A Simple Approach to Specifying Concurrent Systems

Leslie Lamport

December 25, 1986
revised January 26, 1988
corrected August 11, 1988

Author's Abstract

In the transition axiom method, safety properties of a concurrent system can be specified by programs; liveness properties are specified by assertions in a simple temporal logic. The method is described with some simple examples, and its logical foundation is informally explored through a careful examination of what it means to implement a specification. Language issues and other practical details are largely ignored.

Capsule Review

Concurrent systems are notoriously difficult to design and to specify. The *transition axiom* method provides a simple, sound, and powerful formal basis for the specification of such systems. The method is not as widely understood as it should be, perhaps because previous discussions obscured its fundamental principles in the mass of details needed to rigorously describe its formal basis.

This report focusses on the big picture and subordinates the details. It might well be subtitled "Everything you wanted to know about transition axioms, but were afraid to ask." It takes the form of a dialog between the author and an unusually astute student of the method. The questions are those that the reader might well ask, arranged to permit an orderly exposition of the method. The examples are easily understood, yet rich enough to illustrate all the major points.

Transition axioms can be used to specify safety and liveness properties of systems separately; only the latter require explicit use of temporal logic. Reasoning about safety can be done without concern for liveness, but reasoning about liveness will involve both safety and liveness properties.

The method includes a precise and general characterization of what it means to *implement* a specification, and provides formal methods for verifying that one specification implements another. These methods apply even if the state functions and grain of atomicity differ in the two specifications. Since programs are included as a special kind of specification, program verification becomes an important special case. Informal reasoning can be justified by appeal to the underlying formal methods.

The transition axiom method does not presuppose any particular specification language. One of its charms is that semantics for existing languages and notations can be given in terms of transition axioms. Another important aspect is the separation of transition axioms from interface specifications. The same transition axioms–combined with suitable specifications of interface state functions–might specify the behavior of an interface that is mechanical, electronic, or software.

Jim Horning

Contents

1 Introduction

Over the past few years, I have developed an approach to the formal specification of concurrent systems that I now call the *transition axiom* method. The basic formalism has already been described in [13] and [1], but the formal details tend to obscure the important concepts. Here, I attempt to explain these concepts without discussing the details of the underlying formalism.

Concurrent systems are not easy to specify. Even a simple system can be subtle, and it is often hard to find the appropriate abstractions that make it understandable. Specifying a complex system is a formidable engineering task. We can understand complex structures only if they are composed of simple parts, so a method for specifying complex systems must have a simple conceptual basis. I will try to demonstrate that the transition axiom method provides such a basis. However, I will not address the engineering problems associated with specifying real systems. Instead, the concepts will be illustrated with a series of toy examples that are not meant to be taken seriously as real specifications.

Are you proposing a specification language?

No. The transition axiom method provides a conceptual and logical foundation for writing formal specifications; it is not a specification language. The method determines *what* a specification must say; a language determines in detail *how* it is said.

What do you mean by a formal specification?

I find it helpful to view a specification as a contract between the user of a system and its implementer. The contract should tell the user everything he must know to use the system, and it should tell the implementer everything he must know about the system to implement it. In principle, once this contract has been agreed upon, the user and the implementer have no need for further communication. (This view describes the *function* of the specification; it is not meant as a paradigm for how systems should be built.)

For a specification to be formal, the question of whether an implementation satisfies the specification must be reducible to the question of whether an assertion is provable in some mathematical system. To demonstrate that he has met the terms of the contract, the implementer should resort to logic rather than contract law. This does not mean that an implementation must be accompanied by a mathematical proof. It does mean that it should be possible, in principle though not necessarily in practice, to provide such a proof for a correct implementation. The existence of a formal basis for the specification method is the only way I know to guarantee that specifications are unambiguous.

Ultimately, the systems we specify are physical objects, and mathematics cannot prove physical properties. We can prove properties only of a mathematical model of the system; whether or not the system correctly implements the model must remain a question of law and not of mathematics.

Just what is a system?

By "system", I mean anything that interacts with its environment in a discrete (digital) fashion across a well-defined boundary. An airline reservation system is such a system, where the boundary might be drawn between the agents using the system, who are part of the environment, and the terminals, which are part of the system. A Pascal procedure is a system whose environment is the rest of the program, with which it interacts by responding to procedure calls and accessing global variables. Thus, the system being specified may be just one component of a larger system.

The solar system is not a system in this sense, both because it is not discrete and because there is no well-defined notion of an environment with which it interacts.

A real system has many properties, ranging from its response time to the color of the cabinet. No formal method can specify all of these properties. Which ones can be specified with the transition axiom method?

The transition axiom method specifies the behavior of a system—that is, the sequence of observable actions it performs when interacting with the environment. More precisely, it specifies two classes of behavioral properties: safety and liveness properties. Safety properties assert what the system is allowed to do, or equivalently, what it may not do. Partial correctness is an example of a safety property, asserting that a program may not generate an incorrect answer. Liveness properties assert what the system must do. Termination is an example of a liveness property, asserting that a program must eventually generate an answer. (Alpern and Schneider [2] have formally defined these two classes of properties.) In the transition axiom method, safety and liveness properties are specified separately.

There are important behavioral properties that cannot be specified by the transition axiom method; these include average response time and probability of failure. A transition axiom specification can provide a formal model with which to analyze such properties,[1] but it cannot formally specify them.

There are also important nonbehavioral properties of systems that one might want to specify, such as storage requirements and the color of the cabinet. These lie completely outside the realm of the method.

Why specify safety and liveness properties separately?

There is a single formalism that underlies a transition axiom specification, so there is no formal separation between the specification of safety and liveness properties. However, experience indicates that different methods are used to reason about the two kinds of properties and it is convenient in practice to separate them. I consider the ability to decompose a specification into liveness and safety properties to be one of the advantages of the method. (One must prove safety properties in order to verify liveness properties, but this is a process of decomposing the proof into smaller lemmas.)

Can the method specify real-time behavior?

Worst-case behavior can be specified, since the requirement that the system must respond within a certain length of time can be expressed as a safety property—namely, that the clock is not allowed to reach a certain value without the system having responded. Average response time cannot be expressed as a safety or liveness property.

The transition axiom method can assert that some action either must occur (liveness) or must not occur (safety). Can it also assert that it is possible for the action to occur?

No. A specification serves as a contractual constraint on the behavior of the system. An assertion that the system may or may not do something provides no constraint and therefore serves no function as part of the formal specification. Specification methods that include such assertions generally use them as poor substitutes for liveness properties. Some methods cannot specify that a certain input *must* result in a certain response, specifying instead that it is possible for the input to be followed by the response. Every specification I have encountered that used such assertions was improved

[1] See [20] for an example of failure analysis applied to a specification.

by replacing the possibility assertions with liveness properties that more accurately expressed the system's informal requirements.

Imprecise wording can make it appear that a specification contains a possibility assertion when it really doesn't. For example, one sometimes states that it must be possible for a transmission line to lose messages. However, the specification does not require that the loss of messages be possible, since this would prohibit an implementation that guaranteed no messages were lost. The specification might require that something happens (a liveness property) or doesn't happen (a safety property) despite the loss of messages. Or, the statement that messages may be lost might simply be a comment about the specification, observing that it does not require that all messages be delivered, and not part of the actual specification.

If a safety property asserts that some action cannot happen, doesn't its negation assert that the action is possible?

In a formal system, one must distinguish the logical formula A from the assertion $\vdash A$, which means that A is provable in the logic; $\vdash A$ is not a formula of the logic itself. In the logic underlying the transition axiom method, if A represents a safety property asserting that some action is impossible, then the negation of A, which is the formula $\neg A$, asserts that the action must occur. The action's possibility is expressed by the negation of $\vdash A$, which is a metaformula and not a formula within the logic. See [12] for more details.

