safety) It is always the case that, if the next action is labeled γ, then $pc = \gamma$ in the current state and the next state is unchanged.

Completion Axiom: It is always the case that the next action is labeled either α, β, or γ.

Note that there is no liveness axiom for the γ action, and the Completion Axiom expresses a safety property.

Most readers will not find it immediately obvious that this set of axioms specifies the same set of execution sequences described above; we now show that it does. The initial axiom implies that the first state has $pc = \alpha$. The Completion Axiom implies that the first action must be labeled α, β, or γ. However, the other axioms imply that only an α action can occur when $pc = \alpha$. Hence, the first action must be an α action, which can either leave the state unchanged or else set y to 7. The liveness axiom for α implies that there cannot be an infinite sequence of α actions that leave the state unchanged, because that would mean that there would be an infinite number of α actions and pc would remain forever equal to α, contrary to the axiom. Hence, there can be some finite number of α actions that do not change the state, but they must be followed by an α action that sets y to 7 and sets pc to β. Similar reasoning shows that there must then be a finite number (possibly zero) of β actions that leave the state unchanged followed by one that sets x to $x_0^2 + 7$ and sets pc to γ. Finally, the Completion Axiom implies that there must be an infinite number of actions (since there is always a next one), and the only action possible when $pc = \gamma$ is a γ action, so the sequence must end with an infinite sequence of γ actions that leave the state unchanged.

cobegin
α: $\langle y := 7 \rangle$;
β: $\langle x := x^2 + y \rangle$
γ:
$\square$
while δ: $\langle x \neq 7 \rangle$
do ϵ: $\langle x := x + z \rangle$**od**
η:
coend
θ:

Figure 2: A simple concurrent program.

1.1.5 Concurrent Programs

Thus far, we have considered only a sequential program. Let us now turn to the simple concurrent program of Figure 2. In it, the program of Figure 1 is one process in a two-process program. A state of this program is an assignment of values to the integer variables x, y, and z, and to the "program counter" pc, which denotes the control state of the two processes, or else equals θ when both processes have halted. Let $(x = 7, y = -2, pc = (\beta, \delta))$ denote the state that assigns the value 7 to x, -2 to y, 4 to z, and in which control in the first process is at β and control in the second process is at δ. Let pc_1 and pc_2 denote the two components of pc, so, in this state, $pc_1 = \beta$ and $pc_2 = \delta$. For notational convenience, let $(\gamma, \eta) = \theta$, so if $pc = \theta$ then $pc_1 = \gamma$ and $pc_2 = \eta$.

An execution sequence of this program consists of an interleaving of atomic operations from the two processes. One such execution begins

$$\begin{aligned}
&(x = 0, y = 1, z = 5, pc = (\alpha, \delta)) \xrightarrow{\alpha} (x = 0, y = 1, z = 5, pc = (\alpha, \delta)) \xrightarrow{\delta} \\
&\quad (x = 0, y = 1, z = 5, pc = (\alpha, \epsilon)) \xrightarrow{\alpha} (x = 0, y = 7, z = 5, pc = (\beta, \epsilon)) \xrightarrow{\beta} \\
&\quad (x = 7, y = 7, z = 5, pc = (\gamma, \epsilon)) \xrightarrow{\epsilon} (x = 12, y = 7, z = 5, pc = (\gamma, \delta)) \xrightarrow{\delta} \\
&\quad (x = 12, y = 7, z = 5, pc = (\gamma, \delta)) \xrightarrow{\delta} (x = 12, y = 7, z = 5, pc = (\gamma, \epsilon)) \xrightarrow{\epsilon} \\
&\quad (x = 17, y = 7, z = 5, pc = (\gamma, \delta)) \xrightarrow{\delta} \cdots
\end{aligned}$$

and continues with the second process cycling forever through δ and ϵ actions (some of which may be stuttering actions).

This program also permits halting executions, which occur if statement

δ happens to be executed when x has the value 7. Such execution sequences end with an infinite string of stuttering θ actions.

We specify the set of behaviors of this program with the following axioms. For convenience, pc_1 and pc_2 are considered to be variables when asserting that "no other variables" are changed. Note that the axioms for α and β are almost the same as before, the only change being the substitution of pc_1 for pc.

> *Initial Axiom*: In the starting state, $pc = (\alpha, \delta)$ and x, y, and z have integer values.
>
> α *Transition Axiom*: (Safety) It is always the case that, if the next action is labeled α, then $pc_1 = \alpha$ (in the current state) and the next state is either unchanged or else has pc_1 set equal to β, y set equal to 7, and all other variables are unchanged.
>
> (Liveness) It is always the case that, if there are infinitely many α actions, then eventually $pc_1 \neq \alpha$.
>
> β *Transition Axiom*: (Safety) It is always the case that, if $x = x_0$, $y = y_0$ (in the current state) and the next action is labeled β, then $pc_1 = \beta$ and the next state is either unchanged or else has pc_1 set to γ, x set to $x_0^2 + y_0$, and all other variables are unchanged.
>
> (Liveness) It is always the case that, if there are infinitely many β actions, then eventually $pc_1 \neq \beta$.
>
> δ *Transition Axiom*: (Safety) It is always the case that, if the next action is labeled δ, then $pc_1 = \delta$ (in the current state) and the the next state is either unchanged or else only the value of pc_2 is changed and its value in the next state equals ϵ if $x \neq 7$ in the current state and equals η if $x = 7$ in the current state.
>
> (Liveness) It is always the case that, if there are infinitely many δ actions, then eventually $pc_2 \neq \delta$.
>
> ϵ *Transition Axiom*: (Safety) It is always the case that, if $x = x_0$ and $z = z_0$ (in the current state) and the next action is labeled ϵ, then $pc_2 = \epsilon$ and the next state is either unchanged or else has x set to $x_0 + z_0$, pc_2 changed to ϵ, and all other variables unchanged.
>
> (Liveness) It is always the case that, if there are infinitely many ϵ actions, then eventually $pc_2 \neq \epsilon$.

> θ *Transition Axiom*: (Safety) It is always the case that, if the next action is labeled θ, then $pc = \theta$ in the current state and the next state is unchanged.
>
> *Completion Axiom*: It is always the case that the next action is labeled either α, β, δ, ϵ, or θ.

Note that there are no γ or η actions; when one process has terminated, all the actions are generated by the other process until it halts, at which time only stuttering θ actions occur.

The reader can check that these axioms permit all the execution sequences we expect. They also guarantee that nonstuttering actions keep happening unless the program halts—that is, until pc equals θ. However, they do not rule out the possibility that the second process loops forever and the first process never terminates. A **cobegin** statement whose semantics allows one process to be "starved" in this way is called an *unfair* **cobegin**. An alternative semantics for **cobegin** rules out that possibility, defining a *fair* **cobegin**. To specify the set of behaviors allowed by a fair **cobegin**, we must add another axiom. Let us say that control is in the first process if pc_1 equals α or β, and that control is in the second process if pc_2 equals δ or ϵ. Let us also say that α and β are the atomic operations of the first process, and that δ and ϵ are the atomic operations of the second process. Fairness is expressed by the following axiom:

> *Fairness Axiom*: For each process of the **cobegin**, if it is always the case that control is in the process then eventually there will be an action labeled with some atomic operation of the process.

The Fairness Axiom insures that the first process must terminate in any execution sequence of the program of Figure 2. To see this, observe that initially $pc_1 = \alpha$, and the Completion Axiom together with the safety axioms for all the actions imply that pc_1 must then remain equal to α unless an α action changes it. We show by contradiction that pc_1 cannot remain forever equal to α. Assume the contrary. Then the Fairness Axiom implies that it is always the case that there will eventually be an action of the first process, which means that there must be infinitely many actions of the first process. Since pc_1 always equals α, the transition axioms imply that the only actions of the first process that can occur are α actions, so there must be infinitely many α actions. The liveness axiom for α then implies that pc_1 must eventually become unequal to α, which is the required contradiction. Thus, we have proved that there must eventually be an α action that changes the value of pc_1. By the safety part of the Action α Axiom, pc_1 can be changed only to β. Hence, we must eventually have $pc_1 = \beta$. A similar argument then shows that pc_1 must eventually equal γ.

This reasoning may seem rather convoluted, but one expects proving properties of a program directly from an axiomatic semantics to be a long-winded affair. We could have written the axioms in a somewhat more straightforward way that would have simplified this proof. However, writing the axioms the way we did should make it clear how one writes similar axioms for any program built from simple sequential constructs and unfair **cobegin**s. There is an axiom describing the initial state, a safety and liveness axiom for each atomic operation, a safety axiom for the stuttering action that represents termination (not necessary if termination is impossible), and a completeness axiom saying that the next action is always one of the program's atomic actions.

When a program contains fair **cobegin**s, an additional fairness axiom is needed for each fair **cobegin**. If all **cobegin**s are assumed to be fair, then the fairness axiom given above can be used. If a fair **cobegin** can be nested inside an unfair one, then the following axiom defines the most natural semantics.

> *Fairness Axiom*: If infinitely many actions are labeled with the atomic operations of the **cobegin**, then, for each process of the **cobegin**, it is always the case that if control is in the process then eventually there will be an action labeled with some atomic operation of the process.

This axiom asserts that no process in the **cobegin** is starved unless the entire **cobegin** is starved.

1.1.6 Programs as Axioms

One normally thinks of the program of Figure 2 as a machine for generating execution sequences—in other words, we base our understanding of the program upon an intuitive constructive semantics. The above discussion showed that we can represent our intuitive understanding (modified to allow stuttering actions) by a set of axioms. Thus, the program can equally well be thought of as an axiomatic description of a set of behaviors.

