Op. 69

An Axiomatic Semantics of Concurrent Programming Languages

Leslie Lamport

19 September 1984

To appear in the lecture notes of the Advanced Seminar on Logics and Models for Verification and Specification of Concurrent Systems, held in La-Colle-Sur-Loup, France in October, 1984.

Work Supported in part by the National Science Foundation under grant number MCS-8104459 and by the Army Research Office under grant number DAAG29-83-K-0119.

Contents

1 An Introduction to this Paper **1**

2 An Introduction to Semantics **3**
2.1 What Are Semantics? 3
2.2 Different Kinds of Semantics 6
2.2.1 Behavioral Semantics 6
2.2.2 Action Semantics 7
2.2.3 Action-Axiom Semantics 8
2.3 Is This Fair? 9
2.4 Programs and Implementations 11
2.4.1 Correctness of an Implementation 11
2.4.2 The Interface 13

3 The Programming Language **16**

4 States and Actions **21**
4.1 Program Variables 21
4.2 Control Variables 23
4.3 Are There Other State Components? 25
4.4 Renaming 26
4.5 Actions 27
4.6 States and Actions: A Formal Summary 28

5 Temporal Logic **30**
5.1 Predicates 30
5.2 The Unary Temporal Operators 31
5.3 The Binary Temporal Operators 32
5.4 Renaming 34
5.5 Temporal Logic as Semantics 34
5.6 There Won't Be a Next Time 35
5.7 Implementation Mappings 36

6 The Semantics of Language L **39**
6.1 Syntactic Predicates 39
6.1.1 Aliasing 41
6.1.2 Syntactic Typing Relations 44
6.1.3 Reasoning About Syntactic Expressions 44
6.1.4 Logical Name Variables 45

6.2 Starting States 45
6.3 Behavior Axioms 46
6.3.1 Assignment 47
6.3.2 The **if** Statement 49
6.3.3 The **while** Statement 49
6.3.4 The **new** Statement 50
6.3.5 The **cobegin** Statement 51
6.3.6 Sequences of Statements 51
6.3.7 A Complete Program 51

7 Other Language Features **52**
7.1 Constructs That Constrain Their Environment 52
7.1.1 The **assign processor** Command 52
7.1.2 Atomic Actions 52
7.1.3 Write Protection 53
7.2 Synchronization and Communication 53
7.2.1 Semaphores 53
7.2.2 CSP-Like Communication Primitives 54
7.3 Procedures 55
7.4 More General Types and Aliasing 56
7.5 Nonatomic Operations 57

8 Conclusion **59**

1 An Introduction to this Paper

A large body of research on the logic of concurrent programs may be characterized as the "axiomatic" school. Members of this school reason about safety properties ("something bad never happens") in terms of invariance, and liveness properties ("something good eventually does happen") using temporal logic.

While they are quite successful at proving properties of a given program, axiomatic methods have not provided a satisfactory semantics for concurrent programming languages. Axiomatic methods usually reason about the entire program, while a semantics should be compositional—deriving the meaning of a program from the meanings of its components. Even the Generalized Hoare Logic described in [4] and [9], which looks compositional, actually assumes a context of a complete program. The only attempt we know of at a truly compositional axiomatic semantics for concurrent programs that handles both safety and liveness properties is given in [14]. However, while it is axiomatic in a strict logical sense, that approach is not in the spirit of the axiomatic school because it essentially defines a new temporal logic operator for every programming-language construct.

In this paper, I present a new compositional, truly axiomatic semantics for concurrent programming languages. It is based upon temporal logic, but employs five fundamental ideas beyond those found in most temporal logic methods:

1. The addition of action predicates to describe "who" performs an action.

2. Defining an assertion to be true of a statement only if it is true of every program containing that statement.

3. The introduction of renaming operations that map an assertion about a statement S into an assertion about a larger statement containing S as a substatement.

4. Defining the relations between control points, described in [4] as state predicates, to be aliasing relations among variables.

5. Allowing "stuttering" actions, so an atomic operation is represented by a finite sequence of actions, only the last one having any effect.

The first idea was developed by Susan Owicki and myself in the late 1970's, and was published in [6]. (It was developed independently, in different contexts, by other researchers [1].) The second and third ideas were developed by me shortly afterwards. The second was also used in [6], though not featured prominently there. The third idea has never appeared in print, though I have talked about it in lectures starting in 1981. The fourth idea was discovered by Fred Schneider and myself in the spring of 1984 [10]. The fifth idea has been present in all of my work on temporal logic, starting with [5]. I was originally led to it by my philosophical objections to the "next time" operator; only later did I recognize its practical significance [8].

The first two ideas were used in [14], but they are not enough to permit a compositional semantics based upon a simple temporal logic. Combined with the third idea, they do permit a compositional semantics, but a semantics that I did not find satisfying. It seemed like a large, complicated structure had to be erected solely to reason about program control, making the enterprise of dubious merit. It was the fourth idea that gelled the method into a coherent form. The apparatus for handling program control was no longer an *ad hoc* "Kludge". Rather, it was the appropriate structure to deal with aliasing. Aliasing was not considered in other approaches, but it is a problem that must be dealt with in any realistic language, if only to handle procedure calls. The fifth idea is not needed for the semantics itself; in fact the semantics would be somewhat easier to understand had I abandoned it and employed the next-time operator favored in most other temporal logic approaches. However, allowing stuttering actions enables the semantics to address the practical issue of what it means for a compiler to be correct.

In this paper, I develop these five ideas, and show how they lead to a method for defining the semantics of concurrent programming languages. A complete semantics is given only for a simple language. However, the approach is "meta-compositional" in the sense that the meaning of each language construct is defined independently of the other constructs in the language. The semantics of a richer language can be given by defining the meanings of its additional constructs, without changing the meanings of the constructs from the simple language. (This is not the case in [14] where, for example, the axioms for the assignment statement would be invalid if an unfair **cobegin** were added to the language.) I will indicate the power of my method by informally describing how the meanings of some more complicated language constructs can be defined.

2 An Introduction to Semantics

2.1 What Are Semantics?

The syntax of a programming language defines the set of syntactically well-formed programs of that language. However, a program is more than just a string of characters; there should be a well-defined set of possible results of executing the program. The purpose of a semantics is to assign a mathematical *meaning* to each syntactically correct program that describes the effect of executing it.

I will regard a program Π to be a syntactic object, and denote by $\mathcal{M}[\![\Pi]\!]$ the mathematical object denoting its meaning. To define a formal semantics, one must specify the mapping $\Pi \to \mathcal{M}[\![\Pi]\!]$.

What is the purpose of a formal semantics? One purpose is to help us to understand the language. However, "understanding" is too vague to usefully characterize a formalism. I propose that a formal semantics should provide a formal basis for the following:

1. Deducing properties of a program written in the language.

2. Deciding if a compiler is correct, given a formal semantics for the target language into which the programs are compiled.

A semantics should provide a formal foundation, but not necessarily a practical method, for doing these things. A method for deducing properties of a program is called a *proof system*. A proof system is used to decide if a program works properly; a semantics is used to decide if a programming language is defined properly. One wants to reason about programs at a high level, hiding as much detail of the language as possible; a semantics should expose the language details. Although a semantics allows one, in principle, to verify properties of programs, its real purpose is to explain the language. A semantics should be used to verify the correctness of a proof system; it need not provide a practical method for reasoning about programs.

Programs can be very large and complicated. We want to reduce the problem of understanding a complex program to that of understanding its components. The meaning of a program should therefore be defined in terms of the meanings of its components. We must therefore define the meaning not just of an entire program, but of individual components—usually individual program statements. So, $\mathcal{M}[\![S]\!]$ must be defined for any program statement S, and it must be defined in terms of the substatements of S. For example, $\mathcal{M}[\![S_1; S_2]\!]$ should be defined in terms of $\mathcal{M}[\![S_1]\!]$ and $\mathcal{M}[\![S_2]\!]$. I will say that a semantics with this property is *compositional*.[1]

When defining a formal semantics, the first thing one has to decide is what kind of object $\mathcal{M}[\![S]\!]$ should be. The execution of a statement in a sequential program is usually considered to start in some input state and produce an output state, and $\mathcal{M}[\![S]\!]$ is defined to be a mathemtical object that describes the relation between the input and the output states. One way of doing this is to define $\mathcal{M}[\![S]\!]$ to be a set of ordered pairs of states.

Concurrent programs cannot be described with such a simple input/output semantics. Consider the following two program statements, where angle brackets denote indivisible atomic operations.

1. $\langle\, x := x + 1 \,\rangle$

2. **begin** $\langle\, x := x + y \,\rangle$;
   $\langle\, x := x - y + 1 \,\rangle$
   end

These statements both have the same relation between input and output states—they both increment the value of x by one. However, they are not equivalent when used as part of a concurrent program. Executing the first always has the effect of adding one to x, but executing the second can have a very different effect if the value of y is changed by some other process between the two assignments to x.

A semantics for a concurrent programming language must define the meaning of a statement in terms of its behavior. There are two fundamentally different approaches to doing this. The first approach is to define the meaning of a statement S in terms of the effects it produces that are "visible" outside S. For example, in a shared-variable language, the only visible effects of executing a statement are changes to shared variables. The alternative approach to defining the semantics of concurrent languages defines the meaning of a statement in terms of complete behaviors, which include all the effects of a statement's actions, whether externally visible or not. In most languages, these invisible effects include changes to the control state (the values of "program counters").

An approach that mentions only visible effects is very appealing, and it has been taken by a number of researchers [3, 11]. For many years, I regarded it as the proper way to think about programs, and found it

[1]The terms *denotational*, *syntax-directed*, and *modular* have also been used to denote this property.

unnatural to reason about things like the control state that are internal to the program. However, years of experience reasoning about concurrent programs has led me to conclude that one should think about them in terms of the complete state, including externally invisible compents of the state. I will not attempt to justify this conclusion here, and will simply adopt the second approach, defining the meaning of a program in terms of complete behaviors that describe the internal as well as the externally-visible effects of program operations.

Having decided that the meaning of a program is its set of possible behaviors, we must decide what a behavior is. The simplest notion of a behavior is a sequence of states. Each action of the program transforms the state. Nondeterminism, leading to sets of behaviors, appears when there are several choices of a possibile next state from the same current state.

It is sometimes argued that a sequence of states cannot adequately model the execution of a concurrent program because it has no notion of concurrent activity, and that one should instead use a partially ordered set of actions. However, a partially ordered set contains exactly the same information as the set of all total orderings consistent with the partial order. Since the meaning of a statement is the *set* of behaviors, which includes all possible sequences that represent the real, partially ordered set of actions, nothing has been lost by considering sequences. The basic assumptions being made are that the execution of a program consists of discrete atomic actions, and the possible effect of an atomic action depends only upon the current state. It appears that any digital system can be accurately modeled in this way by making the atomic actions small enough and including enough information in the current state.

It turns out that to define the semantics of concurrent languages, one needs more information about a behavior than just the sequence of states; one must also know "who" performed the actions. For example, the natural definition of a fair **cobegin** states that in each infinite behavior, every nonterminating process performs infinitely many actions. Formalizing this definition requires the ability to decide which process performs each action. I will therefore define a behavior to be a sequence of the form:

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots$$

where the s_i are states and the α_i are actions. Fairness of a **cobegin** can be expressed by stating that for every process and every n: if there is no state s_i with $i > n$ in which the process has terminated, then infinitely many of the α_i are actions of that process. I will explain later exactly what states and actions are.

The semantics that I am aiming for is an axiomatic one, in which the meaning of a program is a set of axioms in a formal system. An important advantage of an axiomatic semantics is that it is very formal. A formal mathematical system is one in which reasoning can be reduced to a strict application of axioms and inference rules. Automated deduction systems can usually be applied only to a formal system. A semantics in which $\mathcal{M}[\![S]\!]$ is defined to be a set of sequences is really semi-formal, based upon the informal mathematical concepts of sets and sequences. Formalizing it requires formalizing these mathematical concepts. With an axiomatic semantics, this extra step is unnecessary; $\mathcal{M}[\![S]\!]$ is already a set of axioms in a formal system.

The problem with an axiomatic semantics is that one can understand the meaning of a formal logical system only by constructing a semantic model for it in terms of concepts that we already understand. Having constructed a semantics in which $\mathcal{M}[\![S]\!]$ is a set of axioms in some formal system is only half the job; we also have to define a semantics for the formal system in terms of well-understood mathematical concepts.

I will give a temporal logic semantics—one in which the axioms are temporal logic formulas. I will rely upon the usual semantic model of temporal logic, described later, to provide a basis for an intuitive understanding of the axioms.

2.2 Different Kinds of Semantics

2.2.1 Behavioral Semantics

An obvious method of defining a semantics for concurrent programs is to let the meaning of a statement be its set of possible behaviors, and to explicitly construct the behaviors in $\mathcal{M}[\![S]\!]$ from the behaviors of its components. For example, the set of behaviors $\mathcal{M}[\![S_1\,;\,S_2]\!]$ consists of all infinite behaviors in $\mathcal{M}[\![S_1]\!]$ together with all concatenations of finite behaviors in $\mathcal{M}[\![S_1]\!]$ with behaviors in $\mathcal{M}[\![S_2]\!]$. This can be expressed formally by:

$$\begin{aligned}\mathcal{M}[\![S_1\,;\,S_2]\!] \quad = \quad & \{\sigma \in \mathcal{M}[\![S_1]\!] : \sigma \text{ infinite}\} \\ & \quad \cup\ \{\sigma\tau : \sigma \in \mathcal{M}[\![S_1]\!],\ \tau \in \mathcal{M}[\![S_2]\!],\ \text{and } \sigma \text{ finite}\}\end{aligned}$$

I will call such a semantics a *behavioral* semantics.

Behavioral semantics have their problems. While they work well for sequential programming constructs, they are less satisfactory for concurrent languages. The behaviors of

cobegin $S_1 \square S_2$ **coend**

are obtained by forming interleavings of behaviors from S_1 and S_2, and interleavings are rather awkward mathematically—especially for a fair **cobegin**, where only fair interleavings are allowed.

A more serious problem is raised by the language construct

assign processor to S

Intuitively, this statement causes the compiler to assign a physical (or virtual) processor to execute S. In terms of behaviors, it means that any behavior that reaches S must either subsequently reach the end of S or else include an infinite number of actions of S. In other words, a process cannot be "starved" while it is executing this statement. This is a perfectly reasonable—and compilable—statement. It can be used to construct a fair **cobegin** from an unfair one as follows:

unfair cobegin **assign processor to** S_1 $\square$
assign processor to S_2 **coend**

More complicated uses of the **assign processor** statement are also possible.

Considered completely by themselves, the statements S and

assign processor to S

have the same sets of behaviors, so it is not clear how one could apply to the **assign processor** statement the same approach used above to define $\mathcal{M}[\![S_1; S_2]\!]$.

2.2.2 Action Semantics

Instead of defining $\mathcal{M}[\![S]\!]$ to be the set of behaviors itself, one can define it to be something that can be used to construct the set of behaviors. Since a behavior is generated by a sequence of actions starting in some state, an obvious approach is to let $\mathcal{M}[\![S]\!]$ be the set of all possible actions together with the set of all possible starting states. I will call such a semantics an *action* semantics. Given an action semantics, one can define the behaviors of S to be the set of all behaviors that can be obtained from these actions starting from the specified starting states.