2 Safety Properties

2.1 A Soda Machine

We begin with a system consisting of a soda machine, in which the user deposits either a half dollar or two quarters and the machine in return dispenses a can of soda.[2] Figure 1, together with the initial condition that the machine starts in state I, provides a simple specification of the safety properties of this machine.

Figure 1 specifies that, when the machine is in state I, either a *deposit quarter* action can occur that takes the machine to state II or a *deposit half dollar* action can occur that takes it to state III. From state II, only a *deposit*

[2] For the reader unfamiliar with colloquial American English and United States currency: *soda* is a carbonated soft drink, a *quarter* is a coin worth \$0.25, and a *half dollar* coin is worth \$0.50.

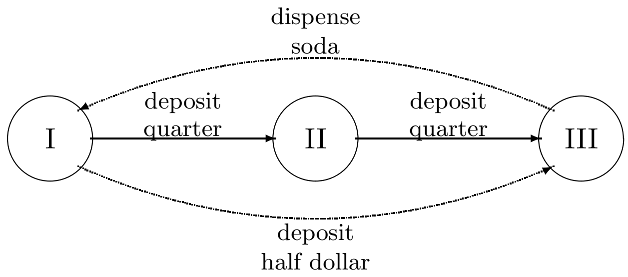

Figure 1: Specification of a soda machine

quarter action taking the machine to state III can occur. From state III, only a *dispense soda* action taking the machine to state I can occur. This is a safety specification, so it asserts that these are the only actions that are allowed to occur; it does not assert that any actions must occur.

What happens if the user deposits first a quarter then a half dollar?

The specification disallows this behavior. (Remember that the examples are not supposed to be realistic specifications.) There are two ways to view this aspect of the specification:

- The specification constrains the behavior of the user, forbidding him to deposit a half dollar after he has deposited a quarter.
- The specification does not state what the soda machine is supposed to do if the user deposits a quarter then a half dollar; the implementer is free to build the machine so it does anything he wants if the user exhibits this kind of "incorrect" behavior.

Which view we take makes no difference to how we write and reason about specifications.

Figure 1 is supposed to specify the soda machine's behavior; why does it also specify the user's behavior?

It is impossible to implement a system that functions properly in the presence of arbitrary behavior by the environment. A more realistic specification would allow the user to deposit an arbitrary sequence of coins, perhaps returning them if an inappropriate sequence had been deposited; it would not allow the user to attack the machine with a sledgehammer. (We shall see later how the sledgehammer is disallowed.) The specification of a program procedure usually includes a precondition that constrains the environment by forbidding calls whose arguments do not satisfy the precondition; the specification of a circuit includes timing constraints that restrict when the environment can change the input levels [16].

Figure 1 is a simple state-transition diagram. Such diagrams work well for very simple examples, but don't they become too complicated when specifying real systems?

Yes; these diagrams do not scale well to larger problems. State-transition diagrams represent just one particular language that can be used with the transition axiom method. The shortcomings of these diagrams are limitations of the language, not of the transition axiom method. Other languages are needed for writing transition axiom specifications of larger systems; I will have more to say about languages later.

What is the fundamental, language-independent concept that is expressed by the state-transition diagram of Figure 1?

Allowed state transitions. In the transition axiom method, one specifies safety properties by describing a set of states and all transitions between states that are allowed to occur. There are many different languages with which one can describe states and transitions.

The concept of state transitions, as illustrated by the diagram of Figure 1, has been used for years. Is there anything different about the transition axiom method?

What is new in the transition axiom method is not the diagram, but its interpretation as a formal specification. This new interpretation is needed because conventional state-transition methods do not adequately address the fundamental question of what it means for an implementation to meet such a specification. One of the advantages of the transition axiom method is that specifications of safety properties can be written in friendly, familiar notations such as state-transition diagrams. The specifications look old; the meaning we assign to them is new.[3]

What is different about the interpretation of Figure 1 in the transition axiom method?

[3] More precisely, I believe that this meaning was new when it was proposed in [13] and [14]; it has since appeared in [9] and elsewhere.

The naive interpretation of Figure 1 is that it specifies a three-state machine. In more sophisticated approaches, such as the one described by Jones [8], the diagram is interpreted to mean that there exists some state function, let's call it f, that assumes the values I, II, and III; the diagram specifies how f can change. More precisely, the soda machine is assumed to have some unspecified set of states, let's call it $\mathbf{S}$; the machine's behavior is described by the sequence of states s_0, s_1, s_2, ... it passes through. The state function f is a mapping from $\mathbf{S}$ to the set of values $\{\mathrm{I}, \mathrm{II}, \mathrm{III}\}$. The diagram of Figure 1 specifies that for each state transition $s_i \rightarrow s_{i+1}$ in this sequence, the change of value from $f(s_i)$ to $f(s_{i+1})$ is one of the following:

- $f(s_i) = \mathrm{I}$, $f(s_{i+1}) = \mathrm{II}$, and the change is caused by a *deposit quarter* action.
- $f(s_i) = \mathrm{I}$, $f(s_{i+1}) = \mathrm{III}$, and the change is caused by a *deposit half dollar* action.
- $f(s_i) = \mathrm{II}$, $f(s_{i+1}) = \mathrm{III}$, and the change is caused by a *deposit quarter* action.
- $f(s_i) = \mathrm{III}$, $f(s_{i+1}) = \mathrm{I}$, and the change is caused by a *dispense soda* action.

With this interpretation, the entire interaction of the user depositing a half dollar and the machine dispensing a soda is performed as two actions. In the transition axiom method's interpretation, we allow the additional possibility that $f(s_i) = f(s_{i+1})$ (even if $s_i \neq s_{i+1}$). The interaction of buying a soda with a half-dollar coin could involve dozens or hundreds of state transitions, only two of which change the value of f.

What is gained by this new interpretation?

In a real soda machine, dispensing a soda could involve hundreds of separate state transitions. If a specification asserts that this is just a single action, then one has to say what it means for a machine operation with hundreds of state transitions to satisfy a specification asserting that it is a single action. In the transition axiom method's interpretation, this is not a problem because the specification asserts that there is a state function that changes only once during the dispensing of the soda; it says nothing about how many separate state transitions occur. The advantages of this interpretation are discussed later.

A formal specification should provide all the necessary information to determine if an implementation is correct. However, from Figure 1, there is no way to determine if the implementation is supposed to consist of: (1) the entire soda machine, including the coin box and the soda rack, (2) a control circuit inside the machine, or (3) a program for the soda machine's microprocessor. The choice of labels on the arcs may provide some clue, but surely this choice can have no formal significance. Can a soda machine, a circuit, and a computer program all be correct implementations of the same formal specification?

A specification must be incomplete if it does not distinguish between a mechanical device, a circuit, and a program. What is missing from Figure 1 is a specification of the *interface*—the mechanism by which the system communicates with the environment. The specification must state whether communication is by depositing coins and dispensing cans, by raising and lowering voltages on wires, or by calling and returning from program procedures. The interface specification stipulates that the *deposit quarter* action may not be performed with a sledgehammer. The difference between depositing a quarter and wielding a sledgehammer, or between raising a 5 volt signal and raising a 5000 volt signal, can be described only in terms of implementation details. The interface must therefore be specified at the implementation level.

How is the interface specified?

The environment and the system communicate by changing the values of state functions. For example, if we are specifying a circuit, then communication between the circuit and its environment is achieved by changing the values of state functions that represent the voltages on wires. [4] To each wire w there might correspond a state function f_w that represents the voltage on the wire. The specification might permit the environment to communicate with the circuit by changing the value of f_w to 4.5 ± 1.2 (the voltage on the wire having some fixed value between 3.3 and 5.7 volts) when the value of $f_{w'}$ for some other wire w' equals 0 ± 1.2. (Despite the continuous range of voltages, this can still be considered a discrete system because the voltage changes are assumed to be instantaneous.)