We ask the reader to change his way of thinking, and to regard the program in just this way: as an axiomatic specification of a set of execution sequences, not as a machine for generating behaviors. Pretend that someone

$$\begin{array}{ll}
a\colon & \langle \text{load } 7 \rangle \\
 & \langle \text{store } Y \rangle \\
b\colon & \langle \text{load } X \rangle \\
 & \langle \text{multiply by } X \rangle \\
 & \langle \text{store } X \rangle \\
 & \langle \text{load } Y \rangle \\
 & \langle \text{add } X \rangle \\
 & \langle \text{store } X \rangle \\
g\colon &
\end{array}$$

Figure 3: Implementation of the program of Figure 1.

has built a machine for generating execution sequences. We will use the program of Figure 2 to determine if this machine is correct, where correctness means that every execution sequence that it generates is in the set specified by this program. Instead of thinking of the program as a mechanism for generating behaviors, think of it as a filter for outlawing incorrect behaviors.

Of course, this way of thinking has no formal significance; formally, all that we have is a correspondence between the program text and the set of behaviors. However, thinking of a program as an axiomatic specification of its set of possible behaviors is a crucial step in understanding hierarchical specification.

1.2 Correctness of an Implementation

Let us now consider what it means for one program to correctly implement another. For our example, we return to the simple sequential program of Figure 1 and consider its implementation with an assembly language program. For simplicity, assume that the assembly language implements infinitely large integers.

The program of Figure 3 is an implementation that might be produced by a very stupid compiler for a computer with a single accumulator. (There are two unnecessary instructions.) The assembly language variables X and Y implement the variables x and y of the higher-level program of Figure 1, statement α is implemented by the instructions at locations a and $a+1$, and statement β is implemented by the instructions at locations b through $b+5$. (We assume that each instruction occupies a single memory location.)

Formally, the programs of Figures 1 and 3 specify sets of execution sequences. We will not bother writing the axioms that define the set of behaviors of the assembly language program. Suffice it to say that the state consists of the values of the variables X and Y, the value of the accumulator, and the value of the program counter PC.

We must define what it means for one set of execution sequences to implement another. The basic idea is that every possible execution sequence of the assembly language program should, when viewed at the higher level, be a possible execution of the higher-level program. What does "viewing at the higher level" mean? We start with what is perhaps the most obvious approach, and show that it does not work.

First we must interpret the state of the lower-level program in terms of the state components of the higher-level one. The values of the assembly-language variables X and Y represent the values of the higher-level variables x and y. If control in the lower-level program is at a, b, or g, then control in the higher-level one is at α, β, or γ, respectively. But what happens when control is at some other point in the assembly language program—for example, at location $b+3$? At that point, X contains an intermediate value of the computation, and its value does not correspond to a valid value of x.

An execution of the higher-level program contains three distinct states, while an execution of the lower-level program contains nine. Three of those nine states, the ones with control at statements a, b, and g, correspond to the three correct states of the higher-level program. The other six states of the lower-level program are "intermediate" states that do not correspond to any states of the higher-level program. We define an implementation to be correct if we can partition the states of the lower-level program into "valid" and "intermediate" states, and define a mapping from valid states to states of the higher-level program such that throwing away the intermediate states and applying the mapping to the remaining sequence of valid states produces a sequence of higher-level states that is a possible behavior of the higher-level program.

This approach is quite natural, and captures the way most people think about implementations. Unfortunately, while it is adequate for sequential programs, it does not work for concurrent ones. In a concurrent program, it is possible for the entire program never to be in a valid state except at the beginning—there could always be some process in an intermediate state. Thus, throwing away the intermediate states leaves us with nothing. Therefore, we must eschew the obvious approach.

The proper approach is more subtle. We cannot throw away any "intermediate" states because, in a concurrent program, almost all the states could be intermediate ones. If we don't throw away any states, how can an execution of the lower-level program with eight different nonstuttering actions correspond to an execution of the higher-level program that has only three different nonstuttering actions? The answer is clear: five of the nonstuttering lower-level actions must correspond to higher-level stuttering actions. How is this possible when the value of X assumes an intermediate value that is never assumed by the higher-level variable x? The answer to this question is more subtle: the value of x is not defined to simply equal X, but is a more complex function of the lower-level program state.

We will define the variables x and y and the control state pc of the higher-level program as functions of the lower-level program state such that: the execution of statement a corresponds to a nonstuttering high-level action α, the execution of statement $a+1$ corresponds to a stuttering action β, the execution of statement b corresponds to a high-level nonstuttering action β, and execution of the remaining statements in the implementation correspond to stuttering actions γ. This is done as follows:

- pc is defined to equal: (i) α if $PC = a$; (ii) β if $PC = a+1$ or $PC = b$; or (iii) γ if $PC > b$.
- x is defined to equal: (i) X if PC equals a, $a+1$, b, or c; (ii) $X^2 + Y$ if PC equals $b+1$ or $b+2$; or (iii) $X+Y$ if PC equals $b+3$, $b+4$, or $b+5$.
- y is defined to equal Y unless $PC = a+1$, in which case it equals 7.

The reader should check that, with these definitions, execution of the lower-level program statements has the higher-level interpretation indicated above—for example, that executing the store instruction at $b+2$ does not change the value of x, y, or pc, so it is a stuttering action.

Although this procedure for proving that an assembly language program implements a higher-level program works in this example, it is not obvious that it works in general. In fact, the method works only if every variable of the higher-level program can be represented as a function of the lower-level variables (including the "variable" PC). This is not always the case. For example, an optimizing compiler could discover that a variable is never used and decide not to implement it, making it impossible to represent that variable's value as a function of the assembly language program's state. In such a case, one must add "dummy variables" to the assembly-language program—variables and extra statements that are not actually implemented (and take up no memory), but which provide the additional state information needed to represent the higher-level variables. This is seldom necessary in practice and will not be explained in any more detail.

Let us examine more formally what we have done. We expressed each of the state components x, y, and *pc* of the higher-level program as a function of the state components X, Y, and PC of the assembly language program, and we expressed each of the actions α, β, and γ of the higher-level program as a set of actions of the lower-level one. This defines several mappings. First, there is a mapping F_{st} from states of the lower-level program to states of the higher-level one and a mapping F_{ac} from actions of the lower-level program to actions of the higher-level one. For example, if s is the state $(X = 2, Y = 5, PC = b + 1)$ of the assembly language program, then $F_{st}(s)$ is the state $(x = 9, y = 5, pc = \gamma)$ of the higher-level program, and $F_{ac}(a + 1) = \beta$—that is, F_{ac} maps the action $a+1$ of the assembly language program (the action corresponding to the *store* Y atomic operation) to the action β of the higher-level program. The mappings F_{st} and F_{ac} define a mapping F on execution sequences, where F maps an execution sequence

$$s_0 \xrightarrow{\xi_1} s_1 \xrightarrow{\xi_2} s_2 \xrightarrow{\xi_3} s_3 \ldots$$

of the assembly language program to the execution sequence

$$F_{st}(s_0) \xrightarrow{F_{ac}(\xi_1)} F_{st}(s_1) \xrightarrow{F_{ac}(\xi_2)} F_{st}(s_2) \xrightarrow{F_{ac}(\xi_3)} F_{st}(s_3) \ldots$$

of the higher-level program. The implementation is correct if, for every execution sequence σ that satisfies the axioms for the assembly language program, the execution $F(\sigma)$ satisfies the axioms for the higher-level program.

Expressing the state components and actions of the higher-level program as functions of the lower-level program state and actions defines a mapping F^* that maps assertions about the higher-level program into assertions about the lower-level program. For example, if A is the assertion (about the state of the higher-level program) that $y = y_0$, then $F^*(A)$ is the assertion (about the state of the assembly language program) obtained by substituting for y its expression as a function of PC and Y—namely, the assertion that if $PC = a + 1$ then $7 = y_0$, else $Y = y_0$. If B is the assertion that the next action (in an execution of the higher-level program) is labeled β, then $F^*(B)$ is the assertion that the next action (in an execution of the lower-level program) is labeled either $a + 1$ or b.

The mappings F and F^* are related as follows. If A is an assertion about execution sequences of the higher-level program, and σ is an execution sequence of the assembly language program, then the assertion A is true of the execution sequence $F(\sigma)$ if and only if the assertion $F^*(A)$ is true of σ.

1.3 The Formal Description of Systems

Let us now abstract the basic method underlying the above example programs. A *system* is a triple $(\mathcal{B}, \mathbf{S}, \mathbf{A})$, where $\mathbf{S}$ is a set of *states*, $\mathbf{A}$ is a set of *actions*, and $\mathcal{B}$ is a set of *behaviors* of the form

$$s_0 \stackrel{\alpha_1}{\rightarrow} s_1 \stackrel{\alpha_2}{\rightarrow} s_2 \stackrel{\alpha_3}{\rightarrow} s_3 \ldots \tag{1}$$

with each s_i an element of $\mathbf{S}$ and each α_i an element of $\mathbf{A}$. The set $\mathcal{B}$ must also be *invariant under stuttering*, which means that given any behavior (1) in $\mathcal{B}$, the behavior obtained by replacing $s_{i-1} \stackrel{\alpha_i}{\rightarrow} s_i$ with $s_{i-1} \stackrel{\alpha_i}{\rightarrow} s_{i-1} \stackrel{\alpha_i}{\rightarrow} s_i$ is also in $\mathcal{B}$.

We now explain how a system is formally described. This involves the formal specification of the sets $\mathbf{S}$ and $\mathbf{A}$ and of the set of behaviors $\mathcal{B}$.

1.3.1 Specifying States and Actions

Specification of the set $\mathbf{A}$ of actions involves simply naming all the actions. In other words, the set $\mathbf{A}$ is specified simply by enumerating its elements. This is easy for a finite set. Infinite sets of actions are also possible, and are usually enumerated in parametrized form—e.g., by including a set of actions α^i for every positive integer i.

