Abstractions of Time

Eric C.R. Hehner
University of Toronto

Introduction

The execution time of programs has been modeled, or measured, or calculated, in a variety of ways. This paper is concerned with measurements of time that are part of a formal semantics of programs. A semantics can enable us to calculate the execution time of programs quite precisely. This is necessary for applications known as real-time. For other applications, a more abstract measure of time, called recursive time, is both sufficient and convenient. More abstract still is the measure of time used in a total correctness semantics; time is reduced to a single bit that distinguishes between finite and infinite execution time. Continuing to the extreme, we find partial correctness, where time has been eliminated. Between the extremes of real-time and partial correctness there are other points of interest, such as the quantified time of temporal logic, and the timed processes of ATP.

It is reasonable to retain several theories with different abstractions of time if they allow us to trade simplicity against accuracy. We use a simple theory whenever we can, moving to a more complex theory when more accuracy is required. But if one theory is both simpler and more accurate than another, requiring less formal labor to obtain more timing information, then the other theory should be discarded. As we shall see, that is indeed the case.

Different abstractions of time can best be compared if they are all presented, as much as possible, within one semantic framework. The framework used in this paper is characterized by the following principles.

• We first decide what quantities are of interest, and introduce a variable for each such quantity. A variable may represent input to a computation, or output from a computation.

• A specification is a boolean expression whose variables represent the quantities of interest. A specification is implemented on a computer when, for any values of the input variables, the computer generates (computes) values of the output variables to satisfy the specification. In other words, we have an implementation when the specification is true of every computation. (Note that we are specifying computations, not programs.)

• A program is a specification that has been implemented.

Suppose we are given specification S. If S is a program, we can execute it. If not, we have some programming to do. That means finding a program P such that $S \Leftarrow P$ is a theorem; this is called refinement. Since S is implied by P, all computer behavior satisfying P also satisfies S. We might refine in steps, finding specifications R, Q, ... such that $S \Leftarrow R \Leftarrow Q \Leftarrow \ldots \Leftarrow P$.

Notation

Here are all the notations used in this paper, arranged by precedence level.

0. *true* *false* ( ) numbers names
1. juxtaposition
2. superscript subscript underscore ::
3. $\times$ / **div** ! $\downarrow$
4. + –
5. = $\neq$ < > $\leq$ $\geq$
6. $\neg$
7. $\wedge$
8. $\vee$
9. $\Rightarrow$ $\Leftarrow$
10. := **if then else** **while do**
11. $\lambda\cdot$ $\forall\cdot$ $\exists\cdot$ $\Sigma\cdot$ $\Pi\cdot$; $\|$ $\square$ $\diamond$ $\bigcirc$
12. $=$ $\Longrightarrow$ $\Longleftarrow$

On level 2, superscripting, subscripting, and underscoring serve to bracket all operations within them. Juxtaposition associates from left to right, so that $a\,b\,c$ means $(a\,b)\,c$. The infix operators / – associate from left to right. The infix operators $\times$ + $\wedge$ $\vee$; $\|$ are associative (they associate in both directions). On levels 5, 9, and 12 the operators are continuing; for example, $a = b = c$ neither associates to the left nor associates to the right, but means $a = b \wedge b = c$. On any one of these levels, a mixture of continuing operators can be used. For example, $a \leq b < c$ means $a \leq b \wedge b < c$. On level 10, the precedence does not apply to operands that are surrounded by the operator. On level 11, the function notation $\lambda v{:}\, D\cdot b$ surrounds D, so the precedence does not apply to D; it applies to b. Similarly for $\forall\cdot$ $\exists\cdot$ $\Sigma\cdot$ $\Pi\cdot$. The operators $=$ $\Longrightarrow$ $\Longleftarrow$ are identical to = $\Rightarrow$ $\Leftarrow$ except for precedence.

Partial Correctness

For simplicity, we'll start with partial correctness, which ignores time. We can observe the initial state of memory, represented by variables x, y, ..., whose values are provided as input. We can also observe the final state of memory, represented by variables x', y', ..., whose values are the

result of a computation. Specification S is implementable if and only if

$$\forall x, y, \ldots \cdot \exists x', y', \ldots \cdot S$$

As specification language, we allow ordinary logic, arithmetic, notations that are specific to the application, and any other well-defined notations that the specifier considers convenient, including notations invented on the spot for the purpose. We also include in our specification language the following notations.