[4] What we are really specifying is not a circuit but a mathematical model of the circuit. The state functions are the mathematical objects within the model that represent the voltages on the wires of the real circuit.

To specify the interface, we must specify how such interface state functions change. This can be done by the same method used to specify changes to internal state functions, such as the function f of the soda machine specification. Thus, no extra machinery need be added to the transition axiom method to specify the interface.

In practice, we usually don't bother specifying the interface in this way. Instead, we specify the interface in the implementation language, making it trivial to check that the interface is implemented correctly. For example, the wires connecting a circuit with its environment would be specified directly in the hardware design language used to implement the circuit. Instead of specifying how the actual voltages on a wire w change, we would describe those changes with the hardware design language's primitives, such as a $w := \mathit{true}$. The actual voltages would not be described.

How are program interfaces specified?

The exact nature of the interface specification depends upon the programming language. For a Pascal procedure, the interface is specified by giving the name of the procedure, the types of its arguments, and the names and types of any global variables accessed by the procedure. For a Modula-2 package, the interface is specified by the definition module [21].

This implies that we cannot specify a procedure independently of the language in which it is implemented. Shouldn't we be able to write a single specification of, say, a square root function that is independent of the language in which it is implemented?

We are not specifying the language in which the procedure is implemented; we are just specifying the implementation of the interface. A system whose interface is specified as a Pascal procedure could be implemented in assembly language; it need only obey the same calling conventions as a Pascal procedure.

While the specifications of a square root function for different programming languages may be similar, they will not be identical. For example, how errors are handled will depend upon whether or not the language provides an exception-handling mechanism. Separating the specification of a square root function into a common part and an interface-dependent part is a specification-language design issue that is addressed by Guttag, Horning, and Wing in Larch [6].

The influence of the interface on the rest of the specification is especially important in concurrent systems. It is shown in [15] that the specification of even so basic a property as first-come-first-served priority cannot be independent of the interface's implementation details.

You are saying that, even for the highest-level specification, the interface must be specified at the implementation level. Can't one hide these low-level implementation details in the high-level specification?

The interface is specified by describing how interface state functions can change. We shall see below how the changes to internal state functions can be specified hierarchically; the same approach can applied to the interface state functions. However, the high level specification is not complete until the interface is completely specified down to the implementation level. A complete specification should eliminate the need for any communication between the user of the system and its implementor. For example, the specification of a control circuit for a soda machine should contain all the information about that circuit's behavior needed by the person designing the rest of the machine, which means that it must specify the actual voltages on the wires.

While a hierarchical decomposition of the interface may be quite useful, it is logically just a method of organizing the high level specification. I will therefore not consider such a decomposition of the interface.

Can interfaces be specified solely in terms of state functions?

In addition to interface state functions, we need to introduce the notion of who is responsible for changing the values of these state functions. A specification of the soda machine interface must state that the environment (the user) performs the *deposit quarter* and *deposit half dollar* actions and the system (the machine) performs the *dispense soda* action. The specification of a procedure interface must state that the environment (the rest of the program) performs the procedure call and the system (the procedure) performs the return.

Usually, actions are performed either by the environment or by the system. However, it is sometimes useful to assert which part of the environment performs an action. For example, to specify a process that interacts with its environment through both shared variables and CSP-style operations [7], it may be useful to distinguish actions performed by a communication channel

("!" and "?" operations) from ones performed by other processes (setting shared variables).[5]

Why is it necessary to state who performs an interface action?

Consider a Modula-2 package that implements a queue by providing *put* and *get* procedures. If we failed to specify that only the environment can call these procedures, then the specification would be satisfied by an implementation that calls the *put* procedure itself to cause random elements to appear in the queue.

What is the general form of a safety specification in the transition axiom method?

A safety specification consists of:

- A set of state functions, partitioned into interface and internal state functions.
- A specification of the initial value of every state function.
- A set of actions, partitioned into interface and internal actions.
- For each interface action, a specification of who performs the action. (Internal actions are always performed by the system.)
- A set of rules, called transition axioms, that describe how each action changes the state functions. An interface state function may be changed only by an interface action.

In the soda machine example, there is a single internal state function f whose initial value is I; the interface state functions have been left unspecified. There are three actions, all of which are interface actions: *deposit quarter* and *deposit half dollar*, performed by the environment, and *dispense soda*, performed by the system. The effect of the *deposit quarter* action is described by a transition axiom asserting that the action can occur only when f equals I, in which case it changes the value of f to II, or when f equals II, in which case it changes the value of f to III. The rules for the *deposit half dollar* and *dispense soda* actions are similar, but a bit simpler. A

[5]Although one often thinks of a CSP communication as an action performed jointly by the sender and the receiver, thinking of it as an action of the channel makes it unnecessary to introduce the concept of joint actions.

complete soda machine specification would also have to describe how these three actions change the interface state functions.

Precisely what is the meaning of such a specification?

The formal meaning of a transition axiom specification is a formula of temporal logic. To give a rigorous definition of that meaning, one must define the formal semantics of temporal logic and provide an algorithm for translating a specification into a temporal logic formula. This is done in [13]. Instead of taking such a formal approach here, I will try to provide an intuitive understanding of what a transition axiom specification means through careful consideration of what it means to implement the specification.

In developing an intuitive understanding of transition axiom specifications, it is useful to know the general shape of the formula underlying a specification. I will ignore the part of the specification having to do with who performs the actions. Let $f_1, \ldots, f_n$ be the internal state functions and $g_1, \ldots, g_m$ be the interface state functions of the specification. The formal meaning of the specification is a temporal logic formula of the form[6]

$$\exists f_1, \ldots, f_n \text{ s.t. } X$$

where X is a formula describing how the f_i and g_i are allowed to change. More precisely, X is a formula that constrains the sequence of states the system assumes by constraining the values of the state functions f_i and g_i on this sequence of states.

Why is there quantification only over internal state functions and not over interface state functions?

The absence of quantification over the interface state functions is the formal expression of the observation that the interface must be specified at the implementation level. The existential quantification over the internal state functions allows complete freedom in how these state functions are implemented. Because the interface state functions are free (not quantified over) in the specification, those same state functions must appear in the implementation. All this should become clearer with the next example, which will serve to address the question of what it means for an implementation to be correct.

[6] Note that the interface state functions g_i are free variables in the formula; the significance of this is discussed below.

interface procedures *deposit_coin* ... ;
dispense_soda ... ;
var x: $\{0, 25, 50\}$;
y: $\{25, 50\}$;
begin loop α: $\langle\, x := 0 \,\rangle$;
β: **while** $\langle\, x < 50 \,\rangle$
do γ: $\langle\, y :=$ *deposit_coin* **only if** $x + y_{\text{new}} \leq 50 \,\rangle$;
δ: $\langle\, x := x + y \,\rangle$
od;
ϵ: $\langle$ *dispense_soda* $\rangle$
end loop

Figure 2: Another specification of a soda machine.

2.2 Another Specification of a Soda Machine

Figure 2 is a soda machine specification written in an *ad hoc* language that resembles an ordinary declarative programming language. The **interface procedures** declarations provide the interface specification, which is omitted; the **var** declarations determine the range of values that can be assumed by x and y. Angle brackets denote that the operation they surround is a single (atomic) action. Statement γ performs an action consisting of a *deposit_coin* interface action plus the action of setting the value of y to the value of the deposited coin, the **only if** clause (a notation invented just for this statement) meaning that the action can take place only if the value of $x + y$ after the assignment is at most 50. Thus, the **only if** clause disallows the possibility of depositing a half dollar after a quarter is deposited.