The set $\mathbf{S}$ of states is described in terms of *state functions*, where a state function is a mapping from $\mathbf{S}$ to some set of values called its *range*. In our specification of the program of Figure 1, we used the three state functions x, y, and pc; the range of x and of y was the set of integers, the range of pc was the set $\{\alpha, \beta, \gamma\}$. In general, the set $\mathbf{S}$ is defined by giving a complete collection of state functions $f_1, \ldots, f_n$. An element s of $\mathbf{S}$ is uniquely determined by the n-tuple of values $(v_1, \ldots, v_n)$ such that $f_1(s) = v_1, \ldots, f_n(s) = v_n$.

One can further restrict the set $\mathbf{S}$ by defining a *constraint* that limits the possible sets of n-tuples $(f_1(s), \ldots, f_n(s))$. For example, consider a sequential program with an integer variable u whose scope does not include the entire program. We could define the range of u to consist of the integers together with the special element $\perp$ denoting an undefined value, and then require that the value of u be an integer for certain values of pc and that it equal $\perp$ for the remaining value of pc.

To express this formally, let R_i be the range of f_i. A constraint is a subset C of $R_1 \times \cdots \times R_n$. Given the functions f_i and the set C, $\mathbf{S}$ is effectively defined by the requirements: (i) for every element s of $\mathbf{S}$, there is a unique element $(v_1, \ldots, v_n)$ in C such that $f_i(s) = v_i$; and (ii) for distinct elements s and t of $\mathbf{S}$, the n-tuples $(f_1(s), \ldots, f_n(s))$ and $(f_1(t), \ldots, f_n(t))$ are unequal.

A subset C of $R_1 \times \cdots \times R_n$ is the same as a boolean-valued function on that set, where the function C is defined by letting $C(v_1, \ldots, v_n)$ equals *true* if and only if $(v_1, \ldots, v_n)$ is in the set C. A constraint C is usually expressed as the relation $C(f_1, \ldots, f_n)$ among the state functions f_i. For example, suppose there are three state functions f_1, f_2, and f_3, where the ranges R_1 and R_2 are the set of integers and R_3 is the set $\{\alpha, \beta, \gamma\}$. We write $[f_1 < f_2] \wedge [(f_3 = \gamma) \supset (f_1 = 0)]$ to mean the subset

$$\{(v_1, v_2, v_3) : v_1 < v_2 \wedge (v_3 = \gamma \supset v_1 = 0)\}$$

of $R_1 \times R_2 \times R_3$.

In the three programs considered above, the state functions consisted of the program variables and the program counter. With more complicated language constructs, other state functions may be needed to describe the program state. A program with subroutines requires a state function to record the current value of the "stack". A concurrent program that uses message-sending primitives may need state functions that record the contents of message buffers. In general, the state functions must completely describe the current state of the system, specifying everything that is necessary to continue its execution. For a deterministic system, such as the program of Figure 1, the current state completely determines its future behavior. For a nondeterministic system, such as the concurrent program of Figure 2, the current state determines all possible future behavior.

Observe that in giving the state functions and constraint, we are describing the essential properties of the set $\mathbf{S}$, but we are not specifying any particular representation of $\mathbf{S}$. In mathematical terms, we are defining $\mathbf{S}$ only up to isomorphism. This little mathematical detail is the formal reason why our specification of the program of Figure 1 does not state whether the variable x is stored in a binary or decimal representation. It doesn't matter whether the elements of $\mathbf{S}$ are strings of bits or decimal digits, or even sequences of voltages on flip-flop wires. All that the specification mentions is the value of the state function x, not the structure of the states.

New state functions can be created as combinations of the state functions f_i. For example, if f_1 and f_2 are integer-valued state functions, then we can define a new boolean-valued state function f by letting $f(s) = (f_1(s) < f_2(s) + 3)$, for any s in **S**. There are two ways to view the state function f. We can think of the f_i as *elementary* state functions and f as a *derived* state function, or we can consider f to have the same status as the f_i by adding the condition $f = (f_1 < f_2 + 3)$ to the constraint. Formally, the two views are equivalent. In practice, the first view seems more convenient and will be adopted.

1.3.2 Specifying Behaviors

Having specified the sets **S** and **A**, we must now specify the set $\mathcal{B}$ of behaviors. The set of behaviors is described formally by a collection of axioms. Four kinds of axioms are used: initial axioms, transition axioms, liveness axioms, halting axioms, and completion axioms.

Initial Axioms A *state predicate* is a boolean-valued state function (either derived or elementary). We say that a predicate P is true for a state s if $P(s)$ equals *true*.

An initial axiom is a state predicate. It is true for the behavior (1) if and only if it is true for the initial state s_0. Initial axioms are used to specify the starting state of the system.

Transition Axioms A *relation* R on the set **S** consists of a set of ordered pairs of elements of **S**. We write sRt to denote that (s, t) is in the relation **S**. We say that the relation R is *enabled* in a state s if there exists a state t such that sRt. The relation R is said to be *self-disabling* if, for any states s and t such that sRt: R is not enabled in t.

A *transition axiom* is a pair (α, R) where α is an action in **A** and R is a self-disabling relation on **S**. We write this pair as $\alpha : R$ instead of (α, R). The transition axiom $\alpha : R$ asserts the following for a behavior of the form (1):

> (Safety) For each i: if $\alpha_i = \alpha$, then R is enabled in state s_{i-1}, and either $s_{i-1}Rs_i$ or else $s_{i-1} = s_i$.
>
> (Liveness) If there exist infinitely many values of i such that $\alpha_i = \alpha$, then, for any i, there exists a $j > i$ such that R is not enabled in s_j.

This is the formal description of the kind of transition axioms we wrote for the programs of Figures 1 and 2. Each atomic operation of the programs was described by a separate transition axiom.

The behavior (1) can be thought of as an infinite sequence of transitions $s_{i-1} \stackrel{\alpha_i}{\to} s_i$ such that the final state of each transition equals the initial state of the next transition. A transition axiom describes the transitions that can appear in a behavior. The safety part of a transition axiom $\alpha : R$ asserts how an α transition can change the state. The conjunction of these assertions for all actions α describes all possible ways that the state can change. However, it does not assert that any change must occur. Asserting that something must change is a liveness property. Interesting liveness properties are asserted by special liveness axioms, described below. Being a liveness axiom, the liveness part of a transition axiom is more logically included with the other liveness axioms. However, it expresses a very weak liveness property—namely, that an infinite number of stuttering α transitions cannot occur with R continuously enabled. (Remember that R is self-disabling, so a nonstuttering α action must disable the transition.) We know of no cases in which one does not want at least this liveness property to hold, so it is easiest to include it as part of the transition axiom.

The requirement that R be self-disabling avoids certain formal difficulties, such as the ones pointed out in [3]. In specifying systems, it seems to be a bad idea to allow actions that could repeat themselves infinitely often with no intervening actions—except for a trivial halting action that denotes termination. Thus, this requirement is not a significant restriction.

A relation R on $\mathbf{S}$ is described as a relation on state functions subscripted *new* or *old*. For example, if f and g are state functions, then $f_{new} < g_{old}$ describes the relation R such that sRt is true if and only if $f(t) < g(s)$. In a practical specification language, one needs a convenient notation for expressing relations on state functions. One method is to write the relations using the *new* and *old* subscripts. Another method is to use ordinary programming language constructs. The assignment statement $x := x + y$ describes a relation such that $x_{new} = x_{old} + y_{old}$ and the new and old values of all other variables are the same. The use of *new* and *old* subscripts is more general, since a relation such as $x^2_{new} + y^2_{old} = x_{old}$ cannot be written conveniently as an assignment statement. On the other hand, sometimes the programming notation is more convenient. A specification language should probably allow both notations.

Liveness Axioms Liveness axioms are expressed with temporal logic. The time has come to describe this logic more formally. Fortunately, we need only a very restricted form of temporal logic—a form that is known in the trade as linear-time temporal logic with unary operators. In particular, we do not need the binary "until" operator, which can make formulas hard to understand. Of course, we do not include an "in the next state" operator.

A temporal logic formula represents a boolean function of behaviors. We write $\sigma \models U$ to denote that the formula U is true on the behavior σ. Recall that a state predicate is a boolean-valued function on the set $\mathbf{S}$ of states. An *action predicate* is a boolean-valued function on the set $\mathbf{A}$ of actions. We identify an action α of $\mathbf{A}$ with the action predicate that is true on an action β of $\mathbf{A}$ if and only if $\beta = \alpha$. The generalization of state predicates and action predicates is a *general predicate*, which is a boolean-valued function on the set $(\mathbf{S} \times \mathbf{A})$ of state, action pairs.

A formula of temporal logic is made up of the following building blocks: general predicates; the ordinary logical operators $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), and $\neg$ (negation); and the unary temporal operator $\Box$. To define the meaning of any formula, we define inductively what it means for such a formula to be true for a behavior σ of the form (1).

A general predicate G is interpreted as a temporal logic formula by defining $\sigma \models G$ to be true if and only if $G(s_0, \alpha_1)$ is true. Thus, a state predicate is true of a behavior if and only if it is true for the first state, and an action predicate is true of a behavior if and only if it is true for the first action.

The meaning of the ordinary boolean operators is defined in the obvious way—for example, $\sigma \models U \wedge V$ is true if and only if both $(\sigma \models U)$ and $(\sigma \models V)$ are true, and $\sigma \models \neg U$ is true if and only if $\sigma \models U$ is false.

The operator $\Box$, read "always" or "henceforth", is defined as follows. If σ is the behavior (1), then let σ^{+n} be the behavior

$$s_n \stackrel{\alpha_{n+1}}{\longrightarrow} s_{n+1} \stackrel{\alpha_{n+2}}{\longrightarrow} \cdots$$

for $n \geq 0$. For any formula U, $\sigma \models \Box U$ is defined to be true if and only if $\sigma^{+n} \models \Box U$ is true for all $n \geq 0$. For example, if P is a state predicate, then $\Box P$ is true for σ if and only if P is true for every state s_n in σ.