$ok = x'=x \wedge y'=y \wedge \ldots$

$x := e = x'=e \wedge y'=y \wedge \ldots$

$\textbf{if } b \textbf{ then } P \textbf{ else } Q = b \wedge P \vee \neg b \wedge Q$

$= (b \Rightarrow P) \wedge (\neg b \Rightarrow Q)$

$P;Q = \exists x'', y'', \ldots \cdot$ (substitute $x'', y'', \ldots$ for $x', y', \ldots$ in P)

$\wedge$ (substitute $x'', y'', \ldots$ for $x, y, \ldots$ in Q)

The notation ok specifies that the final values of all variables equal the corresponding initial values. A computer can satisfy this specification by doing nothing. Let us take ok to be a program. In the assignment notation, x is any state variable and e is any expression in the domain of x. Let us take assignments in which expression e does not use primed variables, and uses only those operators that are implemented, to be programs. In the **if** notation, if b does not use primed variables, and uses only those operators that are implemented, and P and Q are programs, then let us take $\textbf{if } b \textbf{ then } P \textbf{ else } Q$ to be a program. The specification $P;Q$ can be implemented by a computer that first behaves according to P, then behaves according to Q, with the final values from P serving as initial values for Q. It therefore describes sequential execution. If P and Q are programs, let us take $P;Q$ to be a program.

From these definitions, many useful laws of programming can be proven. Here are a few. Note that P, Q, R, and S can be any specifications, not just programs.

$ok; P = P; ok = P$	Identity Law
$P; (Q; R) = (P; Q); R$	Associative Law
$\textbf{if } b \textbf{ then } P \textbf{ else } P = P$	Idempotent Law
$\textbf{if } b \textbf{ then } P \textbf{ else } Q = \textbf{if } \neg b \textbf{ then } Q \textbf{ else } P$	Case Reversal Law
$P = \textbf{if } b \textbf{ then } b \Rightarrow P \textbf{ else } \neg b \Rightarrow P$	Case Creation Law
$P \vee Q; R \vee S = (P; R) \vee (P; S) \vee (Q; R) \vee (Q; S)$	Distributive Law
$(\textbf{if } b \textbf{ then } P \textbf{ else } Q) \wedge R = \textbf{if } b \textbf{ then } P \wedge R \textbf{ else } Q \wedge R$	Distributive Law

and all other operators in place of $\wedge$ including sequential execution:

$(\textbf{if } b \textbf{ then } P \textbf{ else } Q); R = \textbf{if } b \textbf{ then } (P; R) \textbf{ else } (Q; R)$	
$x := \textbf{if } b \textbf{ then } e \textbf{ else } f = \textbf{if } b \textbf{ then } x := e \textbf{ else } x := f$	Functional-Imperative Law
$x := e; P =$ (for x substitute e in P)	Substitution Law

For this paper, we need only the four programming notations we have introduced, but we need one more way to create programs. Any implementable specification S is a program if a program P is provided such that $S \Leftarrow P$ is a theorem. To execute S, just execute P. One can imagine a library of specifications that have become programs by being provided with implementations. Furthermore, recursion is allowed: within P, we can use specification S as a program. A computer executes S by behaving according to program P, and whenever S is encountered again, the behavior is again according to P.

To illustrate, here is a small problem. If ! is the factorial function, then $(a+b)! / (a! \times b!)$ is the number of ways to partition $a+b$ things into a things and b things. In natural variables x, a, and b, the specification is

$$x' = (a+b)! / (a! \times b!)$$

There are many ways to refine this specification. One of them is

$$x' = (a+b)! / (a! \times b!) \quad \Leftarrow \quad x := 1;\ x' = x \times (a+b)! / (a! \times b!)$$

which is proven by one application of the Substitution Law. The right side uses a new specification that requires refinement.

$$x' = x \times (a+b)! / (a! \times b!) \quad \Leftarrow$$
$$\textbf{if } a=0 \vee b=0 \textbf{ then } ok$$
$$\textbf{else } (x := x/a/b \times (a+b-1) \times (a+b);\ a := a-1;\ b := b-1;\ x' = x \times (a+b)! / (a! \times b!))$$

The proof uses three applications of the Substitution Law and some simplification. The right side uses the specification we are refining recursively. We have not used any new, unrefined specifications, so we are done.