Figure 2 looks like a program. How is it interpreted as a transition axiom specification?

To interpret Figure 2 as a transition axiom specification, we must describe the state functions, actions, etc. that it defines. The internal state functions are the variables x and y and an additional state function, let us call it *pc*, that describes the program control state; the interface state functions are presumably specified (perhaps implicitly) in the omitted part of the **interface procedures**. The state function x can assume the values 0, 25, and 50; the state function y can assume the values 25 and 50; and the state function *pc* can assume the values α, β, γ, δ, and ϵ. The initial values of x and y are unspecified; the initial value of *pc* is α, indicating that control is initially at statement α. There are five actions, one for each pair of angle brackets, that are labeled $\alpha \ldots \epsilon$. Actions γ and ϵ are interface actions, the former performed by the environment and the latter by the system; the rest are internal actions. The transition axioms for these actions specify the following allowed changes to the state functions.

α: This action can occur only when pc has the value α. It changes the value of x to 0, it changes the value of pc to β, and it leaves the value of y unchanged.

β: Can occur only when $pc = \beta$. If $x < 50$ then it changes the value of pc to γ, otherwise it changes the value of pc to ϵ. It leaves the values of x and y unchanged.

γ: Represents the user action of depositing a coin. It sets the value of y equal to the value of the coin deposited, which must be either a quarter or a half dollar (because y can equal only 25 or 50); it leaves the value of x unchanged. This action can occur only when $pc = \gamma$ and the new value of y will satisfy $x + y \leq 50$.

δ: Can occur only when $pc = \delta$. It sets the value of x equal to its old value plus the old value of y, it leaves the value of y unchanged, and it sets the value of pc to β.

ϵ: Can occur only when $pc = \epsilon$. It sets pc to α and leaves the values of x and y unchanged. This action represents the dispensing of a can of soda.

In this specification, x and y look like ordinary program variables, but pc seems strange. Isn't there a fundamental difference between the state function pc and the state functions x and y?

No. To describe the execution of a program written in a declarative programming language, we must describe how the program control position changes as well as how the values of variables change. The programmer cannot explicitly refer to the "program counter", but its value is just as much part of the program state as is the value of an ordinary variable. A programmer often has the choice of whether to use an extra variable or a more complicated control structure to represent the state of a computation.

Is every program a specification?

Yes. A program written in any programming language can be interpreted as a transition axiom specification. A major task in writing a compiler from a source language to a target language is to represent, in the target language, the state functions specified by the program, including ones like *pc* and the procedure-calling stack that are not explicitly declared. A program written in a higher-level language is a specification of the object code produced by the compiler. The only difference between a program and a higher-level specification is that the program is implemented by the compiler without human intervention.

If any program is a specification, why not write specifications in an ordinary programming language instead of devising specification languages?

This can be done. However, programming languages are constrained by the requirement that programs must be compiled into reasonably efficient code. Because specifications do not have to be compiled, specification languages can permit simpler specifications than can be written in programming languages. Also, programming languages tend to encourage overly restrictive specifications. For example, in most programming languages it is easy to state that one action must follow another but hard to state that two actions can be performed in either order. Such languages encourage specifications that unnecessarily constrain the order in which actions must be performed.

What kind of constructs can specification languages use that programming languages cannot?

The primary programming language construct for indicating explicit state changes is the assignment statement. In a specification language, the assignment statement can be extended to allow an arbitrary relation between the old and new values of state functions. For example, statement γ of Figure 2 can be described as the following relation between the old and new values of the variables:

$$y_{\text{new}} = \mathit{deposit\_coin} \;\wedge\; x_{\text{new}} = x_{\text{old}} \;\wedge\; y_{\text{new}} + x_{\text{old}} \leq 50 \qquad (1)$$

where *deposit_coin* is some expression involving old and new values of interface state functions that is presumably defined by the omitted interface specification.

An ordinary assignment statement is a specific form of relation in which the new value of a variable equals an expression involving only the old values of variables. However, one can have more general relations, such as

$$a_{\mathrm{old}} = \sin^2 b_{\mathrm{new}} + 3 \cos b_{\mathrm{new}}$$

which expresses a relation between the new value of the variable b and the old value of the variable a. Such a relation cannot be expressed in a programming language because it cannot be compiled into efficient code, but there is no reason not to allow it in a specification language.

A transition axiom for an action, which determines the changes to state functions allowed by the action, is just such a relation between old and new values of state functions. For example, the transition axiom for statement γ is obtained by conjoining relation (1), which asserts how x and y may change, with

$$pc_{\mathrm{old}} = \gamma \ \wedge \ pc_{\mathrm{new}} = \delta$$

The latter relation asserts how pc may change and includes the requirement that the action can be performed only when the initial value of pc equals γ. The program of Figure 2 can be replaced by a set of five transition axioms of this form. However, the specification is easier to follow if we use ordinary programming language constructs like ";" and **while** to describe implicitly how the value of pc may change instead of explicitly writing the relations between its old and new values.

Figures 1 and 2 are two different specifications of the soda machine. How are they related?

They are equivalent—assuming that they are completed with the proper interface specifications. In other words, each one is a correct implementation of the other. I will show that Figure 2 correctly implements Figure 1. Demonstrating the converse requires some concepts that will be introduced with another example.

The interpretation of Figure 1 as a transition axiom specification asserts the existence of a state function f with certain properties. To prove that Figure 2 satisfies this specification, we must demonstrate the existence of f. This is done by defining f in terms of the state functions x, y, and pc, whose existence is asserted by the interpretation of Figure 2 as a transition axiom specification. We first observe that $x < 50$ when $pc = \gamma$ and $x + y \leq 50$ when $pc = \delta$. (This is proved by showing that these two assertions are true initially and are left true by every action.) The value of f is defined by the

$$
\begin{array}{lll}
\textbf{if}\ pc = \alpha & \rightarrow f = \mathrm{I} \quad \square & \\
\quad pc = \beta \text{ or } pc = \gamma & \rightarrow \textbf{if}\ x = 0 \ \rightarrow f = \mathrm{I} \quad \square & \\
 & \qquad x = 25 \rightarrow f = \mathrm{II} \ \square & \\
 & \qquad x = 50 \rightarrow f = \mathrm{III} \text{ (impossible if } pc = \gamma) & \\
 & \ \textbf{fi} \ \square & \\
\quad pc = \delta & \rightarrow \textbf{if}\ x + y = 25 \rightarrow f = \mathrm{II} \quad \square & \\
 & \qquad x + y = 50 \rightarrow f = \mathrm{III} \ \square & \\
 & \qquad x + y = 75 \rightarrow \text{impossible} & \\
 & \ \textbf{fi} \ \square & \\
\quad pc = \epsilon & \rightarrow f = \mathrm{III} & \\
\textbf{fi} & &
\end{array}
$$

Figure 3: Definition of f in terms of x, y, and pc.

expression in Figure 3, written using Dijkstra's **if** construct.[7] Finally, we show that, with this definition of f, every action allowed by the specification of Figure 2 either leaves f unchanged or corresponds to an action (a change of f) allowed by the specification of Figure 1. The reader can check that actions α, β, and δ do not change f. For example, α can be executed only when $pc = \alpha$, in which case $f = \mathrm{I}$, and its execution sets $pc = \beta$ and $x = 0$, which leaves f equal to I. The reader can also check that execution of γ corresponds either to a *deposit quarter* or a *deposit half dollar* action allowed by Figure 1. For example, suppose γ is executed starting in a state with $x = 25$. Since it can only be executed when $pc = \gamma$, this implies that initially $f = \mathrm{II}$. The specification of the γ action implies that, starting with $x = 25$, it can change the values of x, y, and pc only by setting y to 25 and pc to δ, which makes $f = \mathrm{III}$. This change of the value of f from II to III is permitted by the *deposit quarter* action of Figure 1. The reader can check that executing γ starting with $x = 0$, the only other possibility, also yields a change of f allowed by the *deposit quarter* or *deposit half dollar* action of Figure 1, and that executing ϵ changes f as allowed by the *dispense soda* action of Figure 1.