An action semantics is well-suited to expressing parallelism, since the set of possible actions of

cobegin $\pi_1 \,\square\, \pi_2$ **coend**

is just the union of the sets of possible actions of π_1 and π_2. Action semantics have long been favorites of theoretical computer scientists [15] because they lead to mathematically well-behaved formalisms. Unfortunately, these semantics are unsatisfactory because they cannot express fairness. Consider a coin-flipping program with two possible actions: toss a head and toss a tail. It can be viewed as the parallel composition of two processes: one that generates only heads and the other that generates only tails. An unfair coin flipper can generate any infinite sequence of heads and tails, while a fair one can generate only sequences containing infinite numbers of both heads and tails. Both the fair and the unfair coin flipper have the same set of actions (toss a head and toss a tail), so an action semantics cannot distinguish between the two.

2.2.3 Action-Axiom Semantics

The problem of fairness is solved by using an *action-axiom* semantics in which the meaning of a statement consists of a set of actions together with a set of temporal logic axioms that state conditions under which an action must eventually occur. For example, the fair coin flipper requires two axioms:

- At any time, a head must eventually occur.
- At any time, a tail must eventually occur.

The meaning of

assign processor to S

consists of the meaning of S plus the following additional axiom:

- If S is being executed—more precisely, if control is in S—then an action of S must eventually occur.

We are thus led to let $\mathcal{M}[\![S]\!]$ consist of a set of actions, a set of temporal logic axioms, and a set of starting states. But how do we specify the actions? Instead of introducing some new method for specifying actions, I will specify the actions as well as the fairness properties with temporal logic axioms. Starting states will be specified by ordinary, nontemporal axioms.

To give a compositional action semantics, we must define the axioms of $\mathcal{M}[\![S_1; S_2]\!]$ in terms of the axioms of $\mathcal{M}[\![S_1]\!]$ and $\mathcal{M}[\![S_2]\!]$; and, of course, we must also do the same for other language constructs besides the "$;$". We will see that there is a standard prescription for doing this.

The axioms in $\mathcal{M}[\![S]\!]$ define a set of behaviors for S—namely, the set of all behaviors satisfying the temporal logic axioms starting in states that satisfy the axioms for the starting state. Although this defines a set of behaviors for every statement, it is different from a semantics in which $\mathcal{M}[\![S]\!]$ is taken to be a set of behaviors because the meaning of S is obtained from the meaning of its components by "composing" axioms, not by composing behaviors.

2.3 Is This Fair?

It can be argued that the semantics of a programming language should be defined in terms of constructive operations rather than with axioms. One should give a procedure for constructing the set of behaviors of a program rather than a set of axioms to describe it.

While a purely constructive approach would be nice, it seems to be impossible to deal with fairness constructively. Even a behavioral semantics, which looks constructive, really includes axioms for fairness. A behavioral semantics defines the meaning of a fair **cobegin** in terms of fair interleaving. The definition of a fair interleaving of two behaviors goes something like this:

> Construct all interleavings and then throw away the ones that do not satisfy the fairness condition.

This is remarkably similar to the definition of the set of behaviors obtained from a set of actions and a set of constraints, which can be expressed as:

> Construct all behaviors generated by the set of actions and then throw away those that do not satisfy the constraints.

One might argue that fair interleaving is a simple, basic concept, and I have given a particularly jaundiced expression of it. However, there are many different fairness constraints one might want to define, each of which would require a different definition of fair interleaving. For example, consider two coin-flipping processes, one with the single action *head* and the other with two actions: *tail* and *coin lost*. The first process generates only the sequence of all heads, the second process generates either a sequence of all tails or a finite string of tails followed by a sequence of *coin lost* actions. The behaviors resulting from executing the two processes concurrently are defined to consist of all possible fair sequences of heads and tails plus all sequences consisting of a finite number of heads and tails followed by nothing but *coin lost* actions.

This is a perfectly reasonable example, which the reader may find more familiar if he replaces *coin lost* by *abort program*. A behavioral semantics for this way of combining processes would require a more complicated definition of fair interleaving, and a formal statement of this definition would look a lot like a temporal logic axiom. Fair interleaving is not a simple concept. One particular type of fair interleaving has been used so commonly that we tend to take it for granted and forget that we have never seen a constructive definition of it.

Fairness does not appear to be a constructive concept. One specifies fairness by adding axioms to exclude unfair behaviors rather than by explicitly constructing only the fair ones. Infinite objects, such as behaviors, are constructed as limits of finite approximations—a method often described as "denotational". This does not work with fairness because there exist sequences of fair behaviors whose limits are unfair—for example, let $\sigma_1, \sigma_2, \ldots$ be the sequence of coint-flipping behaviors in which all actions of σ_n are heads, except for every $2^n th$ action, which is a tail. Each σ_n is fair, but the limit as n goes to infinity is the behavior having only heads, which is unfair.

The topological approach of [2] solves this problem by considering only convergent sequences and defining a topology in which sequences like the above diverge. However, one might view this approach as:

> Construct all sequences obtainable from the actions and throw away those that do not converge.

This looks suspiciously like the more overtly axiomatic approach. The whole distinction between constructive and axiomatic methods is probably illusory, disappearing when methods are examined closely enough.

2.4 Programs and Implementations

2.4.1 Correctness of an Implementation

One question that a semantics of a programming language should answer is: What does it mean for a compiler to be correct? Given a program Π in the high-level language, the compiler transforms it into a program π in some lower-level language. Correctness of the compiler means that π is a correct implementation of Π, but what does that mean? To speak of correctness, we must have formal semantics for both the high-level and the low-level languages, so $\mathcal{M}[\![\Pi]\!]$ and $\mathcal{M}[\![\pi]\!]$ are defined. However, this is not enough to determine what it means for $\mathcal{M}[\![\pi]\!]$ to represent a correct implementation of $\mathcal{M}[\![\Pi]\!]$.

Consider the case of sequential programs, in which the semantics of a program is a relation on the set of program states, the pair (s, t) being in the relation $\mathcal{M}[\![\Pi]\!]$ if and only if it is possible for program Π to start in state s and terminate in state t. In this case, $\mathcal{M}[\![\Pi]\!]$ and $\mathcal{M}[\![\pi]\!]$ are relations on two different sets of states. The states of Π specify the values of program variables like x and y; the states of π might specify the values of machine registers like *memory location* 3124 or the *program counter*. Correct implementation means that there is a correspondence between the sets of states of Π and π such that, under this correspondence, every possible execution of π is a possible execution of Π.

More formally, to establish a correspondence between the semantics of the two sequential programs, we must define a mapping F from the states of π to the states of Π. For example, suppose the variable x in Π of type *integer* is implemented in π as a two-byte integer stored in bytes 3124 and 3125 of memory. If, in a state s of π, bytes 3124 and 3125 have the values 12 and 97, then the value of x in the state $F(s)$ of Π is $12 \times 256 + 97$. In general, correctness of the implementation means that for each pair (s, t) in $\mathcal{M}[\![\pi]\!]$, the pair $(F(s), F(t))$ must be in $\mathcal{M}[\![\Pi]\!]$.

What about concurrent programs? As we have seen, the meaning of a concurrent program must be expressed in terms of its behavior—either directly, with a behavioral semantics, or indirectly with axioms about its behavior. Let us therefore consider first a behavioral semantics, in which $\mathcal{M}[\![\Pi]\!]$ and $\mathcal{M}[\![\pi]\!]$ are sets of behaviors. Intuitively, π is a correct implementation of Π if every possible behavior of π represents a possible behavior of Π. We therefore need some way of interpreting behaviors of π as possible behaviors of Π—that is a mapping F such that for any behavior σ in $\mathcal{M}[\![\Pi]\!]$,

$F(\sigma)$ is a sequence of states and actions of Π. We can then say that π is a correct implementation of Π if, for every σ in $\mathcal{M}[\![\pi]\!]$, $F(\sigma)$ is in $\mathcal{M}[\![\Pi]\!]$.

In defining this mapping F, we are faced by the problem that Π and π may have different grains of atomicity. An atomic operation of Π may be implemented by a sequence of 42 atomic operations of π. For example, the atomic operation

$$\langle\, x := x + 1 \,\rangle$$

of Π might be implemented in π by 42 machine-language operations. Moreover, interleaved among these 42 atomic operations of π might be other machine-language operations that belong to the implementation of an operation from a different process of Π. It would therefore seem that the mapping F must be quite complicated, taking sets of actions into single actions.

There is very simple solution to this problem—we require that to every action of π there correspond a single action of Π. The execution of a single atomic operation of Π might therefore be represented by 42 actions in a behavior in $\mathcal{M}[\![\Pi]\!]$. The first 41 of these actions will be "stuttering" actions that do not change the state of Π; the 42^{nd} will do all the work. This makes it conceptually very easy to define the mapping F from behaviors of π to behaviors of Π. As in the sequential case, there must be a mapping F from states of π to states of Π. We also assume that F maps actions of π to actions of Π—for example, every machine-language instruction executed by π corresponds to the execution of some atomic operation of Π.[2] To extend F to a mapping on behaviors, if σ is the behavior

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots$$

of π, we define $F(\sigma)$ to be the behavior

$$F(s_0) \xrightarrow{F(\alpha_1)} F(s_1) \xrightarrow{F(\alpha_2)} \cdots$$

The implementation is correct if, for every behavior σ of $\mathcal{M}[\![\pi]\!]$, $F(\sigma)$ is a behavior in $\mathcal{M}[\![\Pi]\!]$.

This seems nice in theory, but how can it be achieved in practice? The first 41 machine-language operations in the implementation of the atomic

[2] A single machine-language instruction could actually be used in the implementation of several atomic actions of Π—for example, if it were part of a subroutine called during the execution of several different statements of Π. The mapping F should therefore take state, action pairs into actions, so the action α_i of π is mapped into the action $F(s_{i-1}, \alpha_i)$ of Π. In other words, the state of π determines which atomic statement of Π is being executed by the execution of a machine-language statement.

assignment must change the state in such a way that they these changes are invisible when viewed at the higher level. More precisely, the 41 intermediate states of the computation must all be mapped by F into the same state as the starting state. How is this possible?

A complete answer to this question is beyond the scope of this paper. The trick lies in the definition of F, which must "unscramble" the intermediate states in the appropriate way. I will not explain here how it is done. I will only mention that, while it sounds like magic, it in fact is a simple extension of the basic idea of invariance that underlies most concurrent program verification. An explanation and examples can be found in [6] and [8].

I won't consider the problem of compiler correctness. The purpose of this discussion is to point out that in order to permit a simple definition of correctness of an implementation, I cannot define a semantics in which the execution of an atomic program statement is always represented as a single atomic action. I must allow "stuttering" actions. In the action-axiom semantics, the specification of an action α must allow a finite series of null transitions $s \xrightarrow{\alpha} s$ as well as the final action $s \xrightarrow{\alpha} t$ that "does the work".

I have described correctness of an implementation in terms of a behavioral semantics, where $\mathcal{M}[\![\Pi]\!]$ is a set of behaviors. In an action-axiom semantics, the meaning $\mathcal{M}[\![\Pi]\!]$ of a program Π is a set of axioms that determines the set of possible behaviors. Section 5.7 explains how this concept of correctness is translated into a relation between the sets of axioms $\mathcal{M}[\![\Pi]\!]$ and $\mathcal{M}[\![\pi]\!]$. The only observation I will make here is that the axioms of $\mathcal{M}[\![\Pi]\!]$ must permit stuttering actions. More precisely, these axioms should not be able to distinguish stuttering; if an axiom is true for a behavior σ, then it should also be true for the behavior obtained from σ by adding stuttering actions. This will be guaranteed by using a temporal logic in which no formula can distinguish stuttering—a temporal logic with no "next-time" operator.

2.4.2 The Interface

The semantics of a program is traditionally defined by describing how it affects the values of variables. However, program variables are internal to the program; all that a user sees is what he types into the program and what the program types out to him. A semantics of a program should describe its input and output, not just how it affects internal objects like variables.

Given the machine-dependence of most input and output, an explicit semantics for input and output seems like a useless exercise. Instead, observe that input and output can be representend by variables. A terminal screen can be represented as a Boolean array, each element representing the presence or absence of light at one point on the screen. Keyboard input can be simulated through a variable whose value represents the sequence of characters that have been typed but not yet processed. I will use the term *interface variables* to describe variables that represent input and output.

In general, an interface variable describes the interaction between the program and its environment. They are *global* or *free* variables, in contrast to the *local* or *bound* variables that are declared in ordinary program declarations. For example, variables declared in a Pascal **var** declaration are local.

Let us again consider the mapping F, introduced above to define what it means for a lower-level program π to be correct implementation of a higher-level program Π. Recall that F describes how the variables of Π are implemented in terms of the "variables" of π—the machine registers, if π is a machine-language program. We really don't care how the local variables of Π are implemented, since they are not externally visible. The compiler is free to implement local variables any way it wishes.

The compiler does not have such freedom in its implementation of interface variables. The implementation of the interface variables must be defined *a priori* if the program is to interact with its environment in a useful way. For example, suppose that the terminal screen is represented by a Boolean array. The semantics of the program Π would provide no information about real output if the compiler could define the array elements to represent completely arbitrarily points on the screen—or to represent the values of arbitrary one-bit registers in the machine.

I have considered the implementation of the states of Π in terms of states of π, but what about the implementation of actions? Just as there are local and interface state functions, there are internal and external actions. Most actions in a program behavior are internal, being caused by program execution. However, some actions represent operations external to the program—for example, the actions that represent the entering of an input character. The semantics of Π does not distinguish this operation, which changes the value of the interface variable representing the input buffer, from program operations that change the value of variables—for example, the program operation that removes a character from the input buffer. The compiler is free to implement internal actions of Π by any internal actions of π. However, the external actions of Π must be implemented by fixed actions of π, which may be internal or external. For example, the sequence of actions of Π that add a character to the input buffer may be implemented by a sequence of actions external to π, representing external operations that put the character into an input register and actions of π that move the character from the input register into the memory registers that implement the input buffer. The compiler would be of little use if it could implement the operation of typing a character, defined in the semantics of Π simply as an operation that changes the variable representing the input buffer, as an internal operation of π that adds a randomly chosen character to the buffer.

Thus, the representation of local variables and internal actions of Π by F may be arbitrary, but the representation of interface variables and external actions must be fixed. The meaning $\mathcal{M}[\![\Pi]\!]$ of Π can be defined in a completely machine-independent fashion. The machine dependency, which exists for any real compiler, is contained in the details of how interface variables and external actions are to be implemented.

Thus far, I have been talking only about implementing a complete program Π. We should consider the problem of implementing a single statement S. In this case, all the global (undeclared) variables of S must be regarded as interface variables, and their implementations must be fixed *a priori*. For example, if statements S and T were to be implemented independently, their implementations could be combined to implement $S; T$ only if a variable x common to both were implemented as the same set of machine registers.

Of course, one is seldom interested in implementing a single statement of a program. These considerations would apply to a language that allows separate compilation of components such as subroutines. I will not consider the problem of separate compilation. My purpose in discussing implementation of individual statements is to point out that the concept of global and local variables occurs at all levels of a program. Variables global to a statement S may be local to a larger statement containing S.