If ! is not an implemented operator, then $x := (a+b)! / (a! \times b!)$ is not a program. Whether it is or not, we may still refine it, to obtain the following solution.

$$x := (a+b)! / (a! \times b!) \quad \Leftarrow$$
$$\textbf{if } a=0 \vee b=0 \textbf{ then } x := 1$$
$$\textbf{else } (\ a := a-1;\ b := b-1;\ x := (a+b)! / (a! \times b!);$$
$$a := a+1;\ b := b+1;\ x := x/a/b \times (a+b-1) \times (a+b)\ )$$

The occurrence of $x := (a+b)! / (a! \times b!)$ on the right side is a recursive call.

Note that we have loops in the refinement structure, but no looping construct in our programming notations. This avoids a lot of semantic complications.

Real-Time

To talk about time, we just add a time variable t. We do not change the theory at all; the time variable is treated just like any other variable, as part of the state. The interpretation of t as time is justified by the way we use it. In an implementation, the other variables x, y, ... require space

in the computer's memory, but the time variable t does not; it simply represents the time at which execution occurs.

We use t for the initial time, the time at which execution starts, and t' for the final time, the time at which execution ends. To allow for nontermination we take the domain of time to be a number system extended with an infinite number ∞.

Time cannot decrease, therefore a specification S with time is implementable if and only if

$$\forall x, y, \ldots, t \cdot \exists x', y', \ldots, t' \cdot S \wedge t' \geq t$$

For each initial state, there must be at least one satisfactory final state in which time has not decreased.

To obtain the real execution time, just insert time increments as appropriate. Of course, this requires intimate knowledge of the implementation, both hardware and software; there's no way to avoid it. Before each assignment $x := e$ insert $t := t+u$ where u is the time required to evaluate and store e. Before each conditional **if** b **then** P **else** Q insert $t := t+v$ where v is the time required to evaluate b and branch. Before each call S insert $t := t+w$ where w is the time required for the call and return. For a call that is implemented in-line, this time will be zero. For a call that is executed last in a refinement, it may be just the time for a branch. Sometimes it will be the time required to push a return address onto a stack and branch, plus the time to pop the return address and branch back. We could place the time increase after each of the programming notations instead of before; by placing it before, we make it easier to use the Substitution Law.

Any specification can talk about time: $t' = t+e$ specifies that e is the execution time; $t' \leq t+e$ specifies that e is an upper bound on the execution time; and $t' \geq t+e$ specifies that e is a lower bound on the execution time.

In the partition example, suppose that the **if**, the assignment, and the call each take time 1. Let $\downarrow$ be the "minimum" operator. Inserting time increments, we can easily prove

$$t' = t + 5\times(a \downarrow b) + 3 \quad \Longleftarrow \quad t := t+1;\ x := 1;\ t := t+1;\ t' = t + 5\times(a \downarrow b) + 1$$

$$t' = t + 5\times(a \downarrow b) + 1 \quad \Longleftarrow$$

$$\begin{aligned}
&t := t+1;\\
&\textbf{if } a=0 \vee b=0 \textbf{ then } ok\\
&\textbf{else } (\ t := t+1;\ x := x/a/b\times(a+b-1)\times(a+b);\ t := t+1;\ a := a-1;\ t := t+1;\ b := b-1;\\
&\qquad t := t+1;\ t' = t + 5\times(a \downarrow b) + 1\ )
\end{aligned}$$

So the execution time is $5\times(a \downarrow b) + 3$. The Law of Refinement by Parts says that we can conjoin specifications that have similar refinements, so without any further proof we have

$x' = (a+b)! / (a!\times b!) \wedge t' = t + 5\times(a\downarrow b) + 3 \Longleftarrow$
$\quad t := t+1;\ x := 1;\ t := t+1;\ x' = x \times (a+b)! / (a!\times b!) \wedge t' = t + 5\times(a\downarrow b) + 1$