Is this all there is to the proof?

We have not proved that Figure 2 correctly implements the interface of Figure 1. This requires showing that *deposit_coin* and *dispense_soda* are

[7] Note that Figure 3 is not a program; it is just an ordinary mathematical definition of f as a function of x, y, and pc written with Dijkstra's notation.

correct implementations of the corresponding actions of Figure 1, which we cannot do because neither they nor the interface of Figure 1 have been specified.

How can we formalize the informal reasoning used in the proof?

Let a *state vector* for the specification of Figure 2 be a triple of possible values of x, y, and pc, and let a state vector for the specification of Figure 1 be a possible value of f (either I, II, or III). To define f in terms of x, y, and pc, we defined a mapping F from state vectors of Figure 2 to state vectors of Figure 1. For example, $F(0, 25, \delta) = \text{II}$ means that $f = \text{II}$ when $x = 0$, $y = 25$, and $pc = \delta$.

For any action ξ, let A_ξ denote the transition axiom for ξ. This is a relation between old and new values—in other words, a set of pairs of state vectors. For example, the pair $((0, 50, \gamma), (0, 25, \delta))$ is in A_γ because it is possible to execute γ starting with $x = 0$, $y = 50$, and $pc = \gamma$ and ending with $x = 0$, $y = 25$, and $pc = \delta$.

Let $\mathbf{A_1}$ and $\mathbf{A_2}$ denote the set of actions of Figures 1 and 2, respectively. Formally, we proved the theorem

$$\forall \xi \in \mathbf{A_2}\ \forall (v, w) \in A_\xi\ \exists \eta \in \mathbf{A_1}\ \text{s.t.}\ (F(v), F(w)) \in A_\eta$$

This formula has the following English translation, where bracketed expressions indicate the correspondence with the formula: for every action [ξ] of Figure 2 [in $\mathbf{A_2}$] and every change to the values of x, y, and pc [from v to w] allowed by this action [(v, w) in A_ξ], there exists an action [η] of Figure 1 [in $\mathbf{A_1}$] such that the corresponding change to the value of f [from $F(v)$ to $F(w)$] is allowed by that action [$(F(v), F(w))$ is in A_η].

Exactly what does this prove?

The formal meaning of the specification of Figure 1 is a formula $\exists f$ s.t. X_1, where X_1 is a formula describing how the value of f is allowed to change; the meaning of Figure 2 is a formula $\exists x, y, pc$ s.t. X_2, where X_2 describes how the values of x, y, and pc are allowed to change. We proved the formula

$$(\exists x, y, pc \text{ s.t. } X_2) \supset (\exists f \text{ s.t. } X_1)$$

Thus, correctness of an implementation means simple logical implication: the specification is implied by the (specification of the) implementation. This implication was proved by proving $X_2 \supset \overline{X_1}$, where $\overline{X_1}$ is the formula obtained by substituting for f in X_1 its expression in terms of x, y, and pc defined in Figure 3. In other words, assuming the existence of the state functions x, y, and pc satisfying X_2, we proved the existence of a state function f satisfying X_1 by explicitly constructing the required f in terms of x, y, and pc.

In a complete specification, X_1 and X_2 would also describe the allowed changes to the interface state functions. However, because there is no quantification over interface state functions, this type of argument can work only if X_1 and X_2 contain the same interface state functions, and the behavior of those interface state functions asserted by X_1 is implied by the assertions about their behavior made by X_2. This is the formal statement of the observation that the interface must be specified (in X_1) at the implementation level (using the same state functions and interface actions as in X_2).

The proof that Figure 2 correctly implements Figure 1 can be reduced to logical implication because both specifications are expressed by formulas in the same logical system. This in turn is possible only because we interpret the state-transition diagram of Figure 1 in terms of the state function f. If Figure 1 were interpreted as specifying the behavior of a three-state machine while Figure 2 specified the behavior of a 30-state machine, then they would express formulas in different logical systems and it would not be clear what correctness of an implementation meant. Traditional definitions of correctness of an implementation, involving mappings on behaviors, have ignored problems that arise in specifying the interface. See [14] for further discussion of this issue.

2.3 A Database

Let us now consider a toy specification of a database concurrency control mechanism. Clients of the database issue operations by calling an *exec* procedure with arguments indicating the operation to be performed. There are two arguments: *op*, indicating the change to the database and *res*, indicating the value to be returned. (I assume that, as in Modula-2, procedures may return values.) These arguments are described formally as functions. Although the *exec* procedure may be called concurrently by multiple clients, the operations are to be performed as if they occur in a serial order. In other words, the database operations are to be performed as if they were atomic.[8]

[8] In database circles, atomicity often means that a failure cannot result in a partially completed operation. The possibilty of failure is not considered in this example.

type dbase : ... ;
value : ... ;
op_fcn : dbase $\rightarrow$ dbase;
res_fcn : dbase $\rightarrow$ value;

internal state function *data* : dbase;

procedure *exec*(*op*: op_fcn; *res*: res_fcn) : value
begin α: $\langle\, data_{\mathrm{new}} = op(data_{\mathrm{old}}) \ \wedge\ exec_{\mathrm{new}} = res(data_{\mathrm{old}})\,\rangle$
end

Figure 4: A database specification.

The specification is given in Figure 4, using programming language notation. The internal state function *data* represents the state of the database. The interface is specified as a procedure call. There is a single internal action α. In this action, *op* and *res* denote the arguments of the procedure call, and *exec* is the value that is returned by the procedure. The procedure call and return are interface actions that are not explicitly specified.

How is Figure 4 interpreted as a transition axiom specification?

The interface specification must contain state functions whose values indicate the set of processes currently executing the *exec* procedure—that is, the set of processes that executed a call that has not yet returned. There must also be state functions that indicate the following information for each such process:

- The values of *op* and *res*.
- The program control location, indicating whether the process is executing the call, is at control point α, or is executing the return.
- The value of *exec* (the value to be returned), if the process has executed action α.

Figure 4 seems to specify that the entire database operation must be done as a single atomic action; doesn't this rule out concurrency?

The specification asserts the existence of a state function *data* that changes atomically; it does not assert that changes to the database must actually be performed atomically. Figure 2 implements Figure 1 even though the operations of depositing a half dollar and dispensing a can of soda consist of two atomic actions in Figure 1 and six atomic actions in Figure 2. The implementation was proved correct by defining f in such a way that only two of those six actions change f, doing so as allowed by Figure 1; the other four actions leave f unchanged.

In the same way, the change to the database, which is represented by executing the single atomic action α in Figure 4, can be implemented as a sequence of thousands of atomic actions. Correctness of the implementation means that the state function *data* can be defined as a function of the implementation state functions in such a way that only one of those thousands of atomic actions changes its value, doing so as indicated by Figure 4.

The state function must be defined so that a single atomic action causes the entire database operation, which could be arbitrarily complex, suddenly to be performed—even though each atomic action makes only a small change to the actual database. How is this possible?

The only way to understand how it is done is by working out an example. One such example is the specification and implementation of a FIFO queue in [13], where the specification asserts that the operations of adding and removing an element from the queue are atomic, but an implementation that adds and removes elements one bit at a time is proved correct.