The derived operator $\Diamond$, read "eventually", is defined by letting $\Diamond U$ equal $\neg\Box\neg U$ for any formula U. Thus, $\sigma \models \Diamond U$ is true if and only if $\sigma^{+n} \models \Diamond U$ is true for some $n \geq 0$. In particular, if P is a state predicate, then $\Diamond P$ is true for σ if and only if P is true on some state s_n in σ.

The derived operator $\leadsto$, read "leads to", is defined by letting $U \leadsto V$ equal $\Box(U \supset \Diamond V)$. Intuitively, $U \leadsto V$ means that whenever U is true, V must be true then or at some later time. Thus, if P and Q are state predicates and σ is the behavior (1), then $P \leadsto Q$ is true for σ if and only if, for every n: if P is true on state s_n then Q is true on s_m for some $m \geq n$.

You should convince yourself that, for a state predicate P, the formula $\Box\Diamond P$ (read "infinitely often P") is true for the behavior (1) if and only if P is true on infinitely many states s_n, and $\Diamond\Box P$ is true if and only if there is some n such that P is true on all states s_m for $m \geq n$. With a little practice, it is easy to understand the type of temporal logic formulas one writes to specify liveness properties.

Most liveness properties are expressed with the $\leadsto$ operator. A typical property asserts that if a transition becomes enabled then it will eventually "fire". This property is expressed by a formula of the form $P \leadsto \neg P$, where P is the state predicate asserting that the transition is enabled. The liveness part of a transition axiom $\alpha : R$ is $\Box\Diamond A \leadsto \neg P$, where A is the action predicate that is true for action α and false for all other actions, and P is the predicate that is true for a state s if and only if R is enabled on s.

Halting and Completion Axioms A *halting axiom* consists of a pair $\alpha : P$ where α is an action in $\mathbf{A}$ and P is a state predicate. The halting axiom $\alpha : P$ is true for the behavior of (1) if the following condition is satisfied:

For every i: if $\alpha_i = \alpha$, then P is true in state s_{i-1} and $s_i = s_{i-1}$.

The γ Transition Axiom for the program of Figure 1 and the θ Transition Axiom for the program of Figure 2 are examples of halting axioms.

A halting axiom is needed to allow the possibility of halting—the absence of any more nonstuttering transitions. It turns out that, when writing descriptions of individual modules rather than of complete programs, one usually does not need a halting axiom.

The completion axiom defines the set $\mathbf{A}$ of actions, asserting what the possible actions α_i are in a behavior of the form (1). In writing specifications, a slightly different form of completion axiom will be used that asserts which actions are performed by which modules.

1.3.3 Completeness of the Method

Implicit in the work of Alpern and Schneider [2] is a proof that any system that can be described using a very powerful formal system for writing temporal axioms (much more powerful than the simple temporal logic defined above) can also be described by initial axioms, transition axioms, simple liveness axioms of the form $P \leadsto Q$ with P and Q state predicates, a halting axiom, and a completion axiom. Our method for describing systems can therefore be used to provide a formal description of any system that we would expect it to.

Of course, theoretically possible does not necessarily mean practical and convenient. The utility of our formal method as the basis for a practical method for specifying and reasoning about concurrent systems must be demonstrated with examples.

1.3.4 Programs as Axioms

It is customary to adopt the point of view that the transition axiom describes the semantics of the atomic operation. From now on, it will be useful to reverse this way of thinking and instead to think of the atomic operation as a convenient way to write the transition axiom. A program then becomes an easy way to write a collection of axioms.

When writing specifications, programs seem to be convenient for expressing the transition axioms but not so convenient for expressing other liveness properties.

1.4 Implementing One System with Another

1.4.1 The Formal Definition

Let $(\mathcal{B}, \mathbf{S}, \mathbf{A})$ and $(\mathcal{B}', \mathbf{S}', \mathbf{A}')$ be two systems. We now formally define what it means for the first to implement the second. We call $(\mathcal{B}, \mathbf{S}, \mathbf{A})$ the *lower-level* system and its state functions, behaviors, etc. are called lower-level objects; $(\mathcal{B}', \mathbf{S}', \mathbf{A}')$ is said to be the *higher-level* system and its state functions, etc. are called higher-level objects.

Recall the approach used in Section 1.2, where the lower-level system was an assembly language program and the higher-level one was a program written in a higher-level language. We defined a mapping F from lower-level behaviors to higher-level behaviors. More precisely, if σ is a behavior with lower-level states and actions, then $F(\sigma)$ is a behavior with higher-level states and actions. Correctness of the implementation meant that for any behavior σ of the lower-level system, $F(\sigma)$ is a behavior of the higher-level one. The mapping F was derived from mappings F_{st} from lower-level states to higher-level states and F_{ac} from lower-level actions to higher-level actions.

We generalize this definition slightly by allowing F_{ac} to be a function of both the action and the state of the lower-level system rather than just of the action. This generalization permits the same lower-level action to implement several different higher-level actions. For example, suppose the compiler translates (higher-level) exponential operations into calls of a single (assembly language) exponentiation subroutine. An atomic operation performed by the exponentiation subroutine can correspond to an execution of one of many different atomic operations of the higher-level program, which one depending upon the value of the register that contains the return address of the subroutine call.

The formal definition states that $(\mathcal{B}, \mathbf{S}, \mathbf{A})$ implements $(\mathcal{B}', \mathbf{S}', \mathbf{A}')$ if there exist mappings $F_{st} : \mathbf{S} \to \mathbf{S}'$ and $F_{ac} : \mathbf{A} \times \mathbf{S} \to \mathbf{A}'$ such that $F(\mathcal{B}) \subset \mathcal{B}'$, where the mapping F is defined by letting $F(\sigma)$, for the behavior σ given by (1), equal

$$F_{st}(s_0) \stackrel{F_{ac}(\alpha_1,s_0)}{\longrightarrow} F_{st}(s_1) \stackrel{F_{ac}(\alpha_2,s_1)}{\longrightarrow} F_{st}(s_2) \stackrel{F_{ac}(\alpha_3,s_2)}{\longrightarrow} \cdots \tag{2}$$

1.4.2 The Definition in Terms of Axioms

The Mappings The set of states is specified by elementary state functions and a constraint. Let $f_1, \ldots, f_n$ be the elementary state functions and C the constraint that define the set $\mathbf{S}$ of lower-level states, and let $f'_1, \ldots, f'_m$, C' be the elementary state functions and constraint defining the set $\mathbf{S}'$ of higher-level states. To define the mapping F_{st}, we must express each higher-level state function f'_j in terms of the lower-level ones f_i. That is, we must choose mappings $F_j : R_1 \times \cdots \times R_n \to R'_j$, where R_i is the range of f_i and R'_j is the range of f'_j, and define f'_j to equal $F_j(f_1, \ldots, f_n)$. The F_j must be constraint-preserving—that is, for any v in $R_1 \times \cdots \times R_n$, if $C(v) = \mathit{true}$ then $C'(F_1(v), \ldots, F_m(v)) = \mathit{true}$.

The mappings F_j define the mapping $F_{st} : \mathbf{S} \to \mathbf{S}'$ as follows. If s is the (unique) element of $\mathbf{S}$ such that $f_i(s) = v_i$, for $i = 1, \ldots, n$, then $F_{st}(s)$ is the element of $\mathbf{S}'$ such that $f'_j(F_{st}(s)) = F_j(v_1, \ldots, v_n)$, for $j = 1, \ldots, m$.

For any higher-level state function f, let $F^*_{st}(f)$ be the lower-level state function such that $F^*_{st}(f)(s)$ is defined to equal $f(F(s))$. It is a simple exercise in unraveling the notation to verify that, for the elementary higher-level state function f'_j, the lower-level state function $F^*_{st}(f'_j)$ is the function $F_j(f_1, \ldots, f_n)$. Thus, for the elementary higher-level state function f'_j, the lower-level state function $F^*_{st}(f'_j)$ is just the definition of f'_j as a function of the lower-level state functions. For a derived higher-level state function f, one can compute $F^*_{st}(f)$ in terms of the functions $F^*_{st}(f'_j)$. For example, $F^*_{st}(3f'_1 + f'_2)$ equals $F^*_{st}(3f'_1) + F^*_{st}(f'_2)$.

These definitions are basically quite simple. Unfortunately, simple ideas can appear complicated when they are expressed in the abstract formalism of n-tuples and mappings. The mapping F_{st} from lower-level states to higher-level ones and the mapping F^*_{st} from higher-level state functions to lower-level ones are the basis for our method of verifying the correctness of an implementation. To fully understand this method, the reader should develop an intuitive understanding of these mappings, and the relation between them. This is best done by expressing the mappings from the example in Section 1.2 in terms of this formalism.

Our specifications talk about state functions rather than states. Hence, it is the mapping F^*_{st} from higher-level state functions to lower-level ones that we must use rather than the mapping F_{st} from lower-level states to higher-level ones.

For any action α, let $A(\alpha)$ be the action predicate such that $A(\alpha)(\xi)$ is true if and only if $\xi = \alpha$. Action predicates of the form $A(\alpha)$ are called *elementary* action predicates; they play the same role for the set of actions that the elementary state functions play for the set of states. Any action predicate can be expressed as a function of elementary action predicates.

Just as the mapping F_{st} is defined by expressing the higher-level elementary state functions in terms of the lower-level ones, the mapping F_{ac} from $\mathbf{A}\times\mathbf{S}$ to $\mathbf{A}'$ is defined by expressing each higher-level elementary action predicate as a function of the lower-level elementary action predicates and state functions. This defines a mapping F^*_{ac} from higher-level action predicates to lower-level general predicates such that $F^*_{ac}(A')(\sigma, s) = A'(F_{ac}(\alpha, s))$ for any higher-level action predicate A. (Recall that a lower-level general predicate is a boolean function on $\mathbf{S} \times \mathbf{A}$.) The formal definitions are analogous to the ones for F_{st} and F^*_{st}, and we won't bother with the details.