$x' = x \times (a+b)! / (a!\times b!) \wedge t' = t + 5\times(a\downarrow b) + 1 \Longleftarrow$
$\quad t := t+1;$
$\quad \textbf{if } a=0 \vee b=0 \textbf{ then } ok$
$\quad \textbf{else } (\ t := t+1;\ x := x/a/b\times(a+b-1)\times(a+b);\ t := t+1;\ a := a-1;\ t := t+1;\ b := b-1;$
$\quad\quad t := t+1;\ x' = x \times (a+b)! / (a!\times b!) \wedge t' = t + 5\times(a\downarrow b) + 1\ )$

When we place a time increment $t := t+e$ in a program, the expression e can depend on the values of variables; it doesn't have to be a constant. If we cannot say precisely what the time increment is, perhaps we can say what its bounds are: $a \le t'-t \le b$.

Recursive Time

To free ourselves from having to know implementation details, we allow any arbitrary scheme for inserting time increments $t := t+u$ into programs. Each scheme defines a new measure of time. In the recursive time measure, each recursive call costs time 1 , and all else is free. This measure neglects the time for straight-line and branching programs, charging only for loops.

In the recursive measure, our earlier example becomes

$t' = t + a\downarrow b \Longleftarrow x := 1;\ t' = t + a\downarrow b$

$t' = t + a\downarrow b \Longleftarrow$
$\quad \textbf{if } a=0 \vee b=0 \textbf{ then } ok$
$\quad \textbf{else } (x := x/a/b\times(a+b-1)\times(a+b);\ a := a-1;\ b := b-1;\ t := t+1;\ t' = t + a\downarrow b)$

Since implication is reflexive, we can refine any specification by itself. For example,

$x'=2 \Longleftarrow x'=2$

With this refinement, $x'=2$ is a program that claims x will have the final value 2 , but it doesn't say when. Now let's add time. If we specify that execution time is finite, say $t' = t+n$, and insert the time increment before the recursive call we find that

$x'=2 \wedge t' = t+n \Longleftarrow t := t+1;\ x'=2 \wedge t' = t+n$

is not a theorem. The only implementable specification we can refine this way is $t'=\infty$:

$x'=2 \wedge t'=\infty \Longleftarrow t := t+1;\ x'=2 \wedge t'=\infty$

This specification says that execution takes forever.

In the partition example, suppose now that a and b are integer variables. We can prove the following specification of execution time

$$(0 \le a \le b \vee b < 0 \le a \;\Rightarrow\; t' = t+a)$$
$$\wedge\quad (0 \le b \le a \vee a < 0 \le b \;\Rightarrow\; t' = t+b)$$
$$\wedge\quad (a < 0 \wedge b < 0 \;\Rightarrow\; t' = \infty)$$

Total Correctness

In a total correctness semantics, the only question asked about time is whether it is finite or infinite. Since we only want to know one bit of information about time, we might consider using a boolean abstraction. Let s mean "execution starts at a finite time" and s' mean "execution ends at a finite time". The programming notations can remain as they were, satisfying the same axioms and laws, except that s replaces t. This sort of total correctness semantics has been suggested in [Hehner84] and in [Hoare92]. But there's a problem: we can no longer insert a time increment into a recursion. We have nothing to correspond to the tick of a clock. So we cannot account for the passage of time in a recursive refinement.

One solution to the problem is to abandon recursive refinement, and to invent a loop construct; a well-known syntax is **while** b **do** S. If we are to make use of our theory for programming (and surely that is its purpose), we must define **while** b **do** S for arbitrary specifications S, not just for programs. That is necessary so that we can introduce a loop and show that we have done so correctly, separately from the refinement of its body. There are essentially two ways to do it: as a limit of a sequence of approximations, or as a least (pre)fixed-point.