The proof method is a generalization of assertional methods for proving safety properties of concurrent programs [18]. In these assertional methods, one constructs an invariant, which is a boolean state function whose value never changes; in the transition axiom method, one constructs state functions whose values change only in the manner prescribed by the transition axioms.

2.4 Another Database Specification

The concurrency control mechanism specified by Figure 4 is called serialization [5] because database operations are executed as if they occurred in some serial order. However, Figure 4 is not the most general specification of serialization because it requires that the actual reading and writing of the database occurs between the call of *exec* and the subsequent return, which implies that if one call to *exec* returns before another call is initiated, then the operation performed by the first call precedes the operation performed by the second in the serialization order. Some concurrency control algorithms that are considered to be serializable do not have this property. So,

type dbase : ... ;
value : ... ;
op_fcn : dbase $\rightarrow$ dbase;
res_fcn : dbase $\rightarrow$ value;

internal state function
gdata : dbase;
saved_ops : **bag of** (op_fcn, res_fcn, value);

procedure *exec*(*op*: op_fcn; *res*: res_fcn) : value
begin β: $\langle\, \exists v \text{ s.t. } exec_{\text{new}} = v \;\wedge$
$saved\_ops_{\text{new}} = saved\_ops_{\text{old}} \cup \{(op, res, v)\}\,\rangle$
end

internal process
begin loop γ: $\langle\, \exists (o, r, v) \in saved\_ops_{\text{old}}$ s.t.
$gdata_{\text{new}} = o(gdata_{\text{old}}) \;\wedge$
$v = r(gdata_{\text{old}}) \;\wedge$
$saved\_ops_{\text{new}} = saved\_ops_{\text{old}} - \{(o, r, v)\}\,\rangle$
endloop
end

Figure 5: A more general database specification.

let us now consider the more general specification of a serializable database given in Figure 5.

The interface of the new specification is the same as that of Figure 4: a procedure named *exec* with two arguments that specify the operation. However, instead of performing the operation immediately, action β chooses a completely arbitrary value to return (the value v) and saves that value together with the arguments of the procedure call in *saved_ops*, a bag of operations to be performed later.[9] A separate, asynchronous action γ at some later time will perform the operation. The **internal process** keyword denotes that its actions are performed by the system independently of actions performed through calls to the *exec* procedure. The clause $v = r(gdata_{\text{old}})$ in the specification of action γ means that the action is performed only if the database state is such that the result that was already chosen (by the β

[9] A bag, also called a *multiset*, is a set in which the same element can appear more than once.

execution that put the triple (o, r, v) in the bag *saved_ops*) was the correct one.

Since this is a safety specification, it does not assert that γ will ever do anything; it simply asserts that γ cannot perform a database operation unless the result agrees with the one that the β action had already decided to return. We must also require the liveness property that every operation saved in *saved_ops* is eventually performed. Section 3 indicates how this property is specified.

This specification is completely bizarre; it requires that the exec procedure guess what the correct result of the database operation will be before actually executing it. How can one possibly implement such a specification?

Figure 5 is bizarre only if viewed as a description of *how* the *exec* procedure is to be implemented. A program describes how something is to be done, while a specification describes only *what* is to be done. Figure 5 describes the observable behavior of the database system; it makes no formal assertion about how that behavior is to be implemented.

It is important to realize that even though a transition axiom specification may look superficially like a program, apparently describing how the system is to be implemented, it really specifies only the externally visible behavior—that is, how the interface state functions may change. Internal state functions such as *saved_ops* need not appear in any obvious form in the implementation's data structures. Indeed, as I will explain below, they need not appear at all.

If Figure 5 is a more general specification than Figure 4, then Figure 4 should implement Figure 5. However, the implementation executes fewer actions than the specification, a single α action performing the database operation that the specification asserts is done by a β action (to put the operation in saved_ops) and a γ action (to change the database). How do we prove that this is a correct implementation?

As in the proof for the soda machine example, we must define the specification state functions *gdata* and *saved_ops* in terms of the implementation state function *data*. We let *gdata* equal *data* and define *saved_ops* always to equal the empty bag. Again we must show that every action of the implementation changes the values of the specification state functions as allowed by the specification actions. However, we drop the requirement that each implementation action corresponds to at most a single specification action and allow it to correspond to a sequence of specification actions. There is only one internal action in Figure 4: the action α. Executing action α produces the same changes to *gdata* and *saved_ops* as an execution of a β action followed by an execution of a γ action. The γ action immediately executes the operation that the β action puts in *saved_ops*, the total effect being to leave *saved_ops* empty and to produce the required change to *gdata*.

Is one always allowed to implement a sequence of specification actions with a single implementation action?

Two conditions must be satisfied for a sequence of specification actions to be implementable by a single implementation action:

- At most one of the specification actions can be an interface action.
- All the actions (the specification actions and the implementation action) must be performed by the same agent—either the system or the environment.

These two conditions rule out pathological implementations.

The case of several specification actions implemented with a single action arises only when demonstrating that one specification is at least as general as another. (I just demonstrated that Figure 5 is at least as general as Figure 4.) In real implementations, a single specification action is usually implemented with dozens or even thousands of separate actions.

What is the formal justification for the correctness of implementing several specification actions with a single action?

Recall that the formula represented by a transition axiom specification is of the form $\exists f_1, \ldots, f_n$ s.t. X, where the f_i are the internal state functions. The reason we are allowed to implement several actions with a single one lies in the formal meaning, given in [4], of existential quantification of a state function. In proving that the formula represented by the implementation implies the formula represented by the specification, the existential quantification over the internal state functions allows one to consider the execution obtained by splitting one action into several successive actions to be equivalent to the original execution if the extra actions change only the internal state functions. However, an explanation of why this is so involves subtle points of temporal logic that are beyond the scope of this paper.

Is this all there is to the proof that Figure 4 correctly implements Figure 5?

Yes. The two systems have identical interfaces, so it is obvious that the interface actions of Figure 4—the ones that perform the procedure call and the return—correctly implement the interface actions of Figure 4; they are the same actions. Therefore, we just have to show that the internal state functions of Figure 5 are correctly implemented by Figure 4, which we did.

3 Liveness Properties

Liveness properties assert that something must happen. In a transition axiom specification, the things that happen are changes to values of state functions; what must happen is expressed by explicit axioms about how these values must eventually change.

Axioms to specify liveness are written in *temporal logic*, obtained by extending ordinary logic with the temporal operators $\Box$ (read *henceforth*) and $\Diamond$ (read *eventually*). The formula $\Box P$ asserts that P is true now and at all future times, and the formula $\Diamond P$ asserts that P is true now or at some future time. Since P is eventually true if and only if it is not always false, $\Diamond P$ is equivalent to $\neg\Box\neg P$. (See [12] for a discussion of this equivalence.) It is convenient to define the operator $\leadsto$ (read *leads to*) by letting $P \leadsto Q$ equal $\Box(P \supset \Diamond Q)$, which asserts that whenever P becomes true, Q will be true then or at some later time. A more detailed exposition of our temporal logic can be found in [19].

In the soda machine specification of Figure 1, we might require that, after the user has deposited enough money, the machine must eventually dispense the soda. This is expressed by the formula $(f = \mathrm{III}) \leadsto (f = \mathrm{I})$, which asserts that if $f = \mathrm{III}$ then f must eventually equal I.

The soda machine specification should probably have no other liveness axioms, since we don't require that the user must deposit money. However, we might require that if he deposits one quarter then he must deposit another, which is asserted by the axiom $(f = \mathrm{II}) \leadsto (f = \mathrm{III})$.