The mappings F^*_{st} and F^*_{ac} induce a mapping F^* from higher-level general predicates to lower-level ones, where, for any higher-level general predicate G, $F^*(G)$ is the predicate whose value on (s, α) equals $G(F_{st}(s), F_{ac}(s, \alpha))$. For a high-level state predicate P, $F^*(P)$ is the same as $F^*_{st}(P)$, and, for a high-level action predicate A, $F^*(A)$ is the same as $F^*_{st}(A)$.[1] Any general predicate is represented as a function of elementary state and action predicates, so F^* can be computed from the $F^*_{st}(f'_j)$ and the $F^*_{ac}(A(\alpha))$ for the higher-level elementary state functions f'_j and action predicates $A(\alpha)$. For example, $F^*(A(\alpha) \supset f_1 < f_2)$ is the lower-level general predicate

[1]Formally, this requires identifying the state predicate P with the general predicate $\overline{P}$ such that $\overline{P}(s, \alpha) = P(s)$, and similarly for the action predicate A.

$F^*_{ac}(A(\alpha)) \supset F^*_{st}(f_1) < F^*_{st}(f_2)$.

The mapping F^* is extended to arbitrary temporal formulas in the natural way—for example, for general predicates G and H,

$$F^*(\diamond(\Box G \vee \diamond H)) = \diamond(\Box F^*(G) \vee \diamond F^*(H))$$

Thus, F^* maps higher-level temporal logic formulas into lower-level ones. It is a simple exercise in untangling the notation to verify that, for any lower-level level behavior σ and any higher-level temporal logic formula U: $F(\sigma) \models U$ is true if and only if $\sigma \models F^*(U)$ is.

For later use, we note that F_{st} also induces a mapping F^* from higher-level relations on $\mathbf{S}'$ to lower-level relations on $\mathbf{S}$, where $s\, F^*(R)\, t$ is defined to equal $F_{st}(s)\, R\, F_{st}(t)$. The mapping F^* on relations is easily computed from the mapping F^*_{st} on state functions. For example, if R is the higher-level relation defined by $f_{new} < g_{old}$ for higher-level state functions f and g, then $F^*(R)$ is the lower-level relation defined by $F^*_{st}(f)_{new} < F^*_{st}(g)_{old}$.

Mapping the Axioms Suppose that the sets of behaviors $\mathcal{B}$ and $\mathcal{B}'$ are specified by sets of axioms $\mathcal{U}$ and $\mathcal{U}'$, respectively. Thus, a behavior σ is in $\mathcal{B}$ if and only if $\sigma \models U$ is true for every axiom U in $\mathcal{U}$, and similarly for the behaviors in $\mathcal{B}'$. Recall that the lower-level system implements the higher-level one if and only if, for every behavior σ in $\mathcal{B}$, $F(\sigma)$ is in $\mathcal{B}'$. The behavior $F(\sigma)$ is in $\mathcal{B}'$ if and only if $F(\sigma) \models U$ is true for every axiom U in $\mathcal{U}'$, which is true if and only if $\sigma \models F^*(U)$ is true. Hence, to show that $\sigma \in \mathcal{B}$ implies $F(\sigma) \in \mathcal{B}'$, it is necessary and sufficient to show, for all U in $\mathcal{U}'$, that $\forall\, V \in \mathcal{U} : \sigma \models V$ implies $\sigma \models F^*(U)$, which is the same as showing $\sigma \models (\forall\, V \in \mathcal{U}) \supset F^*(U)$. Proving that $\sigma \models (\forall\, V \in \mathcal{U}) \supset F^*(U)$ for all σ means showing that the axioms of $\mathcal{U}$ imply $F^*(U)$.

Thus, to show that the lower-level system implements the higher-level one, we must show that for every higher-level axiom U, the lower-level temporal logic formula $F^*(U)$ is provable from the lower-level axioms. We now consider what this means for the different kinds of axioms that constitute the formal description of a system.

An initial axiom is simply a state predicate. For each higher-level initial axiom P, we must prove that the state predicate $F^*_{st}(P)$ follows from the lower-level initial axioms.

A higher-level liveness axiom is a temporal logic formula, and for each such axiom U we must prove that $F^*(U)$ is a logical consequence of the lower-level liveness axioms. The liveness part of a transition axiom is also considered to be a liveness axiom, and is handled in this way.

If U is the higher-level completion axiom, then $F^*(U)$ follows immediately from the lower-level completion axiom and the fact that the range of values assumed by F_{ac} is contained in the set $\mathbf{A}'$.

Finally, we consider the conjunction of the halting axioms and the safety part of the transition axioms as a single higher-level axiom U. The axiom U asserts that for any higher-level transition $s' \stackrel{\alpha'}{\rightarrow} t'$:

- If there is a transition axiom $\alpha' : R'$ then R' is enabled in state s' and either $s'R't'$ or $s' = t'$.
- If there is a halting axiom $\alpha' : P'$ then P' is true for state s' and $s' = t'$.

The formula $F^*(U)$ asserts that for any lower-level transition $s \stackrel{\alpha}{\rightarrow} t$:

- If there is a transition axiom $F_{ac}(\alpha, s) : R'$ then R' is enabled in state $F_{st}(s)$ and either $F_{st}(s)\, R'\, F_{st}(t)$ or $F_{st}(s) = F_{st}(t)$.
- If there is a halting axiom $F_{ac}(\alpha, s) : P'$ then P' is true for state $F_{st}(s)$ and $F_{st}(s) = F_{st}(t)$.

The lower-level transition axioms determine the possible lower-level transitions $s \stackrel{\alpha}{\rightarrow} t$. Assume that for each action there is exactly one transition or halting axiom.[2] Then a lower-level transition $s \stackrel{\alpha}{\rightarrow} t$ must satisfy either a transition axiom $\alpha : R$ or a halting axiom $\alpha : P$. It follows from the above characterization of the formula $F^*(U)$ that the lower-level transition and halting axioms imply $F^*(U)$ if and only if the following conditions hold:

- For every lower-level transition axiom $\alpha : R$, if sRt then:
  - If there is a transition axiom $F_{ac}(\alpha, s) : R'$ then R' is enabled in state $F_{st}(s)$ and either $F_{st}(s)\, R'\, F_{st}(t)$ or $F_{st}(s) = F_{st}(t)$.
  - If there is a halting axiom $F_{ac}(\alpha, s) : P'$ then P' is true for state $F_{st}(s)$ and $F_{st}(s) = F_{st}(t)$.
- For every lower-level halting axiom $\alpha : P$, if P is true on a state s then there is a halting axiom $F_{ac}(\alpha, s) : P'$ and P' is true on state $F_{st}(s)$.

[2] It makes no sense to have both a transition and a halting axiom for the same transition, and the conjunction of two transition axioms $\alpha : R$ and $\alpha : R'$ is equivalent to the single transition axiom $\alpha : R \cap R'$.

These conditions imply that for every transition $s \stackrel{\alpha}{\to} t$ satisfying the lower-level transition or halting axiom for α, the transition $F_{st}(s) \stackrel{F_{ac}(\alpha,s)}{\longrightarrow} F_{st}(t)$ satisfies the higher-level transition or halting axiom for $F_{ac}(\alpha, s)$.

The above conditions can be written more compactly in the common case when $F_{ac}(\alpha, s)$ depends only upon the action α. In that case, the conditions can be expressed as follows, where $=$ denotes the equality relation (the set of pairs (s, s)) and enabled(R) is the predicate that is true for state s if and only if R is enabled in s. (Recall that $F^*(R')$ is the relation such that $s\, F^*(R')\, t$ if and only if $F_{st}(s)\, R'\, F_{st}(t)$.)

- For every lower-level transition axiom $\alpha : R$:
  - If there is a transition axiom $F_{ac}(\alpha) : R'$ then enabled$(R) \supset F^*(\text{enabled}(R'))$ and $R \subset F^*(R' \cup =)$.
  - If there is a halting axiom $F_{ac}(\alpha) : P'$ then enabled$(R) \supset F^*(P')$ and $R \subset F^*(=)$.
- For every lower-level halting axiom $\alpha : P$, there is a halting axiom $F_{ac}(\alpha) : P'$ such that $P \supset F^*(P')$.

2 Specification

Thus far, we have been discussing the formal description of a complete system. A prerequisite for a *specification* is the splitting of a system into two parts: the part to be specified, which we call a *module*, and the rest of the system, which we will call the *environment*. A Modula-2 [11] module is an example of something that could qualify as a "module", but it is not the only example. A piece of hardware, such as RAM chip, could also be a module. The purpose of a specification is to describe how the module interacts with the environment, so that (i) the environment can use the module with no further knowledge about how it is implemented, and (ii) the module can be implemented with no further knowledge of how it will be used. (A Modula-2 definition module describes the "syntax" of this interaction; a complete specification must describe the semantics.)

One can regard a Modula-2 module as a module, with the rest of the program as the environment. A specification describes the effects of calling the module's procedures, where these effects can include the setting of exported variables and **var** arguments, calls to other procedures that are part of the environment, and the eventual setting of the "program counter" to the location immediately following the procedure call. The specification describes only the procedure's interaction with the environment, not how this interaction is implemented. For example, it should not rule out the possibility of an implementation that invokes machine-language subroutines, or even special-purpose hardware. Any other requirements—for example, that the procedure be implemented in ASCII standard Pascal, or that it be delivered on 1600 bpi magnetic tape, or that it be written on parchment in green ink—are not part of the specifications that we will write. This omission is not meant to imply that these other requirements are unimportant. Any formal method must restrict itself to some aspect of a system, and we choose to consider only the specification of the interface.