3 The Programming Language

The goal of this paper is to explain how the semantics of any programming language can be defined, and not to give a complete semantics for aparticular language. However, to show how the formalism works, it is helpful to define rigorously the semantics of some language. I will therefore formally define the semantics of a simple language called L, and will indicate informally how the semantics of language primitives other than those in L can defined.

The language L contains an atomic assignment statement—one whose execution is an indivisible, atomic action. L has the usual sequential control structures: concatenation (;), **if** and **while** statements, plus a fair **cobegin**. The tests in **while** and **if** statements are also taken to be atomic. L has a **new** statement that declares a local variable, so

$$\textbf{new } x \textbf{ : integer in } S \textbf{ ni}$$

declares x to be a local variable of type **integer** whose scope consists of the statement S. The **new** statement has an optional **init** clause to specify the initial value, so

$$\textbf{new } x \textbf{ : integer init } 2 * x \textbf{ in } S \textbf{ ni}$$

declares that the initial value of x in S is twice the value of the variable x whose scope includes the **new** statement. The assignment of the initial value to x is assumed to be an atomic action. The **new** statement also has an optional **alias** clause that is used to declare that the new variable is the alias for something else. For example,

$$\textbf{new } x \textbf{ : integer alias } y \textbf{ in } S \textbf{ ni}$$

declares x to be an alias for y.

It may seem strange to introduce aliasing—a concept usually ignored in simple examples—in the language L. Aliasing is an important concept because it underlies the semantics of procedure calls. If *proc* is a procedure defined with single integer-valued a call-by-name parameter *param*, then the call *proc*(*arg*) can be simulated by the statement

$$\textbf{new } \mathit{param} \textbf{ : integer alias } \mathit{arg} \textbf{ in } S \textbf{ ni}$$

where S is the body of *proc*. (Call by value and call by reference can be simulated with call by name through the use of auxiliary variables.)

For reasons that will be clear later, the concept of aliasing is central to our semantics, and we will need to understand a more general kind of aliasing than real programming languages usually allow. In particular, L will allow a variable to be aliased to an expression. To understand what that means, consider the declaration

$$\textbf{new } f : \textbf{real} \;\; \textbf{alias } 9 * c/5 + 32 \;\; \textbf{in } S \textbf{ ni}$$

In this case, we can think of f and c as representing a single temperature, where f is its value in degrees Farenheit and c is its value in degrees Celsius. The two assignment statements $f := 32$ and $c := 0$ have exactly the same effect; executing either one changes the value of f to 32 and the value of c to zero.

As another example, assume a type **gaussian** which represents a Gaussian integer—a number of the form $m + n\sqrt{-1}$, where m and n are integers. If x and y are variables of type **integer**, then

$$\textbf{new } z : \textbf{gaussian} \;\; \textbf{alias } x + y * \sqrt{-1} \;\; \textbf{in } S \textbf{ ni}$$

defines z to be a variable of type **gaussian** whose real part is aliased to x and whose imaginary part is aliased to y. Assigning a value to z in S also assigns values to x and y so that the relation

$$z = x + y * \sqrt{-1}$$

holds throughout the execution of S. Similarly, changing the value of x in S also changes the value of z.

In the examples of **alias** clauses given so far, assigning a value to any variable produces a well-defined result. However, this need not be the case. Inside the body S of the statement

$$\textbf{new } c : \textbf{integer} \;\; \textbf{alias } a + b \;\; \textbf{in } S \textbf{ ni}$$

assigning a value to a or b changes the value of c in the obvious way, but what is the result of assigning a value to c? I define an assignment to c to be a nondeterministic statement that can change the values of a and b in any way such that $a + b$ equals the new value of c.

However, I will assume that the aliasing relations are such that they can always be maintained by the proper choice of values. More precisely, a program is considered illegal if its execution would force the aliasing relations to be violated. For example, the statement

$$\textbf{new } b : \textbf{integer} \;\; \textbf{alias } \sqrt{a} \;\; \textbf{in } S \textbf{ ni}$$

is illegal if, at any time during its execution, the value of a is not a perfect square.

This approach to aliasing is similar to the one I will take for type constraints—namely, a program is illegal if its execution would force a type violation. For example, the statement

$$\textbf{new } b \textbf{ : boolean} \;\; \textbf{init} \; \neg c \;\; \textbf{in} \; \ldots$$

is illegal in any context in which c is not declared to be of type **boolean**.

While typing consistency is easy for a compiler to enforce in language L, the consistency of aliasing relations can be determined at compile time only if the kind of expression that can appear in an **alias** is restricted in some way. In fact, some restriction is obviously necessary if the compiler is to have any chance at compiling the code. Those restrictions are irrelevant to our semantics, so they are not discussed.

The basic syntax of L is given by the syntax diagrams in Figure 1. I will not bother to give a formal syntax for identifiers. The only types that I will use in L are **integer** and **boolean**. Expressions are assumed to be the usual ones constructed from variable names and the ordinary operations on integers and booleans—for example, an expression like

$$(x * y + z = 17) \supset (x > y \vee \neg b)$$

I will enclose **if** and **while** tests, assignment statements, and the **init** clause of a **new** statement in angle brackets to emphasize their atomicity.

In addition to the usual information, the syntax diagrams of Figure 1 also have labels attached to the nonterminal components. These labels are called *primitive selectors*. A primitive selector identifies a component of a compound statement—for example, the primitive selector *then* identifies the "**then**-clause" of an **if** statement. The "..." label in the specifications of the **cobegin** indicates that the primitive selectors for the clauses of a **cobegin** are integers, and likewise for a list of statements.

In more formal terms, the primitive selectors label the edges in the parse tree of a statement or program.[3] A *selector* for a statement S is a sequence of primitive selectors that represents a path starting from the root in S's parse tree. A selector identifies a component of a program or statement. For example, the selector *else, body, 2* identifies the substatement $\langle\, x := x + 4\,\rangle$ in the following statement:

[3]Trivial nodes that have only a single son are eliminated from the parse tree, which is why there is no primitive selector associated with the first box in the syntax diagram defining a statement.

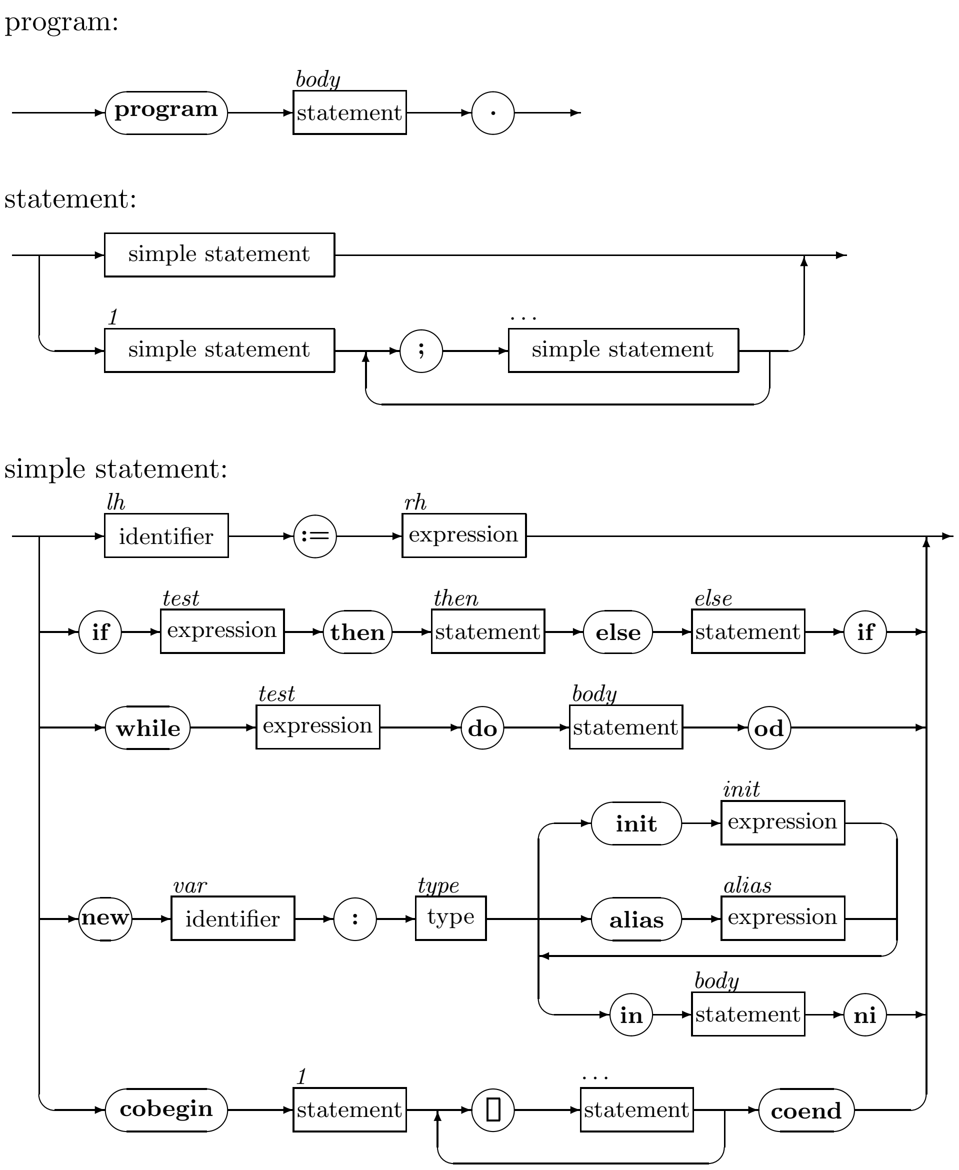

Figure 1: Basic syntax of language L.

$$
\begin{array}{l}
\textbf{if}\ \langle\, x > 0 \,\rangle \\
\quad \textbf{then}\ \langle\, x := x + 1 \,\rangle \\
\quad \textbf{else}\ \ \textbf{while}\ \langle\, y > 0 \,\rangle \\
\qquad\qquad\quad \textbf{do}\ \langle\, y := y - 1 \,\rangle; \\
\qquad\qquad\qquad\ \ \langle\, x := x + 4 \,\rangle \\
\qquad\qquad\quad \textbf{od}; \\
\qquad\qquad\ \langle\, y := 17 \,\rangle \\
\textbf{fi}
\end{array}
$$

More formally, given a program or statement S, a substatement of S consists of a pair S, γ, where γ is a selector for S. A substatement of S is, when viewed by itself, a statement. I will often write something like: "T is the substatement S, γ of S." This means that the substatement, when viewed alone, is the same as the statement T. However, T and S, γ are formally two different kinds of objects—one is a complete statement and the other is part of a statement.

The null selector selects the entire statement, so "S," denotes S viewed as a substatement of itself. Since "S," looks rather strange, I will simply write S to denote both the entire statement S and that statement viewed as a substatement of itself.

4 States and Actions

The meaning $\mathcal{M}[\![S]\!]$ of a statement S will be a set of temporal logic axioms defining the behaviors of S and a set of nontemporal axioms defining its set of initial states. To give a semantics for these axioms, I must define the the set $\mathcal{S}(S)$ of all possible states of S and the set $\mathcal{A}(S)$ of all possible actions of S. The initial-state axioms then define a set of states—namely, the set of all states in $\mathcal{S}(S)$ that satisfy those axioms; and the temporal axioms define a set of behaviors—namely, the set of all sequences

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_1} \cdots$$

with $s_i \in \mathcal{S}(S)$ and $\alpha_i \in \mathcal{A}(S)$ that satisfy those axioms.

Intuitively, the state of a statement at some time during its execution contains all the information needed to describe its possible behavior at future times. To define the set $\mathcal{S}(S)$, we must consider what information must be in the state of S.

4.1 Program Variables

The future behavior of a program certainly depends upon the current values of its variables, so a state must specify the values of all program variables. More precisely, a state in $\mathcal{S}(S)$ must include a mapping *val* from the set of variables of S to a set of values. For the simple language L, in which all variables are of type **integer** or **boolean**, the set of values consists of the set $\mathcal{Z} \cup \{\mathit{true}, \mathit{false}\}$, where $\mathcal{Z}$ denotes the set of all integers.

Let S be the statement

$$\textbf{cobegin } S_1 \,\square\, S_2 \textbf{ coend}$$

and suppose that S_1 and S_2 both contain **new** x statements. Each of these statements declares a different variable, but both variables have the same name x. Both of the variables named x may be defined at the same time, and may have different values. To facilitate the discussion, I will use the term *identifier* to denote the syntactic object constituting the name of a variable, and the term *variable* to denote the variable itself. Thus, S has two different variables having the same identifier x. This situation does not arise in a sequential program because, at any instant during its execution, there is at most one currently active variable for any identifier. However, it does arise in concurrent programs and must be considered.

To define *val*, we must define the value it assigns to each of the variables of S, which requires giving different names to different variables. Assigning unique names to variables is a nontrivial problem, since different variables may be represented in the program by the same identifier. It is solved with selectors. I let $x(S, \gamma)$ be the name of the variable with identifier x that is declared in a **new** statement whose selector in S is γ—in other words, where γ is the path in the parse tree of S leading to the **new** statement. A "global" variable with identifier x—that is, the variable denoted by an occurrence of the identifier x outside the scope of any **new** x statement—is given the name $x()$.

I consider $x()$ to be a variable of any statement S, even if the identifier x never appears in S. For example, suppose S is the statement:

$\langle\, y := y + 1 \,\rangle$;
new z **: in** $\langle\, z := y \,\rangle$ **ni**

Then the variables of S consist of the single "bound" variable $z(S, 2)$ plus the infinite set of "free" variables $x()$, $y()$, $z()$, …, only one of which actually appears in S. Even though the variable $x()$ does not appear in S, it may appear in other statements in the complete program. The correctness of a program containing

cobegin $S \,\square\, T$ **coend**

may depend upon the obvious fact that S does not change the value of $x()$. The value of $x()$ is included as part of S's state so we can say formally that S does not change that value.

To summarize, defining the set of states $\mathcal{S}(S)$ of S requires defining the set of variables of S. The variables of S consist of the following:

- For any **new** x statement of S with selector γ, the variable named "$x(S, \gamma)$".
- For any identifier x, the variable named "$x()$".

The mapping *val* assigns a value to each of these variables.