In the soda machine specification of Figure 2, we require that the next action must eventually be performed, except if it is a γ action, which the user need never perform. If the next action is an α action, this is asserted by the axiom $(pc = \alpha) \leadsto (pc = \beta)$. However, we could instead make only the weaker assertion that $(pc = \alpha) \leadsto (pc \neq \alpha)$, since Figure 2 implies that if $pc = \alpha$, then the only way the value of pc can change is for it to become equal to β. The complete liveness specification for this example is

$$\forall \xi \neq \gamma : \ (pc = \xi) \leadsto (pc \neq \xi) \tag{2}$$

These liveness axioms are obvious from looking at Figures 1 and 2. Can't we just make the liveness axioms implicit in the language instead of having to write them separately?

One might want to make certain liveness axioms implicit in the language. However, the liveness conditions that appear in specifications are too varied to be expressed only implicitly by any reasonable collection of language constructs.

The informal liveness requirement for the database specification of Figure 5 is that any operation saved in saved_ops *is eventually executed. How is this expressed formally?*

Our first attempt at specifying this might be the axiom

$$\forall (o, r, v) : \ (o, r, v) \in \mathit{saved\_ops} \leadsto (o, r, v) \notin \mathit{saved\_ops}$$

which asserts that if a triple (o, r, v) is in the bag *saved_ops*, then eventually it will not be in that bag. The rest of the specification implies that the only way a triple can be removed from the bag is by performing the appropriate database operation with a γ action.

This axiom would express the desired requirement if *saved_ops* could never contain two copies of one triple. However, if the same triple (o, r, v) were continually inserted by different calls to the *exec* procedure, then *saved_ops* might always contain a copy of (o, r, v), so the axiom would not be satisfied. All we can assert is that, if some triple (o, r, v) is in *saved_ops*, then eventually at least one copy of it is removed—that is, eventually there is a γ action that removes (o, r, v).[10] Our formulas mention states, not actions; we assert that a γ action occurs by a temporal formula asserting that, at some time, the bag contains k copies of the triple and, at a later time, it contains fewer copies. The desired liveness condition is expressed by the following axiom, where $e \sharp B$ denotes the number of copies of element e in bag B:

$$\begin{array}{l} \forall (o, r, v) : \ [(o, r, v) \in \mathit{saved\_ops}] \leadsto \\ \qquad [\exists k \text{ s.t. } ((o, r, v) \sharp \mathit{saved\_ops} = k) \ \wedge \ \Diamond((o, r, v) \sharp \mathit{saved\_ops} < k)] \end{array}$$

[10]Note that identical triples are indistinguishable, so it makes no sense to ask which copy of a triple is removed.

One can introduce notations that make it easier to assert that a certain action eventually occurs, allowing this axiom to be written more or less as

$$[(o, r, v) \in \mathit{saved\_ops}] \leadsto \gamma(o, r, v)$$

However, explaining these notations would lead us into language design issues that I do not wish to discuss here.

Are $\Box$ and $\Diamond$ (and operators like $\leadsto$ defined in terms of them) all one needs for specifying liveness properties?

Yes.

How does one verify that an implementation satisfies the liveness properties of a specification?

One must verify each liveness axiom. Consider the liveness axiom

$$(f = \mathrm{III}) \leadsto (f = \mathrm{I}) \tag{3}$$

for the specification of Figure 1. To prove that Figure 2 implements this specification, we defined f in terms of the implementation state functions x, y, and pc, the definition appearing in Figure 3. Substituting this expression for f in (3) yields

$$[(pc = \beta \wedge x = 50) \vee (pc = \delta \wedge x + y = 50) \vee (pc = \epsilon)] \leadsto [(pc = \alpha) \vee \ldots] \tag{4}$$

To verify that the implementation satisfies axiom (3), we must prove that the liveness axioms and the safety properties of the specification of Figure 2 imply (4). (This makes sense because (4) is an expression about the implementation state functions.)

From the liveness axiom (2) and the safety properties, we can establish the following chain of $\leadsto$ relations:[11]

$$(pc = \delta \wedge x + y = 50) \leadsto (pc = \beta \wedge x = 50) \leadsto (pc = \epsilon) \leadsto (pc = \alpha)$$

For example, to verify $(pc = \beta \wedge x = 50) \leadsto (pc = \epsilon)$, observe that (2) implies that eventually $pc \neq \beta$, and the transition axioms imply that if $pc = \beta$ and $x = 50$, then the value of pc can change only to ϵ. (Note that the proof uses both safety and liveness properties of the implementation.)

We leave it to the reader to check that this chain of $\leadsto$ relations intuitively implies (4). The formal method underlying all of this informal reasoning is described in [19].

[11] The formula $A \leadsto B \leadsto C$ is an abbreviation for $(A \leadsto B) \wedge (B \leadsto C)$.

What is the general method behind this example?

Recall that formally, a specification is a formula $\exists f_1 \ldots f_n$ s.t. X, where the f_i are the internal state functions and X is a formula specifying how the values of the internal and interface state functions change. Similarly, the implementation is represented by a formula $\exists h_1 \ldots h_m$ s.t. Y, where the h_i are the implementation's internal state functions and Y is a formula involving the h_i and the interface state functions. Correctness of the implementation is expressed by the formula

$$(\exists h_1 \ldots h_m \,\text{s.t.}\, Y) \supset (\exists f_1 \ldots f_n \,\text{s.t.}\, X)$$

This formula is proved by expressing the specification state functions f_i in terms of the implementation state functions h_i and proving $Y \supset \overline{X}$, where $\overline{X}$ is the formula obtained from X by substituting for the f_i their expressions in terms of the h_i.

Splitting the specification into its safety and liveness requirements means writing $X = X_s \wedge X_l$, where X_s are the safety axioms and X_l are the liveness axioms, and similarly writing $Y = Y_s \wedge Y_l$. When we prove that the safety properties of the specification are satisfied, which we do by showing that every implementation action that changes the specification state functions does so as allowed by some specification action, we are proving $Y_s \supset X_s$. To prove that the liveness properties of the specification are satisfied, we prove $(Y_s \wedge Y_l) \supset X_l$; in other words, we use both safety and liveness properties of the implementation to prove the liveness properties of the specification.

4 Further Questions

4.1 Internal State Functions

A specification should specify only the externally observable behavior of a system, yet the transition axiom method introduces internal state functions and internal transitions. Doesn't this produce overly restrictive specifications?

To specify externally observable behavior, one must describe all permitted sequences of interface actions. Most conventional methods for specifying sequences of actions use implicit internal states. For example, a context-free grammar is equivalent to an automaton, whose states are implicit in the grammar. Milner's CCS [17] can be viewed as a single automaton whose states are the set of CCS formulas. It would be easy to use context-free grammars or CCS as the language in which to express transition axioms. Using explicit rather than implicit internal state functions does not make the specifications any more restrictive.

Temporal logic and other axiomatic methods have been used to write specifications that do not mention internal states. Aren't these specifications more general than transition axiom specifications?

Let us call a specification *purely temporal* if it does not mention internal states. The work of Alpern and Schneider [3] shows that purely temporal specifications are no more general than transition axiom specifications. They defined a logic that is at least as powerful as most of the logics used for purely temporal specifications and showed that any formula in their logic is equivalent to an assertion about an automaton constructed from the formula. This automaton can be interpreted as a transition axiom specification that is equivalent to the purely temporal specification represented by the original formula.

Even if purely temporal specifications are logically no more general than transition axiom specifications, doesn't their avoidance of explicit internal state functions mean that, in practice, they are less likely to overly constrain the implementation?

The transition axiom method does make it easier than purely temporal methods to describe a particular implementation instead of specifying only the desired interface behavior. However, eliminating internal state functions requires the use of complicated temporal formulas. The reader can appreciate the extra complexity needed to specify behavior with purely temporal methods by writing two informal prose specifications of a memory register: one that uses the value of the register (which is an internal state function) and a purely temporal one that talks only about read and write operations without mentioning the register's value.