It is reasonably clear what is meant by specifying a Modula-2 module because the boundary between the module and the environment is evident. On the other hand, we have no idea what it would mean to specify the solar system because we do not know what is the module to be specified and what is its environment. One can give a formal description of some aspect of the solar system, such as the ones developed by Ptolemy, Copernicus, and Kepler. The distinction we make between a specification and a formal description appears not to be universally accepted, since a workshop on specifying concurrent systems [4] was devoted to ten "specification" problems, only three of which had moderately clear boundaries between the module to be specified and its environment.

2.1 The Axioms

As we have seen, the complete system is described formally by a triple $(\mathcal{B}, \mathbf{S}, \mathbf{A})$, where $\mathcal{B}$ is a set of behaviors. A behavior is a sequence of transitions $s \stackrel{\alpha}{\to} t$, where α is a transition in $\mathbf{A}$ and s and t are states in $\mathbf{S}$. When specifying a module, we don't know what the complete set of states $\mathbf{S}$ is, nor what the complete set of actions $\mathbf{A}$ is. All we can know about are the part of the state accessed by the module and the subset of actions that are relevant to the module's activities, including those actions performed by the module itself.

A complete system $(\mathcal{B}, \mathbf{S}, \mathbf{A})$ is specified by a collection of axioms. This collection of axioms should be partitioned into two subcollections: ones that specify the module and ones that specify the rest of the system. Our task is to write the axioms that specify the module without making any unnecessary restrictions on the behavior of the rest of the system.

A major purpose of labeling the arc in a transition $s \stackrel{\alpha}{\to} t$ is that it allows us to identify whether the transition is performed by the module or the environment. The operation of the module is specified by axioms about transitions performed by the module—usually describing how they change the state. The specification of the module must also include axioms about transitions performed by the environment, since no module can work properly in the face of completely arbitrary behavior by the environment. (Imagine trying to write a procedure that works correctly even though concurrently operating processes are randomly modifying the procedure's local variables.) Axioms about the environment's transitions usually specify what the environment cannot do—for example, that it cannot change parts of the state that are local to the module.

We will partition a module's specification into axioms that constrain the module's behavior and ones that constrain the environment. This will be done by talking about transitions—axioms that describe the module's transitions constrain the module's behavior, and ones that describe the environment's transitions constrain the environment's behavior. However, this is not as easy as it sounds. The implications of axioms are not always obvious, and axioms that appear to specify the module may actually constrain the behavior of the environment. For example, consider a specification of a Modula-2 procedure that returns the square root of its argument, the result and argument being of type **real**. Neglecting problems of round-off error, we might specify this procedure by requiring that, if it is called with an argument x, then it will eventually return a value y such that $y^2 = x$. Such a specification contains only axioms specifying the module, and no axioms specifying the environment. However, observe that the specification implies that if the procedure is called with an argument of -4, then it must return a real value y such that $y^2 = -4$. This is impossible. The axioms specifying the module therefore constrain the environment never to call the procedure with a negative argument.

In general, for an axiom that specifies a safety property, it is possible to determine if an axiom constrains the module, the environment, or both. However, it appears to be impossible to do this for a liveness axiom.

In practice, one specifies safety properties by transition axioms, and it is easy to see if a transition axiom constrains the module or the environment; a transition axiom for a module action constrains the module and one for an environment action constrains the environment. Liveness properties are more subtle. A liveness property is specified by an axiom asserting that, under certain conditions, a particular transition must eventually occur. For example, when the subroutine has been called, a "return" action must eventually occur. We view such an axiom as constraining the module in question if the transition is performed by the module, and as constraining the environment if it is performed by the environment. However, we must realize that a liveness axiom could have nonobvious implications. In the above example, a simple liveness property of the module (that it must return an answer) implies a safety property of the environment (that it may not call the module with a negative argument).

2.2 The Interface

The mechanism by which the module and the environment communicate is called the *interface*. The specification should describe everything that the environment needs to know in order to use the module, which implies that the interface must be specified at the implementation level. A procedure to compute a square root will not function properly if it expects its argument to be represented as a double-word binary floating point number and it is called with an argument represented as a string of ASCII characters.

The need to specify the interface at the implementation level is not restricted to the relatively minor problems of data representation. See [8] for an example indicating how the interface's implementation details can influence the specification of fundamental properties of concurrent systems.

In practice, specifying the interface at the implementation level is not a problem. When writing a specification, one generally knows if the implementation is going to be in Modula 2, Ada, or CMOS. One can then specify the interface as, for example, a collection of procedure calls with arguments of a certain type. For a Modula 2 module, the definition module will usually provide the interface specification. (Unfortunately, it is unlikely that the semantics of any existing concurrent programming language are specified precisely enough to insure that specifications are always independent of the particular compiler.)

We shall see that the specification can be decomposed into two parts: the *interface* specification and the *internal* specification. The interface specification is implementation dependent. In principle, the internal specification is independent of the implementation. However, details of the interface are likely to manifest themselves in the internal specification as well. For example, what a procedure does when called with incorrect input may depend upon whether or not the language provides an exception-handling mechanism.

The need to specify the interface at the implementation level was recognized by Guttag and Horning in the design of Larch [5]. What they call the language-dependent part of the specification corresponds to the interface specification. The language-independent part of a Larch specification includes some aspects of our internal specification. However, to handle concurrency, we need to describe the behavior of the module during a procedure call, a concept not present in a Larch specification, which describes only input/output relations.

2.3 State Functions

2.3.1 The Module's State Functions

To describe the set $\mathbf{S}$ of states of the complete program, we give a collection of state functions $f_1, \ldots, f_n$, and a constraint C, and assert that $\mathbf{S}$ is determined by the values of these functions—for every n-tuple $(v_1, \ldots, v_n)$ of values that satisfies the constraint C, there is a *unique* element s of $\mathbf{S}$ such that, for all i, $f_i(s) = v_i$.

When specifying a module, we do not know the complete state because we know very little about the environment. We can only know about the part of $\mathbf{S}$ that is relevant to the module. Fortunately, this causes no difficulty. To specify a module, we specify n state functions f_i and constraint C that describe the relevant part of the state, and we drop the requirement that the n-tuple of values $f_i(s)$ uniquely determines the state s. There can be many states s that have the same values $f_i(s)$, but have different values of $g(s)$ for some state function g that is relevant only to the environment. For example, if our module is a Modula-2 module, g could represent the value of a variable local to some separate module.

2.3.2 Interface and Internal State Functions

The decomposition of the system into environment and module implies that there are two different types of state functions: *interface* state functions and *internal* state functions. Interface state functions are part of the interface. They are externally visible, and are specified at the implementation level. To explain why they are needed, we briefly discuss the nature of communication.

Synchronous processes can communicate through transient phenomena; if you are listening to me, waiting for me to say something, I can communicate by sound waves, which are a transient disturbance in the atmosphere. However, if we are not synchronized in this way, and I don't know whether or not you are listening to me, we cannot communicate in this way. In the asynchronous case, I have to make some nontransient state change—for example, writing a message on your blackboard or magnetizing the surface of the tape in your answering machine. You can receive my communication by examining the state of your blackboard or answering machine. Communication is effected with a nontransient change to a communication medium. In computer systems, we sometimes pretend that asynchronous processes communicate by transient events such as sending a message. However, a closer examination reveals that the transient event actually institutes a nontransient state change to a communication medium—for example, by putting a message in a buffer. Communication is achieved through the use of this medium. We specify the communication between the environment and the module in terms of the state of the communication medium.

The interface state functions represent the communication medium by which the environment and the module communicate. In the specification of a hardware component, the interface state functions might include the voltage levels on certain wires. In the specification of a procedure, the interface state functions might include parameter-passing mechanisms. For example, immediately after the environment executes a call to this procedure, there must be some state component that records the fact that the procedure has just been called and the argument with which it was called. The state functions that provide this information are part of the interface specification. For a Modula-2 module, the interface functions are implicitly specified by the definition module. Fortunately, there is usually no need explicitly to describe those interface state functions in detail.

Interface state functions can be directly observed or modified by the environment; the environment can read the voltage on the wires leading to the hardware device, or set the value of the state function that describes the argument with which a procedure is called. Internal state functions are not directly observable by the environment. Their values can only be inferred indirectly, by observing the external behavior of the module as indicated by the values of its interface state functions. For example, consider the specification of a Modula-2 module that implements a queue, having a procedure that adds an element to the end of the queue and one that removes an element from the head of the queue. Its specification will include, as an internal state function, the value of the queue—that is, the sequence of elements comprising the current contents of the queue. This state function is probably not directly observable; the queue is probably implemented by variables that are local to the module and not visible externally. One can only infer the contents of the queue by the module's response to a sequence of procedure calls.

2.3.3 Aliasing and Orthogonality

In a program, we usually assume that distinct variables represent disjoint data objects—that is, we assume the absence of *aliasing*. Given a complete program, without pointers and dereferencing operations, aliasing can be handled by explicitly determining which variable is aliased to what. However, with the introduction of pointers or, equivalently, of procedures with "**var**" parameters, aliasing is no longer such a simple matter. Two procedure parameters with different names may, in a particular call of that procedure, represent the identical data object.

In programming languages, aliasing manifests itself most clearly in an assignment statement—x and y are aliased if assigning a value to x can change the value of y. The usual case is the absence of aliasing, which we call *orthogonality*. In an ordinary programming language, two variables x and y are said to be orthogonal to one another if assigning to one of them does not change the value of the other. The concepts of aliasing and orthogonality must be precisely defined in any specification language. Any method for specifying a transition must describe both what state functions change and what state functions do not change. Specifying what state functions change, and how they change, is conceptually simple. For example, we can write the assignment statement $x := x + y$ to specify that the value of x changes such that $x_{new} = x_{old} + y_{old}$. However, describing what state functions don't change is more difficult. Implicit in the assignment statement $x := x + y$ is the assumption that state functions orthogonal to x are not changed. However, a precise definition of orthogonality is difficult—especially when the state includes pointers and transient objects. A complete discussion of the problem of aliasing is beyond the scope of this paper. See [9] for an introduction to aliasing and orthogonality in sequential programs.