The names of variables come up quite often when talking about programs. If S is a hundred page program, then the name $x(S, \gamma)$ takes up one hundred pages. Writing even the simplest statements about S would therefore require quite a bit of paper. Such a practical consideration is as irrelevant for the semantics of a programming language as is the cost of tape for the theory of Turing-machine computability. However, it does pose a problem in writing examples, since a simple assertion about a five-line program might take one or two pages. The solution is, of course, to give names to statements and substatements. I will use the ordinary labeling convention to do this. For example, consider the program.

$$
\begin{array}{l}
s\text{: } \textbf{if } \langle\, x > 0 \,\rangle \\
\quad \textbf{then} \quad \langle\, x := x + 1 \,\rangle \\
\quad \textbf{else} \;\; t\text{: } \textbf{new } y \\
\qquad\qquad\quad \textbf{in } \; \langle\, y := x \,\rangle; \\
\qquad\qquad\qquad\; \langle\, x := y + 2 \,\rangle \\
\qquad\qquad\quad \textbf{ni}; \\
\qquad\qquad\;\; \langle\, z := 17 \,\rangle \\
\quad \textbf{fi}
\end{array}
$$

The variable y declared by the **new** y statement will be called simply $y(t \text{ of } s)$. However, you should remember that its complete formal name is:

$$
y\left(\begin{array}{l}
\textbf{if}\langle\, x > 0 \,\rangle \\
\quad \textbf{then}\ldots \\
\quad \textbf{else}\ldots \\
\textbf{fi}
\end{array} \quad , \; \textit{else}, \textit{1} \right)
$$

4.2 Control Variables

There is more to a state than the values of program variables. To determine the future behavior at some point during the execution of the statement

$$
\begin{array}{l}
u\text{: } \textbf{begin } s\text{: } \langle\, x := x + 1 \,\rangle; \\
\qquad\qquad\;\; t\text{: } \langle\, y := y + 1 \,\rangle \\
\quad \textbf{end}
\end{array}
$$

we need to know whether control is at the beginning of statement s, at the beginning of statement t, or at the end of statment t. Since the state must determine the statement's possible future behavior, it must contain this control information.

I will describe this control information in terms of the boolean-valued *control* variables *at*, *in*, and *after*. For any substatement S, γ, there are control variables $at(S, \gamma)$, $in(S, \gamma)$, and $after(S, \gamma)$, where the values of these variables equal *true* when

$at(S, \gamma)$: control is at the beginning of substatement S, γ

in(S, γ): control is at the beginning of or inside S, γ, but not at its exit point. Note that $at(S, \gamma) \supset in(S, \gamma)$ is always true.

after(S, γ): control is at the exit point of S, γ—that is, at the point just after its execution is completed.

In addition to complete substatements, the *at*, *in*, and *after* variables are also defined for certain parts of statements that denote atomic operations—namely, the *test* of an **if** or **while** statement and the *init* clause of a **new** statement (if it has one). Also, the control variables $at(\Pi)$, $in(\Pi)$, and $after(\Pi)$ are defined for a complete program Π.

The statement u above thus has eight control variables: $at(u)$, $in(u)$, $after(u)$, $at(s \text{ of } u)$, $in(s \text{ of } u)$, $after(s \text{ of } u)$, $at(t \text{ of } u)$, $in(t \text{ of } u)$, and $after(t \text{ of } u)$. They are not all independent, however, since we have

$$\begin{aligned} at(u) &= at(s \text{ of } u) \\ after(u) &= after(t \text{ of } u) \\ in(u) &= in(s \text{ of } u) \wedge in(t \text{ of } u) \\ after(s \text{ of } u) &= at(t \text{ of } u) \end{aligned} \tag{1}$$

These equations represent aliasing relations between the control variables. The study of these aliasing relations is deferred until later.

The mapping *val* that assigns values to variables must assign values to the control variables as well as the ordinary program variables. Of course, we must assume that *at*, *in*, and *after* are not identifiers, so they cannot be used for ordinary program variables.

Variables like *at*, *in*, and *after* are sometimes called "dummy" or "ghost" variables. This seems to imply that they are not as real as ordinary program variables. Indeed, I have found that many computer scientists regard their use as somewhat distasteful—perhaps even immoral. Control variables are every bit as real as ordinary program variables. They differ from program variables only in that the programmer does not explicitly write them. Every programmer knows that he can often simplify a program's control structure—that is, eliminate control variables—by adding program variables; and, conversely, he can eliminate program variables by using a more complex control structure—that is, by adding control variables. A compiler handles both kinds of variables in very much the same way; in the compiled version of a program, the values of program variables and control variables are both encoded in terms of the contents of memory registers and program-location counters.

I therefore prefer to use the term *implicit* variables for variables other than ordinary program variables. Some languages employ other implicit variables besides control variables. For example, a language that provides a buffered message-passing primitive will contain implicit variables whose values describe the set of messages in the queues.

Ordinary program variables may be free (undeclared), like $x()$, or bound (declared), like $x(S, \gamma)$. I have written all control variables as bound variables, but are they really bound? Remember that the free variables are interface variables and bound variables are internal ones. In order to use a compiled version of a statement S, one must know where its starting and ending control points are, but need know nothing of its internal control points. This suggests that $at(S)$, $in(S)$, and $after(S)$ are interface variables for statement S, while, for any non-null selector γ, $at(S, \gamma)$, $in(S, \gamma)$, and $after(S, \gamma)$ are internal variables. The control variables $at(S)$, $in(S)$, and $after(S)$ are best viewed as undeclared and might better be written as $at()$, $in()$, and $after()$. (They are not written that way both for historical reasons and because it would tend to be confusing.) These variables are implicitly declared, and aliased to other control variables, when S is written as part of a larger statement.

4.3 Are There Other State Components?

Does a mapping *val* from variable names of S values tell us everything we need to know about the current state of S in order to determine its future behavior? At first glance, it might seem that it doesn't. For example, what is the effect of executing

s: $x := y + 1$

when the value of y is 17? The answer depends upon the type of x. If x is of type **integer**, then the execution sets x to 18. However, if x is of type **boolean**, then executing s produces an error.

Moreover, suppose x is of type integer and $y = 17$, so executing s changes the value of x to 18. What does this execution do to the value of y? If y is not aliased to x, then its value is left unchanged. However, if s appears inside the statement

new y **: integer alias** x **in** ... **ni**

then the value of y is also changed to 18.

It would therefore seem that we should add types and aliasing information to the state. In fact, we needn't. The reason is that, in language L, types and aliasing relations are *static* properties; they do not change during execution of the program. Executing an action of L does not change the type of a variable or any aliasing relations. (We sometimes think of executing a **new** statement by first executing its declarations, but that makes no sense because declarations are not actions.)

In a more complex language, types and aliasing relations can be dynamic. For example, in Pascal, if x is a variable of type pointer, then the aliasing relation "$x\uparrow$ is aliased to y" is dynamic, since its truth is changed by assigning a new value to x. In these cases, it may be necessary to add types and aliasing information to the state. However, in most languages, aliasing relations among control variables will be static, and can be handled the same way as in language L.

4.4 Renaming

For any statement S, let $\mathcal{V}(S)$ denote the set of names of variables of S. A state *val* of $\mathcal{S}(S)$ is a mapping that assigns a value to each variable name in $\mathcal{V}(S)$. For a compositional semantics, we must be able to derive information about the states of S from information about the states of its component substatements. This requires the fundamental concept of a *renaming* mapping.

Let statement T be the substatement S, γ of S. Every variable of T is a variable of S, except that it may be known by a different name. I will define $\rho_{S,\gamma}$ to be the mapping on names such that if v is the name of a variable in T, then $\rho_{S,\gamma}(v)$ is the name of the corresponding variable in S. Hence,

$$\rho_{S,\gamma} : \mathcal{V}(T) \rightarrow \mathcal{V}(S)$$

The variable $x(T, \mu)$, which is the variable of T with identifier x that is declared in the **new** statement T, μ, is called by the name $x(S, \gamma, \mu)$ when it is regarded as a variable of S. Thus,

$$\rho_{S,\gamma}(x(T, \mu)) = x(S, \gamma, \mu)$$

A variable that has the name $x()$ as a variable in T is undeclared in T. If it is undeclared in S, then it has the same name as a variable of S, so $\rho_{S,\gamma}(x()) = x()$. However, if it is declared in the **new** x statement S, ν, so ν is a prefix of γ, then $\rho_{S,\gamma}(x()) = x(S, \nu)$.

The renaming mappings compose in the natural way. If T is the substatement S, γ of S, then for any substatement T, δ of T, we have

$$\rho_{S,\gamma,\delta} = \rho_{S,\gamma} \circ \rho_{T,\delta} \tag{2}$$

By taking this equality to be a definition, we can formally define $\rho_{S,\gamma}$ for any selector γ by defining it for all primitive selectors. This formal definition should be obvious and is omitted.

Let T be the substatement S, γ of S. A state *val* of $\mathcal{S}(S)$ is a mapping from $\mathcal{V}(S)$ to values, and $\rho_{S,\gamma}$ is a mapping from $\mathcal{V}(T)$ to $\mathcal{V}(S)$. The composition $val \circ \rho_{S,\gamma}$ is therefore a mapping from from $\mathcal{V}(T)$ to values, which is a state in $\mathcal{S}(T)$. Thus, the mapping $\rho_{S,\gamma}$ induces a mapping

$$\rho^*_{S,\gamma} : \mathcal{S}(S) \rightarrow \mathcal{S}(T)$$

from states of S to states of T, defined by

$$\rho^*_{S,\gamma}(val) \stackrel{\text{def}}{=} val \circ \rho_{S,\gamma} \tag{3}$$

for any $val \in \mathcal{S}(S)$.[4]

It follows easily from (2) that the mappings $\rho^*_{S,\gamma}$ satisfy the following "adjoint" form of (2).

$$\rho^*_{S,\gamma,\delta} = \rho^*_{T,\delta} \circ \rho^*_{S,\gamma} \tag{4}$$

4.5 Actions

The states of S are defined using the set $\mathcal{V}(S)$ of variable names. The actions of S will be defined in terms of a set $\mathcal{A}(S)$ of atomic-action names of S.

The atomic actions of S are the components written in angle brackets. In the language L, there are just four kinds of atomic actions: assignment statements, **if** tests, **while** tests, and **init** clauses (of **new** statements). (I assume that the initial-value assignment of the **new** statement is performed as a single atomic action.) The set $\mathcal{A}(S)$ of atomic-action names of S consists of the set of all components S, γ of S such that γ is a selector for one of the following: an assignment statement, the *test* component of an **if** statement, the *test* component of a **while** statement, or the *init* component of a **new** statement. For reasons having to do with defining compiler correctness that

[4]The renaming mapping $\rho^*_{S,\gamma}$ has no connection with the mapping F discussed in Section 2.4.1 between the states of an implementation and the states of a higher-level program.

are irrelevant to the remainder of this paper, if Π is a complete program, then $\mathcal{A}(\Pi)$ is defined to contain one additional action name: the name λ, which is the name of a null action.[5]

Since action names are just the names of substatements, the renaming mappings can be applied to them in the usual way. Thus, if T is the substatement S, γ of S, then

$$\rho_{S,\gamma} : \mathcal{A}(T) \rightarrow \mathcal{A}(S)$$

is defined in the obvious way—namely, $\rho_{S,\gamma}(T, \mu) = S, \gamma, \mu$.

4.6 States and Actions: A Formal Summary

For every statement S in the language L, I have defined the following:

- A set $\mathcal{V}(S)$ of variable names, consisting of:
  - all program-variable names of the form $x()$ for every identifier x and of the form $x(S, \gamma)$, where γ is the selector in S of a **new** x statement.
  - all control-variable names of the form $at(S, \gamma)$, $in(S, \gamma)$, and $after(S, \gamma)$, for all substatements S, γ of S.
- The set $\mathcal{S}(S)$ of states of S, which is defined defined to be the set of all mappings

  $$val : \mathcal{V}(S) \rightarrow \mathcal{Z} \cup \{true, false\}$$

- The set $\mathcal{A}(S)$ of atomic-action names of S, defined to be the set of all components of the form S, γ where S, γ is a **while** or **if** test, an atomic assignment, or an **init** clause of a **new** statement.

---

[5]The following example shows why the λ action is needed. Let S be the statement

$$x := x^2$$

of program Π, and consider the 42 steps in the machine-language implementation π of Π that execute statement S. As mentioned earlier, 41 of them will be stuttering actions that leave the value of x unchanged. Before the first 41 steps have been executed, π's state may no longer have the information needed to deduce the initial value of x. For example, after 20 steps, the state of π may show that x will wind up with the value 4, but may not show whether it started equal to 2 or -2. This means that the single nonstuttering action must be among the first 20 steps, and the remaining steps must be stuttering actions of the statement following S. If S is the last statement of Π, then these remaining steps are λ actions.

- If T is the substatement S, γ of S, the renaming mappings

$$\rho_{S,\gamma} : \mathcal{V}(T) \to \mathcal{V}(S)$$
$$\rho^*_{S,\gamma} : \mathcal{S}(S) \to \mathcal{S}(T) \rho_{S,\gamma} : \mathcal{A}(T) \to \mathcal{A}(S)$$

  These renaming mappings satisfy (2) and (4).

5 Temporal Logic

In the action-axiom semantics, I use temporal logic to express the constraints describing when an action must eventually occur. Temporal logic, introduced into the study of concurrent programs by Pnueli [13], is now quite familiar. I will therefore only sketch the logic that I will need, and refer the reader to [5] and the appendix of [6] for more details.

5.1 Predicates

The building-blocks of our temporal logic are *predicates.* For any program statement S, I define a set of predicates. There are two kinds of predicates: state predicates and action predicates.

A state predicate of S is just an expression constructed from variable names in $\mathcal{V}(S)$, including control variable names. For example,

$$at(S, \gamma) \vee \neg b() \quad \supset \quad x(S, \mu) = y() + 1$$

I will also include as predicates such expressions as $v \in \mathcal{Z}$, where v is a variable name and $\mathcal{Z}$ denotes the set of integers.

Since a state in $\mathcal{S}(S)$ assigns a value to all variable names in $\mathcal{V}(S)$, it assigns a value to a predicate. For any state s of $\mathcal{S}(S)$, I denote by $s \models P$ the value assigned to the state predicate P by the state s.

A predicate is normally a boolean-valued expression, but I have not restricted predicates in this way; $y() + 17$ is just as much a predicate as $\neg b()$. The reason is that there is no way of knowing whether an expression has a boolean value without knowing the types of all its variables, and the types of undeclared variables are not known. We must have rules for computing the value of $y() + 17$ even when the value of $y()$ is $true$. I will handle this problem by adding an additional *undefined* value, and define $true + 17$ to equal *undefined.*

The presence of an *undefined* value means that we must be careful when manipulating expressions, since the usual rules of arithmetic and logic don't hold. For example, $x + 1 > x$ does not equal *true* if x is a boolean. However, $x \in \mathcal{Z} \supset (x + 1 > x)$ should always have the value $true$.

It is necessary to allow predicates to have *logical value variables* (not to be confused with program and control variables) and quantifiers. Thus,

$$\forall \eta : x() + \chi > y(S, \gamma) + \eta$$

is a predicate containing the free logical value variable χ, the bound logical value variable η, and the two program variables $x()$ and $y(S, \gamma)$ in $\mathcal{V}(S)$. For any predicate P and state s of $\mathcal{S}(S)$, $s \models P$ is a formula involving values and value variables.

An action predicate of S is an expression of the form $Act(S, \gamma)$, where S, γ is a substatement of S. The action predicate $Act(S, \gamma)$ defines a boolean-valued function on the set $\mathcal{A}(S)$ that has the value *true* on an action-name α if and only if α is the name of an atomic action of the substatement S, γ. I write $\alpha \models Act(S, \gamma)$ to denote the value of $Act(S, \gamma)$ on α. Remembering that atomic-action names are just components S, μ of S, we see that $S, \mu \models Act(S, \gamma)$ equals true if and only if $\mu = \gamma, \nu$ for some ν.