I have found that purely temporal specifications are hard to understand. While they are less likely to overspecify the system, they are much more likely to underspecify it by omitting important constraints. In practice, purely temporal methods are hard to use because they don't tell one where to start (what properties should be specified explicitly and what properties should be consequences of other properties?) or when to stop (do all desired properties follow from the specification?). In contrast, the transition axiom method provides a well structured approach to writing specifications: first choose the state functions, then specify how they are allowed to change (the transition axioms), and finally specify when they must change (the liveness axioms).

Proving the correctness of an implementation requires defining the specification state functions in terms of the implementation state functions. Aren't there cases when this is impossible because some specification state functions are unnecessary and are not actually implemented?

Yes. One example is a program, viewed as a specification of its compiled version, in which an optimizing compiler eliminates a local variable that it discovers is set but never read. Moreover, the unimplemented state function need not be unnecessary. Imagine a specification that begins by letting the system decide if it is to act as a soda machine or a database, and thereafter acts exactly like the single system chosen. This absurd specification describes the state functions for both the soda machine and the database. However, the specification can be met by implementing either a soda machine or a database, without implementing the state functions of the other.

How is the correctness of an implementation proved if it does not implement the specified state functions?

In proving the correctness of the implementation, one is allowed to add *auxiliary state functions* to the implementation. An auxiliary state function is similar to an auxiliary variable added to prove the correctness of a concurrent program [18]. It is an internal state function that is added in such a way that it does not alter the specification of how the "real" state functions are allowed to change. The existing transition axioms are modified to indicate how they change the auxiliary state functions.

By adding auxiliary state functions, isn't one proving the correctness of a new implementation—one with extra state functions—rather than the original implementation?

No. To understand why not, one must again examine the formal meaning of existential quantification over state functions. Intuitively, the formula $\exists h$ s.t. A asserts the existence of h not in the "real world", in which the only state functions that exist are the ones described by the implementation, but in a "mythical world" in which every possible state function is assumed to exist. The auxiliary state functions do not change the implementation; they serve as constructive proofs of the existence of certain possible state functions. One could rewrite the correctness proof to eliminate the auxiliary state functions, but the resulting proof would be harder to understand.

The situation in which a specification state function is not expressible in terms of implementation state functions is atypical. Just as a good program does not compute values that are never used, a good specification does not include state functions that are not needed. Specifications that give the implementer the choice of which state functions to implement are rare; in practice, one does not specify a system that can choose to act as either a soda machine or a database system. Auxiliary state functions are therefore seldom needed. I advise against introducing them just to make it easier to express the specification state functions, since one learns a great deal about an implementation by expressing the specification state functions in terms of the "real" implementation state functions.

4.2 Concurrency

The transition axiom method is supposed to specify concurrent systems, yet the system's behavior is described as a sequence of actions. Where's the concurrency?

Underlying almost all formal methods in computer science is the assumption that the behavior of a system can be described as a collection of discrete atomic actions. The most general approach is to assume that the temporal ordering among these atomic actions is a partial order. However, a partial order is equivalent to the set of all total orders that are consistent with it. It turns out that as long as one is concerned only with safety and liveness properties, no information is lost by replacing a partially ordered set of events by the set of all sequences obtained by extending the partial order to a total order. Thus, we can consider a behavior to be a sequence of actions. Concurrency appears as nondeterminism—if two actions are concurrent, then the set of possible behaviors contains sequences in which they are performed in either order.

A formalism based upon sequences may be inadequate for discussing other properties of the system's behavior, such as whether two actions occur concurrently. While such properties may be of interest when analyzing a given system, I have not found them to be relevant to the system's specification.

The transition axiom method specifies the atomic actions comprising each operation. Can one specify an operation without stating what the atomic actions are?

The transition axiom method can be extended to allow the specification of nonatomic operations—that is, operations composed of an unspecified number of atomic actions. Writing such a specification is easy; for example, we can just remove the angle brackets from Figure 2. However, it is not so easy to say precisely what such a specification means and how one verifies the correctness of an implementation. The transition axiom method can be extended to handle nonatomic operations by introducing the formal concepts described in [11] and [10].

4.3 Modularity and Hierarchical Decomposition

Can one hierarchically decompose transition axiom specifications?

There are two kinds of hierarchical decomposition: (1) decomposition within a single level of abstraction, and (2) representation of a higher-level system as a composition of lower-level ones. The second kind of decomposition involves a change in the grain of atomicity—usually a single atomic action is decomposed as a set of lower-level actions; the first does not.

Decomposition within a single level involves organizing the information contained in a single specification to make it easier to understand. For example, one can decompose a transition axiom by writing it as a conjunction of several relations, where each conjunct is described separately. This type of decomposition is a language design issue that raises no basic logical questions.

Representing a higher-level system as the composition of lower-level systems means implementing the higher-level system with the lower-level composite system. The implementation of one system with a lower-level one has already been discussed here at considerable length.

How are specifications of individual components combined to specify a single system?

Formally, to say that a system M is the composition of two systems M_1 and M_2 means that the specification of M, which is a temporal logic formula, is the conjunction of the specifications of M_1 and M_2. In the transition axiom method, the specification of M is obtained by simply combining the specifications of M_1 and M_2. The state functions in the specification of M

consist of the state functions from the specifications of both M_1 and M_2, and the set of actions of M is the union of the sets of actions of M_1 and M_2.

Combining specifications may necessitate some renaming. Internal state functions and actions may have to be renamed to avoid conflicts, since an internal state function of M_1 represents a different state function from any internal state function of M_2. Also, the act of combining M_1 and M_2 may imply a renaming or identification of interface state functions. Suppose M_1 is a circuit with an interface state function named *output* that represents the voltage on its output wire and M_2 is a circuit with an interface state function named *input* the represents the voltage on its input wire. Connecting the output wire of M_1 to the input wire of M_2 implies that *input* and *output* become two names for the same state function.

Acknowledgments

I wish to thank Amir Pnueli for teaching me what existential quantification over state functions means, Jim Horning and John Guttag for numerous helpful discussions, and the following for their comments on earlier versions of this paper: Jim Gray, Brent Hailpern, Luigi Logrippo, Jay Misra, Susan Owicki, Willem-Paul de Roever, Fred Schneider, Fritz Vogt, Jeanette Wing, and Pamela Zave.

References

[1] M. W. Alford et al. *Distributed Systems: Methods and Tools for Specification*, chapter 5. *Lecture Notes in Computer Science, 190*, Springer-Verlag, 1985.

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

[3] 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.

[4] 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.

[5] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–222, June 1981.

[6] 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.

[7] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[8] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[9] Simon S. Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, July 1984.

[10] Leslie Lamport. win *and* sin*: Predicate Transformers for Concurrency*. Research Report 17, Digital Equipment Corporation, Systems Research Center, May 1987.

[11] Leslie Lamport. Reasoning about nonatomic operations. In *Proceedings of the Tenth Annual Symposium on Principles of Programming Languages*, pages 28–37, ACM SIGACT-SIGPLAN, January 1983.

[12] Leslie Lamport. 'Sometime' is sometimes 'not never': a tutorial on the temporal logic of programs. In *Proceedings of the Seventh Annual Symposium on Principles of Programming Languages*, pages 174–185, ACM SIGACT-SIGPLAN, January 1980.

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

[14] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, IFIP, North Holland, Paris, September 1983.

[15] 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.

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

[17] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, 1980.

[18] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[19] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[20] J. Wensley et al. SIFT: design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1254, October 1978.

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

Index