Just as with program variables, different state functions are usually orthogonal. The constraint determines any "aliasing" relations that may exist between state functions in the same module. A constraint such as $f > g$ may be regarded as a general form of aliasing, since changing the value of f might necessitate a change to g to maintain this constraint. Aliasing relations between state functions from different modules are what makes intermodule communication possible. If two modules communicate through the value of a voltage on a wire, then that value is an interface state function in each of the modules, those two state functions being aliases of one another. As another example, suppose a procedure in a module A calls a procedure in another module B. The interface state function of module A that represents the argument with which A calls B's procedure is aliased to the state function of module B that represents the value of the argument parameter.

Internal state functions of one module are assumed to be orthogonal to internal state functions of any other module and to any interface state functions, including the ones of the same module. By their nature, internal state functions are not directly accessible from the environment, so they cannot be aliased to state functions belonging to or accessible from the environment.

2.4 Axioms

2.4.1 Concepts

Recall that, to specify a complete system, we had the following classes of axioms: initial axioms, transition axioms, liveness axioms, halting axioms and completion axioms. In order to specify a module, which is only part of the complete system, some modifications to this approach are needed.

No change is needed to the way we write initial axioms and liveness axioms. Of course, when specifying liveness properties, we must remember that the module is not executing in isolation. The discussion of the fairness axioms in Section 1.1.5 indicates the type of considerations that this involves.

A halting axiom does not seem to be necessary. The module halts by performing no further transitions; halting transitions can be provided by the environment.

Recall that, for a complete system, the completion axiom specified the set of all actions. In specifying a module, we obviously do not know what the set of *all* actions is; we can specify only what the module's actions are. Thus, the completion axiom asserts that every action of the module is an element of some set $\mathbf{A}_m$ of actions.

Fundamental to splitting the system into module and environment is the ability to distinguish the module's actions from the environment's actions. For example, when presented with a machine-language implementation of a program containing a Modula-2 module, we must be able to determine which machine-language statement executions belong to the module, and which to its environment (the rest of the program). To understand why this is important, consider a specification of a queue, where the interface contains two procedures: *put* to insert an element at the end of the queue and *get* to fetch the element at the front of the queue. An important part of the specification that is often overlooked is the requirement that the *put* and *get* procedures be called by the environment, not by the module. Without this requirement, a "correct" implementation could arbitrarily insert and delete elements from the queue by calling the *put* and *get* procedures itself.

The specification must specify actions performed by the environment as well as those performed by the module. The environment actions that must be specified are the ones by which the environment changes interface functions—for example, the action of calling a procedure in a Modula-2 module. An external action in the specification of a module will be a module action in the specification of some other module.

Environment actions should not change internal state functions. Indeed, allowing the environment to change an internal state function would effectively make that state function part of the interface.

2.4.2 Notation

A specification language requires some convenient notation for writing axioms. As we have seen, the subtle issue is the specification of the relation R of a transition axiom (α, R). For a complete system, we could write R as a simple relation on n-tuples of state function values. This doesn't work specifying a module in a larger system because we don't know what all the state functions are. Therefore, we must write the relation R in two parts: one specifying what state functions it can change, and the other specifying how it can change those state functions. The first part is specified by simply listing the state functions; the second is specified by writing a relation between the old and new values of state functions. One could write a transition axiom in the following fashion:

module transition α **changes only** f, g:
$$(f_{old} = 1) \wedge (f_{new} = g_{old}) \wedge (g_{new} = h + g_{old})$$

This axiom specifies that an α transition, which is a transition performed by the module rather than the environment, is enabled for a state s if and only if $f(s) = 1$, and a nonstuttering α transitions sets the new value of f to the old value of g, sets the new value of g to its old value plus the value of h, and changes no other state functions. Since α does not change h, the new and old values of h are the same, so no subscript is needed. As before, this axiom also includes a liveness part that asserts that there cannot be an infinite number of α transitions without the transition becoming disabled (f assuming a value different from 1).

We could also adopt a more programming language style of notation and write this transition axiom as follows:

$$
\begin{aligned}
\alpha\colon\ \langle\ & f = 1 \rightarrow f := g; \\
& g := h + g \\
& \rangle
\end{aligned}
$$

This assumes a convention that only state functions appearing on the left-hand side of an assignment statement may be changed.

To be more precise, the above transition axioms do not say that an α transition changes only f and g; it could also change the value of state functions aliased to f or g. What it actually asserts is that any state function orthogonal to both f and g is left unchanged by an α transition.

It is often convenient to use parametrized transition axioms, such as:

module transition $\alpha(x : \textbf{integer})$ **changes only** f, g:

$$(f_{old} = x) \wedge (f_{new} = g_{old}) \wedge (g_{new} = h + g_{old})$$

Formally, this specifies a set of transition axioms—one for each (integer) value of x. It is simply a way of saving us from having to write an infinite number of separate axioms. Thus, $\alpha(-5)$ and $\alpha(7)$ are two completely different actions; they are not "invocations" of any single entity α.

For any state function f, the transition axioms for the complete system describe what transitions can change f. If f is an internal state function, then we know that only the module's transitions can change its value. However, an interface state function can be changed by actions of the environment. We need some way to constrain how the environment can change an interface state function. We could use transition axioms to specify all possible ways that the environment can modify an interface state function. However, it is more convenient to write the following kind of axiom:

f **changed only by** $\alpha_1, \ldots, \alpha_m$ **while** P

where f is a state function, the α_i are transitions, and P is a state predicate. This asserts that if a behavior includes a transition $s \xrightarrow{\alpha} t$ for which $f(s) \neq f(t)$, then either $P(s)$ is false or else α is one of the α_i. Of course, one can find syntactic "sugarings" of this type of axiom, such as omitting the **while** clause when P is identically true, thereby asserting that f can be changed only by the indicated transitions and by no others.

A **changed only by** axiom is needed for every interface state function; without one, the environment would be allowed to change the function at any time in any way. Since internal state functions can be changed only by the module's transitions, they do not need a **changed only by** axiom—the module's transition axioms explicitly state what state functions they can change. However, it would probably be a good idea to include one for redundancy.

2.4.3 Formal Interpretation

In Section 1.3, we defined a system to be a triple $(\mathcal{B}, \mathbf{S}, \mathbf{A})$, where $\mathbf{S}$ is a set of states, $\mathbf{A}$ is a set of actions, and $\mathcal{B}$ is a set of behaviors. The set of states was specified by giving the ranges of the state functions $f_1, \ldots, f_n$ and a constraint C that they satisfy. For simplicity, we will drop the constraint C from here on, assuming that it is the trivial constraint that is satisfied by all n-tuples. (It is a simple matter to add the constraint to the formalism.) By requiring that the n values $f_1(s), \ldots, f_n(s)$ uniquely determine s among all the elements of $\mathbf{S}$, we determined $\mathbf{S}$ up to isomorphism. Here, we drop that requirement, so specifying the ranges of the state functions f_i still leaves a great deal of freedom in the choice of the set $\mathbf{S}$.

Instead of a single collection of state functions, we now have two kinds of state functions: the internal state functions, which we denote $f_1, \ldots, f_n$, and interface state functions, which we denote $h_1, \ldots, h_p$. Similarly, there are two types of actions: internal actions $\alpha_1, \ldots, \alpha_s$ and interface actions $\gamma_1, \ldots, \gamma_t$.[3] The specification consists of initial axioms, transition axioms, etc., which can all be expressed as temporal logic formulas. Let $\mathcal{U}$ denote the temporal logic formula that is the conjunction of all these axioms. Then the free variables of this formula are the f_i, h_i, α_i, and γ_i. Formally, the specification is the formula[4]

$$\exists f_1 \ldots f_n \alpha_1 \ldots \alpha_s : \mathcal{U}$$

Thus, the interface state functions h_i and actions γ_i are the free variables of the specification, while the internal state functions and actions are quantified existentially. As we shall see, this is the mathematical expression of the fact that one is free to implement the internal state functions and actions as one chooses, but the interface state functions and actions are given.

[3] For notational convenience, we are assuming that there are a finite number of state functions and actions; however, there could be infinitely many of them.

[4] The formula $\exists x_1 \ldots x_q : X$ is an abbreviation for $\exists x_1 : \exists x_2 : \ldots \exists x_n : X$.

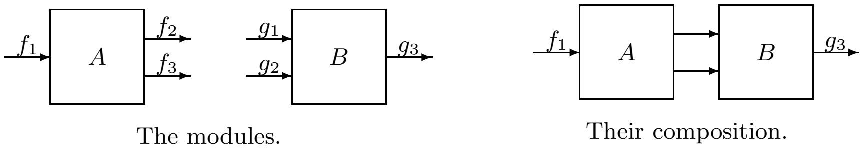

Figure 4: Two hardware modules and their composition.

What does it mean for a system $(\mathcal{B}, \mathbf{S}, \mathbf{A})$ to satisfy this formula? Since $\mathcal{U}$ has the h_i and γ_i as free variables, these state functions and action predicates must be defined on $\mathbf{S}$ and $\mathbf{A}$, respectively. If that is the case, then it makes sense to ask if the temporal logic formula $\mathcal{U}$ is true for a behavior in $\mathcal{B}$. We therefore say that $(\mathcal{B}, \mathbf{S}, \mathbf{A})$ satisfies this formula if and only if the state functions h_i are defined on $\mathbf{S}$, the γ_i are elements of $\mathbf{A}$, and the formula is true for every behavior in $\mathcal{B}$.