Let $\mathcal{PR}(S)$ denote the set of all state and action predicates of S. Since state predicates are built out of variable names and action predicates are all of the form $Act(S, \gamma)$, the renaming mappings induce mappings on predicates in the obvious way. If T is the substatement S, γ of S, then

$$\rho_{S,\gamma} : \mathcal{PR}(T) \to \mathcal{PR}(S)$$

These renaming mappings satisfy the expected relation (2). Moreover, if P is a tautotology of $\mathcal{PR}(T)$, then $\rho_{S,\gamma}$ is a tautology of $\mathcal{PR}(S)$.

5.2 The Unary Temporal Operators

I will begin with the simpler form of temporal logic, using only unary temporal operators. The formulas of this logic are constructed from predicates, the usual logical operations, and the two unary temporal operators $\Diamond$ and $\Box$. More precisely, for any statement S, the set $\mathcal{TL}(S)$ of temporal logic formulas of S consists of all formulas constructed from $\mathcal{PR}(S)$ with the logical operators and the unary operators $\Diamond$ and $\Box$.

Just as predicates are true or false for states, temporal logic formulas are true or false for behaviors. Let $\mathcal{B}(S)$ denote the set of all finite and infinite sequences of the form

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \tag{5}$$

where the s_i are states in $\mathcal{S}(S)$ and the α_i are atomic-action names in $\mathcal{A}(S)$. We give a semantics for temporal logic formulas by defining $\sigma \models A$ for any behavior σ in $\mathcal{B}(S)$ and any temporal logic formula A in $\mathcal{TL}(S)$.

If σ is the sequence (5), for any nonnegative integer n let σ^{+n} be the sequence

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

unless σ is finite and n is less than the length m of σ, in which case σ^{+n} is defined to be the sequence consisting of the single state s_m. We define $\sigma \models A$ inductively as follows.

- If A is a state predicate, then $\sigma \models A \stackrel{\text{def}}{=} s_0 \models A$. (The value of a state predicate is its value in the starting state.)
- If A is an action predicate, then $\sigma \models A \stackrel{\text{def}}{=} \alpha_1 \models A$. (The value of an action predicate is its value for the first action.) However, if σ consists of the single state s_0 with no actions, then $\sigma \models A \stackrel{\text{def}}{=} \mathit{false}$.
- The logical connectives "distribute" in the obvious way. For example,

  $$\sigma \models (A \vee B) \stackrel{\text{def}}{=} (\sigma \models A) \vee (\sigma \models B)$$

- The temporal operators are defined by

  $$\sigma \models \Box A \stackrel{\text{def}}{=} \forall n : \sigma^{+n} \models A$$
  $$\sigma \models \Diamond A \stackrel{\text{def}}{=} \exists n : \sigma^{+n} \models A$$

Note that $\Diamond$ is the dual of $\Box$—that is, $\Box A \equiv \neg\Diamond\neg A$ for any A. The operator $\leadsto$ is defined by

$$A \leadsto B \stackrel{\text{def}}{=} \Box(A \supset \Diamond B)$$

Note also that $\Box\Diamond A$ has the intuitive meaning that A is true infinitely often.

5.3 The Binary Temporal Operators

While the unary temporal operator $\Box$, and the operators derivable from it, are quite natural and easy to understand, they are not sufficiently expressive. We need an additional binary temporal operator. There are many binary operators that are equivalent in the sense that one can be represented by another. My favorite one, introduced in [6], is the operator $\trianglelefteq$, whose semantics is defined as follows.

$$\sigma \models (A \trianglelefteq B) \stackrel{\text{def}}{=} \forall n : (\forall m \leq n : \sigma^{+m} \models A) \supset \sigma^{+n} \models B$$

Intuitively, $A \trianglelefteq B$ means that B holds for at least as long as A does—that is, A holds for a length of time $\leq$ the length of time that B holds, so it represents a "temporal $\leq$".

I will extend the definition of $\mathcal{TL}(S)$ to include temporal formulas constructed with the operator $\trianglelefteq$ as well as $\Box$ and $\Diamond$. The unary operators can be defined in terms of $\trianglelefteq$; for example, $\Box A \equiv \mathit{true} \trianglelefteq A$. Thus, the single operator $\trianglelefteq$ is all we need.

The operator $\triangleleft$ is defined in terms of $\trianglelefteq$ by

$$A \triangleleft B \stackrel{\text{def}}{=} (A \vee \neg B) \trianglelefteq B$$

A little thought shows that

$$\sigma \models (A \triangleleft B) \stackrel{\text{def}}{=} \forall n : (\forall m < n : \sigma^{+m} \models A) \supset \sigma^{+n} \models B$$

so $A \triangleleft B$ means that A holds for a length of time $<$ the length of time that B holds. The operators $\triangleleft$ and $\trianglelefteq$ obey the same transitivity relations that $<$ and $\leq$ do. For example,

$$(A \trianglelefteq B) \wedge (B \triangleleft C) \supset (A \triangleleft C)$$

I will use $\trianglelefteq$ to define a new type of temporal formula that is useful for specifying actions. For any action name $\alpha \in \mathcal{A}(S)$ and any predicates P and Q, I define $\{P\}\langle\alpha\rangle\{Q\}$ to be the temporal logic formula that means that executing α starting in a state in which P is true can produce a state in which Q is true. (It is just the ordinary Hoare triple for the atomic action α, viewed as a temporal formula.) However, we must allow stuttering actions of α which do nothing, and hence leave P true. The formal definition is

$$\{P\}\langle\alpha\rangle\{Q\} \stackrel{\text{def}}{=} P \supset (\mathit{Act}(\alpha) \triangleleft P \vee Q)$$

Intuitively, $\sigma \models \{P\}\langle\alpha\rangle\{Q\}$ asserts that if, P is true in the initial state of σ and the first one or more actions of σ are α actions, then $P \vee Q$ remains true through the first state before the first action that is not an α action. If $Q \supset \neg \mathit{Act}(\alpha)$ holds, which is the only case in which this is formula will be used, then Q will be true only after the last of these initial α actions. Hence, it asserts that α can perform a series of stuttering actions leaving P true, and can also "finish" by making Q true.

The formula $\{P\}\langle\alpha\rangle\{Q\}$ is used to describe how an action can change the state. It is also necessary to state that an action does not change something. I therefore introduce the formula $e \stackrel{\alpha}{\not\leftarrow}$, which asserts that the action α does not change the value of the expression e. It is defined by

$$e \stackrel{\alpha}{\not\leftarrow} \stackrel{\text{def}}{=} \forall \eta : (e = \eta) \supset \mathit{Act}(\alpha) \triangleleft (e = \eta)$$

5.4 Renaming

Having already extended the renaming mappings to predicates, it is easy to extend them to temporal logic formulas constructed from predicates. For example, for any variable names v and w, we have

$$\rho_{S,\gamma}(\Box(v \vee \Diamond w)) \;=\; \Box(\rho_{S,\gamma}(v) \vee \Diamond\rho_{S,\gamma}(w))$$

Thus, if T is the substatement S, γ of S,

$$\rho_{S,\gamma} : \mathcal{TL}(T) \to \mathcal{TL}(S)$$

The renaming mappings do *not* induce any mappings on behaviors. This is because they map states and atomic-action names in opposite directions:

$$\begin{array}{l} \rho^*_{S,\gamma} : \mathcal{S}(S) \to \mathcal{S}(T) \\ \rho_{S,\gamma} : \mathcal{A}(T) \to \mathcal{A}(S) \end{array}$$

Since a behavior consists of an alternating sequence of states and action names, the renaming mappings do not work on behaviors. This may be the source of the difficulties encountered in trying to give a behavioral semantics—one in which $\mathcal{M}[\![S]\!]$ is a set of behaviors—to concurrent programming languages

5.5 Temporal Logic as Semantics

For each statement S of the programming language, I have defined a set $\mathcal{TL}(S)$ of temporal logic formulas, and a notion of semantic validity $\models$ for these formulas. In an action-axiom semantics, the meaning of S includes a set of temporal logic formulas that must be satisfied by the behaviors of S. This set of formulas is specified by giving axioms and inference rules, which means that we have a logical system and a notion of a provable formula. I will not discuss provability, and will restrict myself to validity.

I have defined $\sigma \models A$ for a behavior σ and a temporal logic formula A, but I have not defined the concept $\models A$—validity of a formula. For any formula $A \in \mathcal{TL}(S)$, one usually defines $\models A$ to equal *true* if and only if $\sigma \models A$ equals *true* for all behaviors σ in $\mathcal{B}(S)$.

The formulas A for which $\models A$ is true are those that are true for all sequences of states and actions from S, so their truth rests only on the properties of S's sets of states and actions, not on properties of S's dynamic behavior—for example, the formula $\Box(x() \in \mathcal{Z} \supset (x^2 \geq x))$. A formula A is said to valid for all behaviors in some subset Σ of $\mathcal{B}(S)$, written $\models_\Sigma A$, if

$\sigma \models A = \mathit{true}$ for all $\sigma \in \Sigma$. The valid formulas for a program S are those that are valid for the set of all behaviors of S.

Note that $\models_\Sigma \mathit{false}$ is true if and only if Σ is the empty set. The semantics I give can produce contradictory sets of axioms for a program—axioms from which one can deduce the formula *false*. This is not an inconsistency in the system; rather it is an indication that there are no legal behaviors of the program, so the program is illegal. This will be the case, for example, if a program assigns a boolean value to a variable of type **integer**.

I consider the notion $\models_\Sigma$ of semantic validity only for sets Σ having the property that for any $\sigma \in \Sigma$ and any $n \geq 0$: $\sigma^{+n} \in \Sigma$. Intuitively, this means that the temporal logic does not assume any preferred starting state. Formally, this means that the truth of $\models A$ implies the truth of $\models \Box A$—a rule of inference known to logicians as the *Necessitation Rule*. This rule implies that whenever we give a predicate P as a temporal-logic axiom, we are really asserting that $\Box P$ is true.

The validity of the Necessitation Rule means that it is impossible to write a temporal logic formula which asserts that the program is executed only in certain starting states. Thus, one should define the semantics $\mathcal{M}[\![S]\!]$ of S to consist of both a set of temporal logic axioms that constrain the allowed behaviors of S and a set of nontemporal axioms—that is, predicates—that constrain the starting state. The semantic meaning $\mathcal{M}[\![S]\!]$ defines the set of behaviors of S to be the set of all behaviors σ such that:

- $\sigma \models A$ for every temporal axiom $A \in \mathcal{M}[\![S]\!]$, and
- $s_0 \models A$ for every nontemporal axiom $A \in \mathcal{M}[\![S]\!]$, where s_0 is the starting state of σ.

However, as I will show, it is not necessary to specify any initial states for a substatement S of a program. The only initial-state specification that must be added is that a complete program starts at its entry point.

5.6 There Won't Be a Next Time

An important "feature" of the temporal logic I am using is that there is no "next time" operator. There is no way in this logic to express the concept of the next state in the behavior. In fact, no formula in the logic can distinguish between two behaviors that differ only in the addition of "stuttering" actions—that is, where an action $s \xrightarrow{\alpha} t$ in the behavior is replaced by the finite sequence of actions

$$s \xrightarrow{\alpha} s \xrightarrow{\alpha} \cdots \xrightarrow{\alpha} s \xrightarrow{\alpha} t$$

It is the inability to distinguish stuttering that makes it easy to talk about a lower-level program implementing a higher-level one.

5.7 Implementation Mappings

I can now continue the discussion, begun in Section 2.4, of what it means for a lower-level program to correctly implement a higher-level one. Let Π be the higher-level program and π be its lower-level implementation. From the point of view of behaviors, we saw that there should be mappings F from the states and actions of π to the states and actions of Π so that if σ is the behavior

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots$$

of π, then $F(\sigma)$, which is defined to be

$$F(s_0) \xrightarrow{F(\alpha_1)} F(s_1) \xrightarrow{F(\alpha_2)} \cdots$$

is a behavior of Π.

How are the mappings F defined? In action-axiom semantics, one never mentions states, just state predicates—mappings from the state into a set of Booleans. A state is determined by the values of all state predicates. To define a mapping $F : \mathcal{S}(\pi) \rightarrow \mathcal{S}(\Pi)$, one defines a mapping F^* that maps state predicates of Π into state predicates of π. Intuitively, F^* defines the state predicates of Π in terms of the state predicates of π. For example, $F^*(x(\Pi, \gamma) > 0)$ is the state predicate of π that "implements" the state predicate $x(\Pi, \gamma) > 0$ of Π; in other words, it is the translation of the high-level statement that the value of the variable $x(\Pi, \gamma)$ is positive into a lower-level statement involving the values of memory registers, program counters, etc. Defining the mapping F^* requires describing how the variables (both program and control variables) of Π are implemented by the "variables" (machine registers) of π. The mappings F and F^* are related by

$$s \models F^*(P) \;\equiv\; F(s) \models P$$

for any state s in $\mathcal{S}(\pi)$ and state predicate P in $\mathcal{PR}(\Pi)$.

In a similar way, F^* is defined to map action predicates of Π into action predicates of π, so $F^* : \mathcal{PR}(\Pi) \rightarrow \mathcal{PR}(\pi)$. Finding the mapping F^* is the heart of the proof that π correctly implements Π. The discussion in Section 2.4 of the dual mapping F applies equally well to F^*, and I will not discuss further how F^* is actually constructed.

Since temporal logic formulas are constructed from predicates and temporal operators, there is an obvious extension of F^* to a mapping from $\mathcal{TL}(\Pi)$ to $\mathcal{TL}(\pi)$. For example,

$$F^*(in(\Pi, \rho) \lhd x(\Pi, \gamma) > 0) \quad = \quad F^*(in(\Pi, \rho)) \lhd F^*(x(\Pi, \gamma) > 0)$$

It follows from these definitions that for any behavior σ of π and any formula A in $\mathcal{TL}(\Pi)$:

$$\sigma \models F^*(A) \quad \equiv \quad F(\sigma) \models A$$

In terms of behaviors, π correctly implements Π if, for every possible behavior σ of π, $F(\sigma)$ is a possible behavior of Π. For simplicity, let us ignore the initial-state specification, so the meaning $\mathcal{M}[\![\Pi]\!]$ of Π in an action-axiom semantics is a set of temporal logic axioms, and $F(\sigma)$ is a possible behavior of Π if and only if $F(\sigma) \models A$ is true for all $A \in \mathcal{M}[\![\Pi]\!]$. But $F(\sigma) \models A$ is true if and only if $\sigma \models F^*(A)$ is, so π correctly implements Π if and only if $\sigma \models F^*(A)$ is true for all $A \in \mathcal{M}[\![\Pi]\!]$ and all behaviors σ of π. The behaviors of π consist of the sequences satisfying all the axioms of $\mathcal{M}[\![\pi]\!]$. It follows from this that π correctly implements Π if, for every axiom A in $\mathcal{M}[\![\Pi]\!]$, $F^*(A)$ is implied by the axioms in $\mathcal{M}[\![\pi]\!]$. Thus, proving correctness of the implementation involves deducing, from the axioms for π, the truth of $F^*(A)$—the translation of A into an assertion about π—for every axiom A in $\mathcal{M}[\![\Pi]\!]$.