There is one problem with this definition: we have not yet defined the semantics of the temporal logic formula $\exists x : U$ when x is a state function or action predicate. We give the definition for x a state function; the definition for action predicates is similar. Let σ be the behavior $s_0 \stackrel{\alpha_1}{\rightarrow} s_1 \stackrel{\alpha_2}{\rightarrow} \ldots$ and let σ' be the behavior $s'_0 \stackrel{\alpha_1}{\rightarrow} s'_1 \stackrel{\alpha_2}{\rightarrow}$. We say that σ and σ' are equivalent except for x if, for every i and every state function f that is orthogonal to x, $f(s_i) = f(s'_i)$. We define a *stuttering* behavior of σ to be a behavior obtained from σ by replacing each transition $s_{i-1} \stackrel{\alpha_i}{\rightarrow} s_i$ by a nonempty finite sequence of transitions of the form $s_{i-1} \stackrel{\alpha_i}{\rightarrow} s_{i-1} \stackrel{\alpha_i}{\rightarrow} \ldots \stackrel{\alpha_i}{\rightarrow} s_{i-1} \stackrel{\alpha_i}{\rightarrow} s_i$. We then define $\sigma \models \exists x : U$ to be true if and only if there exists a behavior σ' that is equivalent except for x to some stuttering behavior of σ such that $\sigma' \models x : U$ is true. In other words, $\exists x : U$ is true on a behavior if and only if we can make U true by adding stuttering actions and arbitrarily changing the value of x on each state.[5]

2.5 The Composition of Modules

One builds a system by combining (composing) modules. As an example, consider two hardware modules, A and B, and their composition, illustrated in Figure 4. The specification of module A has three interface state functions, f_1, f_2, and f_3, whose values represent the voltage levels on the indicated wires. Similarly, module B's specification has the interface state functions g_1, g_2, and g_3.

[5]I wish to thank Amir Pnueli for pointing out this definition to me.

Connecting the two modules as shown in the figure means identifying the state functions f_2 and f_3 of A's specification with the state functions g_1 and g_2 of B's specification—that is, declaring $f_2 \equiv g_1$ and $f_3 \equiv g_2$. Suppose the specification of A includes a single transition α of A that can change f_2 and f_3. The specification of B might include a transition axiom for a transition β of B's environment describing how g_1 and g_2 are allowed to change. (The fact that A changes f_2 and f_3 while the environment changes g_1 and g_2 means that f_2 and f_3 are outputs of A while g_1 and g_2 are inputs to B.)

The formal specification of the composition of the two modules is the conjunction of their specifications—that is, the conjunction of the axioms that make up their specifications—conjoined with the aliasing relations $f_2 \equiv g_1$ and $f_3 \equiv g_2$.

2.6 The Correctness of an Implementation

In Section 1.4 we defined what it meant for one complete system $(\mathcal{B}, \mathbf{S}, \mathbf{A})$ to implement another complete system $(\mathcal{B}', \mathbf{S}', \mathbf{A}')$. The definition was based upon mappings $F_{st} : \mathbf{S} \to \mathbf{S}'$ and $F_{ac} : \mathbf{A} \times \mathbf{S} \to \mathbf{A}'$. From these mappings, we defined mappings F^*_{st} from state functions on $\mathbf{S}'$ to state functions on $\mathbf{S}$ and F^*_{ac} from action predicates on $\mathbf{A}'$ to functions on $\mathbf{A} \times \mathbf{S}$.

If we examine how all these mappings are actually defined for a real example, we discover that it is the mappings F^*_{st} and F^*_{ac} that are really being defined. This is because we don't know what the actual states are, just the state functions. Let $f_1, \ldots, f_n$ be the state functions of the first system and $f'_1, \ldots, f'_m$ be the state functions of the second system. To define the mapping F_{ac}, we must express the values of each f'_i in terms of the f_j, which means defining the state function $F^*_{st}(f'_j)$.

The mappings F^*_{st} and F^*_{ac} in turn define a mapping F^* from temporal logic formulas about behaviors in $\mathcal{B}'$ to temporal logic formulas about behaviors in $\mathcal{B}$. If the sets of behaviors $\mathcal{B}$ and $\mathcal{B}'$ are defined by the axioms $\mathcal{U}$ and $\mathcal{U}'$, respectively, then $(\mathcal{B}, \mathbf{S}, \mathbf{A})$ correctly implements $(\mathcal{B}', \mathbf{S}', \mathbf{A}')$ if and only if $\mathcal{U}$ implies $F^*(\mathcal{U}')$.

Now let us turn to the question of what it means for a specification of one or more modules to implement a specification of another module. As we have seen, the formal specification of a module or collection of modules is a formula of the form

$$\exists f_1 \ldots f_n \alpha_1 \ldots \alpha_s : \mathcal{U} \tag{3}$$

where the f_i and α_i are the internal state functions and actions, and $\mathcal{U}$ is a temporal logic formula that depends upon the f_i and α_i as well as on interface state functions h_i and actions γ_i.

Let $\mathcal{M}$ be the formula (3) and let $\mathcal{M}'$ be the similar formula

$$\exists f'_1 \ldots f'_m \alpha'_1 \ldots \alpha'_r : \mathcal{U}' \tag{4}$$

so $\mathcal{M}'$ specifies a module with internal state functions f'_j and internal actions α'_j. There are two characterizations of what it means for the module specified by $\mathcal{M}$ to implement the module specified by $\mathcal{M}'$:

1. Every system $(\mathcal{B}, \mathbf{S}, \mathbf{A})$ that satisfies the formula $\mathcal{M}$ also satisfies the formula $\mathcal{M}'$.
2. $\mathcal{M}$ implies $\mathcal{M}'$.

Since one temporal logic formula implies another if and only if every behavior that satisfies the first also satisfies the second, it is easy to see that these two characterizations are equivalent.

To prove that the module specified by $\mathcal{M}$ implements the module specified by $\mathcal{M}'$, we must prove that $\mathcal{M}$ implies $\mathcal{M}'$. Recall that these two formulas are given by (3) and (4). To prove that $\mathcal{M}$ implies $\mathcal{M}'$, it suffices to construct mappings F^*_{st} and F^*_{ac}, which define the mapping F^*, as above so that $\mathcal{U}$ implies $F^*(\mathcal{U}')$. This is exactly the same procedure used in Section 1.4 to prove that one system implement another. The only difference is that, in addition to the internal state functions and actions f_i, α_i, f'_i, α'_i, the axioms also involve the interface state functions and actions. (Another way of viewing this is to say that F^*_{st} and F^*_{ac} are defined so they map each interface state function h_i and interface action γ_i into itself.) Thus, the method of actually verifying that one module implements another is the same one used to show that one concurrent program implements another.

As we observed earlier, it is not always possible to define the f'_j as functions of the f_i. In this case, it is necessary to add "dummy" internal state functions g_1, …, g_d to the system specified by $\mathcal{M}$, and to define the f'_j as functions of the f_i and the g_i. For notational convenience, suppose that one introduces a single dummy state function g. Let $\mathcal{M}_d$ be the temporal logic formula that represents the new specification. We want to prove that the new specification $\mathcal{M}_d$ correctly implements $\mathcal{M}'$ and infer from that $\mathcal{M}$ correctly implements $\mathcal{M}'$. This means that we must prove that $\mathcal{M}$ implies $\mathcal{M}_d$. If $\mathcal{M}$ is the formula (3), then $\mathcal{M}_d$ is the formula

$$\exists g : \exists f_1 \ldots f_n \alpha_1 \ldots \alpha_s : \mathcal{U}_d$$

Assume that the dummy state function g is orthogonal to all other state functions, both internal and external. (The specification language will have some notation, analogous to variable declarations in programming languages, for introducing a new state functions that is orthogonal to all other state functions.) To prove that $\mathcal{M}$ implies $\mathcal{M}_d$, we must show that $\mathcal{U}_d$ must has the following property: if σ is any behavior that satisfies $\mathcal{U}$, then there is a behavior σ' that is equivalent except for g to some stuttering behavior of σ such that σ' satisfies $\mathcal{U}_d$.

The condition for the specification with the dummy state function to be equivalent to the original specification is stated in terms of the semantics of the temporal logic—whether or not a behavior satisfies an axiom—rather than within the logic itself. One wants syntactic rules for adding dummy state functions that ensure that this condition is satisfied. the specification. These rules will depend upon the particular specification language; they will correspond to the rules given by Owicki [10] for a simple programming language.

References

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

[2] Bowen Alpern and Fred B. Schneider. *Verifying Temporal Properties without using Temporal Logic*. Technical Report TR85-723, Department of Computer Science, Cornell University, December 1985.

[3] Howard Barringer, Ruurd Kuiper, and Amir Pnueli. A really abstract concurrent model and its temporal logic. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 173–183, ACM, January 1986.

[4] B. T. Denvir, W. T. Harwood, M. I. Jackson, and M. J. Wray, editors. *The Analysis of Concurrent Systems.* Volume 207 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1985.

[5] J. V. Guttag, J. J. Horning, and J. M. Wing. *Larch in Five Easy Pieces.* Technical Report 5, Digital Equipment Corporation Systems Research Center, July 1985.

[6] Leslie Lamport. *An Axiomatic Semantics of Concurrent Programming Languages*, pages 77–122. Springer-Verlag, Berlin, 1985.

[7] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[8] Leslie Lamport. What it means for a concurrent program to satisfy a specification: why no one has specified priority. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 78–83, ACM SIGACT-SIGPLAN, New Orleans, January 1985.

[9] Leslie Lamport and Fred B. Schneider. Constraints: a uniform approach to aliasing and typing. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, ACM SIGACT-SIGPLAN, New Orleans, January 1985.

[10] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, August 1975.

[11] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, third edition, 1985.