As explained in Section 2.4, a compiler is free to implement local variables and internal actions in any fashion, but interface (global) variables and external actions have a fixed implementation. The mapping F^* is defined on state predicates by defining $F^*(v)$ in terms of the variables of π, for every variable v of Π. The definition of $F^*(v)$ is arbitrary for a local variable v, but is fixed for an interface variable. To prove the correctness of an implementation, we are allowed to define $F^*(v)$ any way we like if v is a local variable, but must use the predetermined definition if v is an interface variable. Similar comments apply to actions.

I find it helpful to think of the semantics $\mathcal{M}[\![\Pi]\!]$ of Π as the specification of a lower-level implementation. When viewed this way, there is an implicit existential quantification over the names of all local variables and internal actions. More precisely, the specification consists of the conjunction of all the axioms in $\mathcal{M}[\![\Pi]\!]$, with existential quantification over these variable and action names. The names of interface variables and external actions represent fixed, externally defined objects.

6 The Semantics of Language L

With these preliminaries out of the way, I can now give the semantics of language L. This is done by defining the meaning $\mathcal{M}[\![S]\!]$ of S, where S is any statement or complete program. I define $\mathcal{M}[\![S]\!]$ to consist of a set of temporal logic axioms that specify the set of behaviors of S. As discussed below, for a complete program Π, I will also need one nontemporal axiom—that is, a predicate—to specify the starting state.

The basic idea behind achieving a compositional semantics is the requirement that any axiom asserted about a statement T must be valid for any statement containing T as a substatement. Of course, an axiom about T must be renamed to become an axiom about a statement containing T. The formal statement of this idea is:

> *Composition Principle*: If T is substatement S, γ of S, then for any formula A: if $A \in \mathcal{M}[\![T]\!]$ then $\rho_{S,\gamma} A \in \mathcal{M}[\![S, \gamma]\!]$.

6.1 Syntactic Predicates

I observed in Section 4.3 that there is information we need in order to define $\mathcal{M}[\![S]\!]$ that is not in the state of S—namely, type and aliasing information. This information is not in the state because it is determined syntactically and does not change during execution of S. Unfortunately, it may be determined not by the syntax of S, but by the syntax of the complete program containing S. For example, the aliasing relations defined by a **new** statement are not known when defining $\mathcal{M}[\![S]\!]$ for a statement S in its body.

For our simple language L, typing information can be handled by ordinary axioms; the fact that a variable v is of type integer is expressed by the requirement that the value of v always be an integer. Aliasing relations can also be expressed by similar requirements—for example the aliasing relation defined by

$$\textbf{new } z \textbf{ : gausian} \;\; \textbf{alias } x + y\sqrt{-1} \;\; \textbf{in} \ldots$$

is expressed by requiring that the value of $z(S)$ always equals the value of $x() + y()\sqrt{-1}$. However, the fact that $z(S)$ is *not* aliased to the variable $a()$ cannot be expressed in this way.

The absence of aliasing relations is expressed with a new relation $\perp$, where $v \perp w$ means intuitively that assigning a value to the variable named v does not change the value of the variable named w, and vice-versa. An ordinary state predicate such as $v = w$, which asserts that the values of v and w are equal, is true or false for a particular state. However, the truth of the expression $v \perp w$ depends only upon the syntactic structure of the program; it is true for one state of S if and only if it is true for all states of S.

An expression like $v \perp w$, whose value is a boolean that depends only on the program syntax, is called a *syntactic predicate*. Unfortunately, if v and w are undeclared variables of S, then the value of $v \perp w$ depends upon the syntax of the program that contains S, and its value is not determined when we are defining $\mathcal{M}[\![S]\!]$. Thus, a syntactic predicate either has a definite boolean value, or else has an undetermined value.

I will allow syntactic predicates to appear in a temporal logic formula of $\mathcal{TL}(S)$ anywhere that an ordinary state predicate can. However, there is no reason to write $\Diamond(v \perp w)$, since if $v \perp w$ is ever true, then it is always true for every state of S. Formally, a syntactic predicate in a temporal logic formula of $\mathcal{TL}(S)$ is viewed as a boolean constant if its value is determined by S, and as a logical variable if its value is undetermined.

Formally, a syntactic predicate appearing in an axiom of $\mathcal{M}[\![S]\!]$ is a constant if its value is determined by S, and it is a logical variable if its value is not determined. Thus, writing the syntactic predicate $x(S, \gamma) \perp y(S, \mu)$ is simply an "abbreviation" for either *true* or *false*, since the aliasing relations of variables declared in S are determined. On the other hand, a syntactic predicate such as $x() \perp y()$ represents a logical variable, since aliasing relations between undeclared variables are undetermined. Because there is an implicit universal quantification over all free logical variables in an axiom, an axiom containing a syntactic predicate is asserted to be true whatever value is assigned to it.

We can apply renaming mappings to syntactic predicates in the obvious way. Thus, if P is a syntactic predicate for T, and T is the substatement S, γ of S, then $\rho_{S,\gamma}(P)$ is a syntactic predicate for S. When the predicate P occurs in an axiom A of $\mathcal{M}[\![T]\!]$, the expression $\rho_{S,\gamma}(P)$ occurs in $\rho_{S,\gamma}(A)$, which, by the Composition Principle, is an axiom of $\mathcal{M}[\![S]\!]$. A little thought reveals that, to ensure the validity of the Composition Principle, we want the following property to hold:

> *Syntactic Composition Property*: For any syntactic predicate P: if the value of P is defined for T, then the value of $\rho_{S,\gamma}(P)$ is also defined for S and equals the value of P.

The use of syntactic predicates is not really necessary. I could include the information that they express in the state. Had I done so, a syntactic predicate having an undetermined value would become a component of the state, and $\mathcal{S}(S)$ would include states having all possible values of that predicate. A syntactic predicate whose value is determined in a statement S could be represented either as a state component constrained to have only one possible value, or as a constant.

6.1.1 Aliasing

The absence of aliasing will be expressed by the relation $\perp$ between variable names in $\mathcal{V}(S)$. This will be done axiomatically by defining a logical system for deriving $\perp$ relations To do this, I must first introduce a relation $\prec$, where $v \prec \{w_1, \ldots, w_n\}$ means that the variable name v is not directly aliased to any variable names other than $w_1, \ldots, w_n$. It is convenient to extend this relation to a relation between sets of variable names, where $\{v_1, \ldots, v_m\} \prec \{w_1, \ldots, w_n\}$ means that each of the variable names v_i is not directly aliased to any variable names other than the w_j. We then have the obvious inference rule:

> For any sets $V, W \in \mathcal{V}(S)$: if $V' \subseteq V$, $W \subseteq W'$, and $V \prec W$, then $V' \prec W'$.

The relation $\perp$ on $\mathcal{V}(S)$ is defined so that $v \perp w$ means that neither v nor w is aliased, directly or indirectly, to the other. In other words, it means that $v \neq w$ and there do not exist both a chain of $\prec$ relations from v and a chain of $\prec$ relations from w that lead to a common variable name. This leads to the following rules for deriving $\perp$ relations.

- If $v \prec \emptyset$, $w \prec \emptyset$, and $v \neq w$, then $v \perp w$.
- If $v \perp w$ then $w \perp v$.
- If $v \prec \{w_1, \ldots, w_n\}$, $w_1 \perp w$, $\ldots$, $w_n \perp w$, and $v \neq w$, then $v \perp w$.

I extend $\perp$ to a relation on finite sets by letting $\{v_1, \ldots, v_n\} \perp \{w_1, \ldots, w_n\}$ denote $\forall i, j : v_i \perp w_j$.

Having given general rules for reasoning about $\prec$, I must define the relation for variables names in $\mathcal{S}(S)$ for an arbitrary statement S. The value of a syntactic predicate $V \prec W$ or $V \perp W$ will be undetermined if V and W both contain the names of undeclared variables of S. To define the values of the ones that are determined, I will take the Syntactic Composition Property as an axiom, and give a recursive definition based upon the structure of S.

The first observation is that a program variable cannot be aliased to a control variable, and vice-versa. I therefore require that $v \perp w$ equal *true* whenever v is a program variable and w is a control variable.

Since the only dependency relations on program variables are introduced by the **alias** clauses of **new** statements, all dependency relations among program variables are obtained from the Syntactic Composition Property and the following axiom:

> If S is the statement **new** x ... **alias** exp ... and $y_1, \ldots, y_n$ are all the variable names in exp, then $x(S) \prec \{y_1(), \ldots, y_n()\}$.

I must now define the dependency relations on control variable names. I will do this by assuming the Syntactic Composition Property and defining the relations introduced by each language construct. There are a number of aliasing relations that are similar to the ones introduced by an **alias** clause, except that the aliasing relations for the control variables are implicit in the program structure rather stated explicitly in a **new** statement. To define the $\prec$ relations, I will write down these aliasing equations, where the control variable comprising the left-hand side of an equation is considered to depend upon each of the variables on the right-hand side. There is one set of equations for each programming language construct.

Besides these aliasing equalities, some other aliasing relations are given as boolean expressions—that is, asserting that the boolean expressions are true. No dependency relations are implied by these expressions, but they are listed here for future reference.

There is only one axiom that explicitly defines $\perp$ relations; it is given for the **cobegin** statement.

assignment
$$\begin{array}{l}
in(S) \;=\; at(S) \\
\neg(at(S) \;\wedge\; after(S))
\end{array}$$

if
$$\begin{array}{lcl}
in(S, test) & = & at(S, test) \\
after(S, test) & = & at(S, then) \vee at(S, else) \\
at(S) & = & at(S, test) \\
in(S) & = & in(S, test) \vee in(S, then) \vee in(S, else) \\
after(S) & = & after(S, then) \vee after(S, else) \\
\multicolumn{3}{l}{\neg\,(at(S, test) \;\wedge\; (in(S, then) \;\vee\; in(S, else) \;\vee\; after(S)))} \\
\multicolumn{3}{l}{\neg(in(S, then) \;\wedge\; in(S, else))}
\end{array}$$

$$
\begin{array}{llcl}
\textbf{while} & at(S) & = & at(S, test) \\
 & in(S, test) & = & at(S, test) \\
 & after(S, test) & = & at(S, body) \vee after(S) \\
 & after(S, body) & = & at(S, test) \\
 & in(S) & = & at(S, test) \vee in(S, body) \\
 & \multicolumn{3}{l}{\neg\,(at(S, test) \wedge (after(S) \vee in(S, body)))}
\end{array}
$$

new There are two cases. If there is no **init** clause, then:

$$
\begin{array}{rcl}
at(S) & = & at(S, body) \\
after(S) & = & after(S, body) \\
in(S) & = & in(S, body)
\end{array}
$$

If there is an **init** clause, then:

$$
\begin{array}{rcl}
at(S) & = & at(S, init) \\
in(S, init) & = & at(S, init) \\
after(S, init) & = & at(S, body) \\
in(S) & = & in(S, init) \vee in(S, body) \\
after(S) & = & after(S, body) \\
 & & \neg(at(S, init) \wedge in(S, body))
\end{array}
$$

cobegin If there are n clauses in the **cobegin**, then

$$
\begin{array}{rcl}
at(S) & = & at(S,1) \wedge \ldots \wedge at(S,n) \\
after(S) & = & after(S,1) \wedge \ldots \wedge after(S,n) \\
in(S) & = & in(S,1) \wedge \ldots \wedge in(S,n) \\
\{in(S,i), after(S,i)\} & \perp & \{in(S,j), after(S,j)\} \text{ for } i \neq j
\end{array}
$$

sequence If S is $S_1;\ \ldots\ \ S_n$, then for all $i = 1, \ldots, n$:

$$
\begin{array}{rcl}
after(S, i-1) & = & at(S,i) \;\; \text{if } i > 0 \\
in(S) & = & in(S,i) \\
 & & \neg(in(S,i) \wedge in(S,j)) \text{ for } i \neq j
\end{array}
$$

program If S is the complete program, then

$$
\begin{array}{rcl}
at(S) & = & at(S, body) \\
in(S) & = & in(S, body) \\
after(S) & = & after(S, body)
\end{array}
$$

A close study of these aliasing relations reveals that we can prove a relation such as $in(S, \gamma) \perp at(S, \mu)$ if and only if the substatements S, γ and S, μ lie in different clauses of a **cobegin**.

The $\perp$ relations among program variables and the aliasing and $\perp$ relations among control variables are regarded as axioms in a separate system for reasoning about syntactic expressions. However, they play the same function as axioms of $\mathcal{M}[\![S]\!]$. For example, if S, γ is an assignment statement, then the aliasing relation $\neg(at(S, \gamma) \wedge after(S, \gamma))$ allows us to deduce $\Diamond(\neg at(S, \gamma))$ from $\Diamond after(S, \gamma)$.

6.1.2 Syntactic Typing Relations

Because the type structure of our language L is so simple, no explicit reference to types need appear in its semantics. However, this is not the case for a language in which the action of an assignment statement is affected by the types of its left- and right-hand sides—for example, if coercion was performed. We would also have to introduce explicit reference to types if a type mismatch in an assignment statement produced a run-time error or an indeterminate result, or if it halted the process executing the assignment.

Explicit reference to types is done by introducing predicates such as $type(x) =$ **integer**. If the types of variables are determined syntactically by the program text, then these predicates would be syntactic predicates. Otherwise, they would have to be ordinary state predicates, and the state would have to include components that determine their values.

6.1.3 Reasoning About Syntactic Expressions

Although a syntactic predicate like $v \perp w$ resembles an ordinary state predicate like $v = 7$, it is logically quite different. The variable name "v" denotes the value of the variable in the expression $v = 7$, while it denotes the name itself in $v \perp w$. For example, from the expressions $v = 7$ and $w = v$ we can deduce $w = 7$. However, from the syntactic expression $u \perp v$ and the ordinary expression $w = v$ we cannot in general deduce $u \perp w$; just because the *values* of two variables happen to be equal in some state does not imply that the variables have the same aliasing relations. We can only make that conclusion if $w = v$ is a syntactic equality of names, rather than an expression denoting equality of values.

By introducing syntactic predicates as a class of entities separate from ordinary state predicates, with their own logical system for reasoning about them, I have circumvented the need to distinguish between the use of a variable name as a name and as a value. In a syntactic predicate, a variable name represents itself. In a state predicate, it represents the value of the variable. Using two different logical systems avoids confusion. One cannot make invalid deductions, like deducing a $\perp$ relation from the equality of the values of v and w, because inferences about $\perp$ can be made only in the logic for reasoning about syntactic predicates, whereas equality of values can be expressed only with state predicates, and one reasons about them with a separate logic.

For languages in which types and aliasing relations are dynamic properties, so they must be reflected in the state, we cannot use this trick for separating the two different uses of variable names. We must then write $\mathit{value}(v)$ rather than the variable name v to denote the value of v. Equality of values is denoted by the predicate $\mathit{value}(w) = \mathit{value}(v)$, and $w = v$ denotes equality of names.

6.1.4 Logical Name Variables

Just as I introduced logical value variables in state predicates, I will also introduce *logical name variables* for syntactic predicates. A logical value variable is a logical variable with an implicit range in the set of values that a variable may have. Similarly, a logical name variable is a logical variable with an implicit range in the set of names that a variable may have. I will use the letter ν to denote a logical name variable.

The use of logical name variables has an important implication with respect to renaming. Consider an axiom of the form $\forall \nu : A(\nu)$. Viewed as a formula in $\mathcal{TL}(S)$, it is equivalent to an infinite conjunction of the form $A(v_1) \wedge A(v_2) \wedge \cdots$, where the v_i are all the names in $\mathcal{V}(S)$. However, the two formulas behave differently under a renaming mapping ρ. In particular, $\rho(\forall \nu : A(\nu))$ equals $\forall \nu : \rho(A(\nu))$, so the renamed formula includes a quantification over variable names not present in $\rho(A(v_1) \wedge \cdots)$.

6.2 Starting States

One might expect that the meaning $\mathcal{M}[\![S]\!]$ of a statement S should include a set of axioms that determine the set of starting states. However, consider what the initial value of a program variable should be. The user has no way of specifying it, since an **init** clause of a **new** statement is interpreted as an executable action that replaces the initial value with the specified one. One might want to specify that the initial value of a variable v of type **integer** should be an integer. However, $\mathcal{M}[\![S]\!]$ will contain an axiom asserting that this is true for every state during the execution of S, so it is therefore true of the initial state. Similarly, the axioms in $\mathcal{M}[\![S]\!]$ will assert that the aliasing relations specified by **new** statements are true throughout the execution, so they are also constrained to hold in the initial state.

What about the initial values of control variables? Surely we should require that a statement S should start in a state in which $at(S)$ is true. However, this would be a mistake because it would violate the Composition Principle, since $\rho_{T;S,2}(at(S))$ should not be true of the starting state of the sequence of statements $T; S$, and our whole approach is based upon the Composition Principle.

Remember that the only reason for specifying the starting state is to be able to obtain from our semantics a set of behaviors. However, we are really interested only in the set of behaviors of a complete program, not of its substatements. There is no reason to constrain the starting states of substatements; we need only constrain the starting state of a complete program, which we do by simply assuming that $at(\Pi)$ is true of the initial state of a complete program Π. We can do this without violating the Composition Principle because a complete program cannot be part of any larger statement.

6.3 Behavior Axioms

I now define the set $\mathcal{M}[\![S]\!]$ of behavioral axioms for any statement and complete program S. This will, of course, be done compositionally, giving a set axioms for each language construct. Remember that in addition to the axioms given explicitly below, $\mathcal{M}[\![S]\!]$ also contains all the axioms implied by the Composition Principle.

I will include in $\mathcal{M}[\![S]\!]$ axioms to assert that the appropriate aliasing relations hold throughout the execution of S. For control variables, those aliasing relations were already described in Section 6.1.1. Rather than write them over again, I will simply assume that the aliasing relations described there appear as axioms in $\mathcal{M}[\![S]\!]$ for the appropriate construct describing S. For example, the list of axioms for the assignment given below are assumed implicitly to include the axioms $in(S) = at(S)$ and $\neg at(S) \wedge after(S)$ from Section 6.1.1. (However, the $\perp$ relations given for the **cobegin**, being syntactic predicates, are not axioms in $\mathcal{M}[\![S]\!]$.)

In addition to the aliasing relations for control variables, we should also assert their types. Therefore, we implicitly add the axiom $v \in \{\mathit{true}, \mathit{false}\}$ to $\mathcal{M}[\![S]\!]$ for every control variable v in $\mathcal{V}(S)$.

There are also axioms relating the action predicate $\mathit{Act}(S)$ to the action predicates of its components. For example, the axioms for a **while** statement S would include the following:

- $\mathit{Act}(S) \;\equiv\; \mathit{Act}(S, \mathit{test}) \vee \mathit{Act}(S, \mathit{body})$
- $\neg(\mathit{Act}(S, \mathit{test}) \wedge \mathit{Act}(S, \mathit{body}))$

The first asserts that the only actions of S are the *test* action and the actions of its body; the second asserts that the *test* action is not an action of the body. These and similar axioms are assumed for all the constructs and are not included. Note that these axioms are given only for compound statements; there is no such axiom for the assignment statement.

In the following description of the axioms, formal axioms are followed by their informal explanations. For any programming-language expression *exp*, I let *exp*() denote the expression obtained by replacing every identifier y in *exp* by the variable name $y()$.

6.3.1 Assignment

If S is the statement $\langle\, x := \mathit{exp}\,\rangle$, then $\mathcal{M}[\![S]\!]$ contains the following axioms:

1. $\mathit{Act}(S) \supset \mathit{at}(S)$
   The atomic statement S can be executed only when control is at S.

2. $\forall \eta : \{\mathit{at}(S) \wedge \mathit{exp}() = \eta\}\langle S\rangle\{\mathit{after}(S) \wedge x() = \eta\}$
   Executing S sets the value of x to *exp* and changes control from $\mathit{at}(S)$ to $\mathit{after}(S)$.

3. $\forall \nu : \{x(), \mathit{at}(S), \mathit{after}(S)\} \perp \nu \;\supset\; \nu \overset{S}{\not\leftarrow}$
   (Note that ν is a logical name variable.) The statement S does not modify any variable not aliased to x, $\mathit{at}(S)$, or $\mathit{after}(S)$.

4. $\Box\Diamond \mathit{Act}(S) \;\supset\; \Diamond\neg \mathit{at}(S)$
   There cannot be infinitely many actions of S while control remains forever at S. (The reader may find this easier to understand if he replaces the implication by a disjunction.) In other words, there can be only finitely many stuttering actions of S before the assignment is executed.

An understanding of these axioms for assignment is crucial to an appreciation of how action-axiom semantics works, so some further discussion of them is in order. The four axioms are indeed action axioms, since they describe the behavior of the assignment action S. The four axioms assert the following:

1. When the action *may* occur.
2. What changes to the state components executing the action *may* perform.
3. What state components the action may not change.
4. When the action *must* change the state.

Every atomic program action is described by four similar axioms.

Note that axioms 1-3 assert safety properties, while axiom 4 states a liveness property. From the axioms for the other statements, it will follow that in language L, if $at(S)$ ever becomes true, it can be made false only by executing action S. Axiom 2 asserts that $at(S)$ can then become false only when $after(S)$ becomes true and the assignment of exp to x occurs. In a richer language, executing another statement might make $at(S)$ become false—for example, by aborting the process containing statement S. However, Axioms 1-4 would still be valid.

Observe that Axiom 2 determines the value of x immediately after execution of S. However, it asserts nothing about x's value after the execution of any other action.

For language L, Axiom 4 implies that if $at(S)$ is true then eventually it will become false (thereby making $after(S)$ true). However, this depends upon the fact that that L does not have any form of unfair **cobegin**. The axiom is valid for more general languages that do have these features.

It is instructive to consider what these axioms imply in case statement S appears inside declarations that produce a type mismatch—say in which x is of type **integer** and exp of type **boolean**. The axioms for those declarations will imply that the value of x is always an integer and the value of exp is always a boolean. It then follows from Axiom 2 that executing an S action can never make $at(S)$ false, since doing so would require setting the value of x to a boolean, contradicting the axioms for the declarations. However, I have already observed that, for language L, $at(S)$ must eventually become false. Thus, the set of axioms for the incorrect program—the one producing a type mismatch in statement S—are contradictory, implying that only the empty set of behaviors satisfy them. However, in a richer language, if S were contained inside an unfair **cobegin**, then the axioms might not be contradictory, and might be satisfied by a behavior in which a process remained stalled forever with $at(S)$ true. In this case, the type mismatch would force that process to "die", allowing other processes to proceed.

6.3.2 The if Statement

If S is the statement **if** $\langle\, exp \,\rangle$ **then** $\ldots$, then the following axioms are in $\mathcal{M}[\![S]\!]$. They are the standard four action axioms—in this case, for the *test* action. Note their similarity to the corresponding axioms for the assignment statement.

1. $Act(S, test) \supset at(S, test)$
   The test can be executed only when control resides at it.

2. $\{at(S, test)\}\langle S, test\rangle\{[at(S, then) \wedge exp()] \vee [at(S, else) \wedge \neg exp()]\}$
   Control remains at the beginning of the test until it either reaches the entry point of the **then** clause with exp true, or else it reaches the entry point of the **else** clause with exp false.

3. $\forall \nu : \{at(S, test), after(S, test)\} \perp \nu \supset \nu \overset{S,test}{\not\leftarrow}$
   The test does not modify any variable it shouldn't. (Again, ν is a logical name variable.)

4. $\Box\Diamond Act(S, test) \supset \Diamond\neg at(S, test)$
   There can be only finitely many stuttering actions of the test before it is really executed. This is the only liveness axiom for the **if** statement.

6.3.3 The while Statement

The axioms for the statement **while** $\langle\, exp \,\rangle$ **do** $\ldots$ are analogous to the ones for the **if** statement, and are given without comment.

1. $Act(S, test) \supset at(S, test)$

2. $\{at(S, test)\}\langle S, test\rangle\{(at(S, body) \wedge exp()) \vee (after(S) \wedge \neg exp())\}$

3. $\forall \nu : \{at(S, test), after(S, test)\} \perp \nu \supset \nu \overset{S,test}{\not\leftarrow}$

4. $\Box\Diamond Act(S, test) \supset \Diamond\neg at(S, test)$

6.3.4 The new Statement

The **new** statement is a declaration. If it has no **init** clause, then it performs no new action. The axioms describing this statement therefore do not follow the pattern for action axioms followed by the preceding statements. Instead, they assert relations that hold throughout the execution.

If S is the statement

$$\textbf{new}\ x : \mathit{type}\ \ \textbf{in} \ldots$$

then the following axiom is in $\mathcal{M}[\![S]\!]$, where we identify **integer** with the set $\mathcal{Z}$ and **boolean** with the set $\{\mathit{true}, \mathit{false}\}$.

1. $x(S) \in \mathit{type}$
   The value of x is always consistent with the type declaration.

If S is the statement

$$\textbf{new}\ x : \mathit{type}\ \ \textbf{alias}\ \mathit{exp}\ \ \textbf{in} \ldots$$

then $\mathcal{M}[\![S]\!]$ contains the above axiom plus the following:

2. $x(S) = \mathit{exp}()$
   The aliasing relation always holds.

If S is the statement

$$\textbf{new}\ x : \mathit{type}\ \ \textbf{init}\ \mathit{exp}\ \ \textbf{in} \ldots$$

then the following axioms hold. The first is, of course, the same as for the other versions of the **new** statement. The last four are the action axioms for the initial-assignment action, following the standard pattern. They are almost identical to the corresponding axioms for the assignment statement, the only difference (in axiom 3 below) indicating that the **init** clause performs an assignment to the variable $x(S)$ declared in the **new** statement rather than to the undeclared variable $x()$.

1. $x(S) \in \mathit{type}$
2. $\mathit{Act}(S, \mathit{init}) \supset \mathit{at}(S, \mathit{init})$
3. $\forall \eta : \{\mathit{at}(S, \mathit{init})\ \wedge\ \mathit{exp}() = \eta\}\langle S\rangle\{\mathit{after}(S, \mathit{init})\ \wedge\ x(S) = \eta\}$
4. $\forall \nu : \{x(S), \mathit{at}(S), \mathit{after}(S)\} \perp \nu\ \supset\ \nu \overset{S,\mathit{init}}{\not\hookleftarrow}$
5. $\Box\Diamond \mathit{Act}(S, \mathit{init})\ \supset\ \Diamond\neg \mathit{at}(S, \mathit{init})$

6.3.5 The cobegin Statement

If S is the statement

cobegin $S_1 \,\square\, \ldots \,\square\, S_n$ **coend**

then the following axiom is in $\mathcal{M}[\![S]\!]$.

1. $\forall i$ s.t. $1 \leq i \leq n : (\Box\Diamond Act(S)) \supset (\Box\Diamond Act(S, i))$
   If S performs infinitely many actions, then each process of S performs infinitely many actions. In other words, if S is never starved, then no subprocess of S is starved. This is the fairness axiom.

6.3.6 Sequences of Statements

No new axioms are needed for the sequence of statements $S_1; \ldots ; S_n$. All necessary properties are obtained from the aliasing relations among its control variables, the relations among its action predicates, and the Composition Principle.

6.3.7 A Complete Program

If Π is a complete program, then the only additional axiom in $\mathcal{M}[\![\Pi]\!]$ is:

1. $in(\Pi) \supset Act(\Pi, body)$
   The complete program never stops executing until it reaches the end, whereupon $in(\Pi)$ becomes false.

This axiom asserts the absence of any external actions while control is in progam Π, reflecting the absence of any explicit input or output in language L.

7 Other Language Features

While I have given a formal semantics only for the simple language L, action-axiom semantics can be used to describe a wider variety of concurrent programming language constructs than any other method I know of. In this section, I will consider a few interesting constructs. In doing so, I will not bother to give the usual axioms that describe the relations among control variables and among action predicates.

7.1 Constructs That Constrain Their Environment

Most language constructs constrain the behavior of their components. For example, an **if** statement determines when its **then** and **else** clauses can be executed. The following three language constructs constrain the behavior of a larger program containing them. They are therefore impossible to specify in a compositional, purely behavioral semantics. It is the Composition Principle that makes them expressible with action-axiom semantics.

7.1.1 The assign processor Command

As described above, the statement

assign processor to ...

directs the compiler to guarantee that the body of the statement gets its share of computing cycles, so it is not starved. This is expressed by the axiom:

$$in(S) \supset \Diamond Act(S)$$

7.1.2 Atomic Actions

One might want to introduce "angle brackets" as a language construct, so $\langle S \rangle$ denotes that S is to be executed as an indivisible atomic action. This is done by requiring that no other actions are interleaved with the executions of S, expressed formally by:

$$Act(\langle S \rangle) \;\supset\; in(\langle S \rangle) \trianglelefteq Act(\langle S \rangle)$$

7.1.3 Write Protection

Imagine a situation in which one wants the variable x to be modified only in a particular statement, but to be accessible elsewhere. This might be expessed by the following statement S:

encapsulate x **in** S'

The semantics of this statement are described formally by:

$$\forall \eta : x() = \eta \;\supset\; (\neg Act(S)) \lhd (x() = \eta)$$

which asserts that the value of x remains unchanged while any action not in S is executed.

7.2 Synchronization and Communication

The bread and butter of concurrent programming language constructs are the synchronization and interprocess communication mechanisms. I will discuss only two.

7.2.1 Semaphores

The usual semaphore P and V operations are variants of the atomic assignment statement: $P(s)$ looking much like the assignment $\langle\, s := s - 1 \,\rangle$ and $V(s)$ looking like $\langle\, s := s + 1 \,\rangle$. There are two basic differences. First of all, the $P(s)$ operation may be performed only when s is positive. One way of expressing this is to change the first axiom of the assignment statement to:

$$Act(P(s)) \;\supset\; (at(P(s)) \wedge s > 0)$$

However, this would require changing other axioms, since deadlock is represented by the absence of any possible actions, and the axiom given above for the complete program asserts that this is impossible.

The other way of handling this is to allow only stuttering actions of $P(s)$ to occur when $s \leq 0$. This is achieved by replacing the second axiom of the assignment statement with the following:

$$\forall \eta : \{at(P(s)) \wedge (x() = \eta)\}\; \langle P(s) \rangle \; \{after(P(s)) \wedge (x() = \eta - 1 \geq 0)\}$$

The second change that must be made to the assignment axioms is in the liveness condition. We can no longer require that an infinite number of actions of $P(s)$ cause the operation to be completed, since they might all occur when s has the value zero. Several liveness axioms have been proposed for the semaphore. Probably the most common are weak liveness, expressed by

$$(\Box(s > 0) \land \Box\Diamond Act(P(s))) \supset \Box\neg at(P(s))$$

and strong liveness, expressed by

$$(\Box\Diamond(s > 0) \land \Box\Diamond Act(P(s))) \supset \Box\neg at(P(s))$$

(They are discussed in [12].) In both these cases, the $V(s)$ operation is just an ordinary atomic assignment.

More complicated versions of the semaphore impose a specific queueing discipline, like first-come-first-served, on the execution of competing $P(s)$ operations. They may require adding a queue of waiting processes to the state, plus predicates to describe the state of the queue.

7.2.2 CSP-Like Communication Primitives

The easiest way to model the CSP "!" and "?" operations is in terms of channels. We include the operations $\langle\, x?\xi\,\rangle$ and $\langle\, exp!\xi\,\rangle$ for any variable x and expression *exp*. They denote CSP-like synchronous communication over a channel named ξ. We modify the **cobegin** statement by adding a clause of the form **channels** $\xi_1, \ldots, \xi_m$, which declares the channel names ξ_i.

As explained in [9], we consider communication actions to be actions of the channel, so $Act(S)$ is identically *false* if S is a ! or ? operation. A channel ξ has a separate atomic action for every pair of statements $\langle\, x?\xi\,\rangle$, $\langle\, exp!\xi\,\rangle$ contained in different clauses of the **cobegin** in which ξ is declared. This atomic action is axiomatized much like the assignment statement $\langle\, x := exp\,\rangle$, except that its execution changes the values of the four control variables $at(x?\xi)$, $after(x?\xi)$, $at(exp!\xi)$, and $after(exp!\xi)$.

To do this formally, we must extend our variable-naming convention in the obvious way to channel variables and add new syntactic predicates $S, \gamma \in\,!\,v$ and $S, \gamma \in\,?\,v$ to assert that the substatement S, γ is a ! or ? operation of the channel named v. The safety axiom for the declaration of channel ξ will be something like:

$$\begin{array}{l}\forall \gamma, \mu, i, j \text{ s.t. } i \neq j :\ S, i, \gamma \in\,!\,\xi(S) \land S, j, \mu \in\,?\,\xi(S) \supset \\ \quad \forall \eta : \{at(S, i\gamma) \land at(S, j, \gamma) \land \rho_{S,i,\gamma}(S, i, \gamma, left()) = \eta\} \\ \qquad\qquad \langle\, \xi(S)\,\rangle\ \{after(S, i\gamma) \land after(S, j, \gamma) \land \rho_{S,j,\mu}(S, j, \mu, left()) = \eta\}\end{array}$$

Note that $\rho_{S,i,\gamma}(S, i, \gamma, \mathit{left}())$ is the *exp* of *exp*$!\xi$, with all component variables appropriately renamed, and similarly for $\rho_{S,j,\mu}(S, j, \mu, \mathit{left}())$.

It is straightforward to extend this approach to guarded communication commands such as $\langle\, exp \rightarrow x?\xi\,\rangle$, which means that the communication action may be carried out only if *exp* has the value *true*. The new safety axiom is obtained from the above in much the same way that the safety axiom for the $P(s)$ semaphore operation is obtained from the corresponding axiom for the assignment statement—the guards here playing the part of the enabling condition $s > 0$ for the $P(s)$ operation.

There are several different choices of liveness properties that one can require of these channels. They are all basically simple to express with temporal logic formulas. However, their formal statement requires some careful manipulation of syntactic predicates, which I won't bother doing.

The safety properties of CSP-like communication primitives are expressed more easily with a formal semantics based only upon externally observable actions, such as [11]. When shared variables are not allowed, such a semantics can define the meaning of a process as the set of possible communications it can engage in. However, this kind of semantics does not seem capable of handling liveness properties easily.

7.3 Procedures

Although language L does not have procedures, its **new** statement contains the basic mechanism needed for procedure calls. A call of a nonrecursive procedure can be simulated by replacing the procedure call by **new** statements plus the body of the procedure. For example, let *proc* be a procedure with a declaration

$$\mathbf{procedure}\ \mathit{proc}(a : \mathbf{integer},\ \mathbf{var}\ b : \mathbf{boolean}\ )\ \ \mathit{body}$$

in which its first argument is call by value and its second is call by name. The call $\mathit{proc}(x + y, z)$ can be translated to

$$\begin{array}{l}\mathbf{new}\ a\ \ \mathbf{init}\ x + y\ \ \mathbf{in}\ \ \mathbf{new}\ b\ \ \mathbf{alias}\ z \\ \qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \ \mathit{body}\ \ \mathbf{ni}\ \ \mathbf{ni}\end{array}$$

To handle call by reference parameters, one needs to introduce pointer variables into the language. Of course, aliasing and procedure calls become more interesting when pointers and arrays are introduced, but a discussion of the problems raised by pointers and arrays is beyond the scope of this paper.

While this method of handling procedure calls works only for nonrecursive procedures, the basic idea applies to recursive ones as well. Replacing a procedure call by the body of the procedure produces an infinite program text for recursive procedures; but nowhere have I made use of the assumption that the program text is finite. Of course, the compositional method of recursively defining $\mathcal{M}[\![S]\!]$ no longer terminates with a finite set of axioms. However, the definition can be viewed as an algorithm for enumerating an infinite collection of axioms.

Thus, adding recursion means that $\mathcal{M}[\![S]\!]$ consists of an infinite set of axioms. It is in this case that the distinction between a semantics and a proof system becomes evident. An infinite set of axioms is unsatisfactory as a proof system, because ordinary logic provides no way of deducing a conclusion whose correctness is based upon an infinite set of assumptions. Such deductions are required to prove nontrivial properties of recursive programs. Thus, I have not provided a proof system for programs with recursive procedures.

On the other hand, a semantics is concerned with validity, not proof. The meaning of a program Π is the set of behaviors that satisfy the axioms in $\mathcal{M}[\![\Pi]\!]$, and this is well-defined even for an infinite set of axioms. The problem of proof systems is discussed in the conclusion.

7.4 More General Types and Aliasing

Let us now consider a language in which a type mismatch does not produce an illegal program, but generates "incorrect" behavior. As mentioned earlier, this requires adding predicates of the form $type(x) = \ldots$, which are syntactic predicates if types can be determined syntactically and state predicates if types are dynamic.

First, suppose that a type mismatch in the assignment $x := exp$ causes x to be set nondeterministically to any value in its range. This is easily represented by changing axiom 2 of the assignment to the following, where $type\_valid(x, \eta)$ is true if and only if the type of x permits it to be assigned the value η.

$$\begin{array}{l}\forall \eta : \{at(S) \wedge exp() = \eta\}\ \langle S \rangle \\ \qquad\qquad \{after(S) \wedge (x() = \eta \vee \neg type\_valid(x(), \eta))\}\end{array}$$

Next, suppose that a type mismatch causes the assignment to "hang up", effectively deadlocking the process. This requires that axiom 4 be changed so it does not demand termination in this case. There are several different liveness requirements one could make in this case, since the value of the expression *exp* could change. One reasonable possibility is the following:

$$\Box\Diamond(Act(S) \wedge \mathit{type\_valid}(x(), exp())) \supset \Diamond\neg at(S)$$

Allowing a more general form of aliasing, such as the one defined in [10], presents a similar problem if one requires that an assignment which would violate an aliasing constraint cause the process to hang up. One approach to this is to put the aliasing constraints in the state, just as I did with type constraints. The new state components would correspond to the "location" values often used to handle aliasing.

7.5 Nonatomic Operations

Every construct that I have mentioned specifies the atomic actions. For example, I have defined the semantics only of an atomic assignment statement. It is easy to give the semantics of an assignment statement with smaller atomic operations. For example, an assignment

$$\langle x \rangle := \langle exp \rangle$$

in which the evaluation of *exp* and the changing of x are distinct atomic operations can be represented by

$$\langle t := exp \rangle;\ \langle x := t \rangle$$

where t is an implicit variable. A similar translation is possible when the evaluation of the right-hand side is broken into smaller atomic operations; it is described in [4].

The situation changes when no atomicity is specified. For example, consider an assignment statement $x := y + 1$ that has the expected effect only if x and y are not modified by any other operation during the course of its execution. If any such modification does take place, then x may be set to any value consistent with its type. We can think of this assignment as a compound statement for which we know nothing about its internal structure except its partial correctness property (when executed alone) and the fact that it always terminates (unless the process executing it is starved).

Handling such nonatomic operations requires a new class of state predicate—the "generalized dynamic logic" predicates $[S]P$ introduced in [7]. The second assignment axiom for an atomic assignment is replaced by the following one for a nonatomic assignment $x := exp$:

$$\forall \eta : \{in(S) \wedge [S](x() = \eta)\}\ \langle S \rangle\ \{after(S) \wedge x() = \eta\}$$

Note that the rules for reasoning about these generalized dynamic logic predicates imply that

$$at(S) \supset ([S](x() = \eta) \equiv (exp = \eta))$$

The liveness axiom for a nonatomic assignment is simply

$$\Box\Diamond Act(S) \supset \Diamond\neg in(S)$$

8 Conclusion

I have given an axiomatic semantics for a simple concurrent programming language L, and have indicated how the same method can be applied to more complicated language constructs. Most of this paper has been devoted to developing the fundamental ideas upon which the method is based. The axioms themselves are reasonably simple—simple enough so I feel that they do provide an understanding of the language constructs. For example, the difference between a weakly fair and a strongly fair semaphore is described quite concisely and precisely by their respective axioms.

A programming language semantics provides a logical basis for a proof system for reasoning about programs in the language. One can talk about the soundness and completeness of the proof system in terms of the semantics. Note that it makes no sense to talk about soundness and completeness of the semantics. Indeed, the semantics $\mathcal{M}[\![S]\!]$ of a program can include contradictory axioms; this simply means that there are no valid behaviors for S, so there is something wrong with the program, not with the semantics.

The obvious task now is to investigate existing proof systems in terms of this semantics. Unfortunately, such an undertaking is beyond the scope of this paper. However, some brief remarks are in order. The Generalized Hoare Logic (GHL) presented in [4] and [9] introduced *at*, *in*, and *after* as predicates rather than variables. The relation $\|$ used in [4] is just the relation $\perp$.

The semantics of GHL formulas was not stated with sufficient precision in [4], since the relation between the statement S and its name, denoted $'S'$, was never made clear. A close examination of GHL reveals that there is an implicit complete program Π, and that if S is the substatement Π, γ of Π, then a formula written in terms of S should really be written in terms of Π, γ.

To verify the soundness of GHL, one must express the GHL formula $\{P\}\, S\, \{Q\}$ as a temporal logic formula. As explained in [9], it suffices to consider the case $P = Q$, for which the definition is simply:

$$\{P\}\, S\, \{P\} \stackrel{\text{def}}{=} \{P\}\langle \Pi, \gamma\rangle\{P\}$$

where S is the substatement Π, γ of the implied complete program Π. The soundness of the general rules for reasoning about GHL formulas follows easily from their interpretation as temporal logic formulas. The soundness of the axioms and rules given in [4] for each language construct can be deduced from the axioms for the corresponding construct given here in Section 6.3, together with the Composition Principle.

As described in [9], other logical systems for proving safety properties of concurrent programs can be described in terms of GHL, so the soundness of GHL can be used to prove the soundness of the other systems. GHL is manifestly not a complete system for reasoning about concurrent programs, since it does not address questions of liveness. It is not clear how to use our semantics to prove completeness of GHL for the class of properties it can express.

A method for proving liveness properties of programs is given in [12]. It considers a simple language that is essentially the same as language L except without the **new** statement. The method explicitly assumes a complete program Π, and is based upon temporal logic plus the following single axiom:

Atomic Action Axiom: For any atomic action Π, γ of Π:

$$at(\Pi, \gamma) \supset \Diamond after(\Pi, \gamma)$$

To prove the soundness of this axiom, we must show that

$$\Box(in(\Pi, \gamma) \supset \Diamond Act(\Pi, \gamma))$$

holds for every substatement and atomic action Π, γ of Π. This is intuitively clear, since the language contains only fair **cobegin** statements, and is derivable from our axioms by induction on the size of Π. The above Atomic Action Axiom then follows easily from our liveness axiom for complete programs, the liveness axioms for the individual statements, plus the Composition Principle. The additional axioms given in [12] for weakly and strongly fair semaphore operations can similarly be derived from the ones I gave earlier.

Acknowledgements

I wish to thank Fred Schneider and Willem-Paul de Roever for their detailed comments on an earlier draft of this paper.

References

[1] Karl M. Abrahamson. *Decidability and Expressiveness of Logics of Processes.* Ph. D. Thesis, issued as Technical Report No. 80-08-01, Department of Computer Science, University of Washington. (August 1980).

[2] Denotational Semantics of Concurrency. J. W. de Bakker and J. C. Zucker. *Fourteenth ACM Symposium on the Theory of Computing*, San Francisco, (May, 1982), 153-158.

[3] Z. C. Chen and C. A. R. Hoare. Partial Correctness of Communicating Sequential Processes. *Proceedings of the Second IEEE International Conference on Distributed Computer Systems*, (1981) 1-12.

[4] L. Lamport. The "Hoare Logic" of Concurrent Programs. *Acta Informatica 14* (1980), 21-37.

[5] L. Lamport. "Sometime" Is Sometimes "Not Never". *Proceedings of the Seventh Annual ACM Conference on the Principles of Programming Languages*, (January 1980) 174-185.

[6] L. Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Prog. Logic and Sys. 5*, 2 (April 1983) 190-222.

[7] L. Lamport. Reasoning About Nonatomic Operations. *Proceedings of the Tenth Annual ACM Conference on the Principles of Programming Languages*, (January 1983) 28-37.

[8] L. Lamport. What Good Is Temporal Logic? *Information Processing 83*, R. E. Mason (ed.), Elsevier Science Publishers, North Holland (1983), 657-668.

[9] L. Lamport and F. B. Schneider. The Hoare Logic of CSP and All That. *ACM Transactions on Prog. Logic and Sys. 6*, 2 (April 1984) 281-296.

[10] L. Lamport and F. B. Schneider. Constraints: A Uniform Approach to Aliasing and Typing. To appear in *Proceedings of the Twelfth Annual ACM Conference on the Principles of Programming Languages*, (January 1985).

[11] R. Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science, Number 92. Springer-Verlag, Berlin (1980).

[12] S. S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Trans. on Prog. Lang. and Systems 4*, 3 (1982), 455-495.

[13] A. Pnueli. The Temporal Logic of Programs. *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science* (1977), Providence Rhode Island.

[14] H. Barringer, R. Kuiper. and A. Pnueli. Now You May Compose Temporal Logic Specifications. *Sixteenth ACM Symposium on the Theory of Computing*, (May, 1984).

[15] J. Sifakis. A Unified Approach for Studying the Properties of Transition Systems. *Theoretical Computer Science 18* (1982), 227-258.