The limit of approximations works like this. Define

$$W_0 = \mathit{true}$$
$$W_{n+1} = \textbf{if } b \textbf{ then } (S;\ W_n) \textbf{ else } ok$$

Then

$$\textbf{while } b \textbf{ do } S = \forall n \cdot W_n$$

As an example, we will find the semantics of

$$\textbf{while } x \neq 1 \textbf{ do } x := x \textbf{ div } 2$$

in one integer variable x. We find

$$W_0 = \mathit{true}$$
$$W_1 = \textbf{if } b \textbf{ then } (x := x \textbf{ div } 2;\ \mathit{true}) \textbf{ else } ok$$
$$= x=1 \Rightarrow x'=1$$
$$W_2 = \textbf{if } b \textbf{ then } (x := x \textbf{ div } 2;\ x=1 \Rightarrow x'=1) \textbf{ else } ok$$
$$= 1 \le x < 4 \Rightarrow x'=1$$

Jumping to the general case, which we could prove by induction,

$$W_n = 1 \le x < 2^n \Rightarrow x'=1$$

And so

while $x \neq 1$ **do** $x := x$ **div** 2

$=$ $\forall n \cdot \ 1 \leq x < 2^n \Rightarrow x' = 1$

$=$ $1 \leq x \Rightarrow x' = 1$

In effect, we are introducing recursive time in disguise. W_n is the strongest specification of behavior that can be observed before time n, measured recursively.

The other way to define **while**-loops is as a least fixed-point. There are two axioms. The first

while b **do** S $=$ **if** b **then** (S; **while** b **do** S) **else** ok

says that a **while**-loop equals its first unrolling. Stated differently, **while** b **do** S is a solution of the fixed-point equation (in unknown W)

W $=$ **if** b **then** (S; W) **else** ok

The other axiom

$\forall \sigma, \sigma' \cdot$ (W $=$ **if** b **then** (S; W) **else** ok) $\Longrightarrow$ $\forall \sigma, \sigma' \cdot$ (W $\Longrightarrow$ **while** b **do** S)

(where σ is the state variables) says that **while** b **do** S is as weak as any fixed-point, so it is the weakest (least strong) fixed-point.

The two axioms we have just seen are equivalent to the following two axioms.

while b **do** S $\Longrightarrow$ **if** b **then** (S; **while** b **do** S) **else** ok

$\forall \sigma, \sigma' \cdot$ (W $\Longrightarrow$ **if** b **then** (S; W) **else** ok) $\Longrightarrow$ $\forall \sigma, \sigma' \cdot$ (W $\Longrightarrow$ **while** b **do** S)

The first of these says that a **while**-loop refines (implements) its first unrolling; it is a prefixed-point. The second says that **while** b **do** S is as weak as any prefixed-point, so it is the weakest prefixed-point. Though equivalent to the former definition, this definition has an advantage. From the prefixed-point definition it is easy to prove the fixed-point formulas, but the reverse proof is quite difficult. These (pre)fixed-point definitions introduce a form of induction especially for **while**-loops, a kind of **while**-loop arithmetic, in place of the arithmetic of a time variable.

The limit of approximations definition and the (pre)fixed-point definition agree when the body of a loop uses only programming notations, but they sometimes disagree when the body is an arbitrary specification. A famous example, in one integer variable x, is

while $x \neq 0$ **do if** $x > 0$ **then** $x := x - 1$ **else** $x' \geq 0$

According to the limit of approximations, this **while**-loop equals

$x \geq 0 \Rightarrow x' = 0$

According to the (pre)fixed-point, it equals

$x' = 0$

They differ on whether we should consider a computation to be terminating in the absence of any time bound.

A total correctness semantics makes the proof of invariance properties difficult, or even impossible. For example, we cannot prove

$x' \geq x$ $\Longleftarrow$ **while** b **do** $x' \geq x$

which says, quite reasonably, that if the body of a loop doesn't decrease x, then the loop doesn't decrease x. The problem is that the semantics does not allow us to separate such invariance properties from the question of termination. In the recursive time semantics, in place of

$$S \Leftarrow \textbf{while } b \textbf{ do } P$$

we write

$$S \Leftarrow \textbf{if } b \textbf{ then } (P;\ t := t+1;\ S) \textbf{ else } ok$$

and the proof of the invariance property

$$x' \geq x \Leftarrow \textbf{if } b \textbf{ then } (x' \geq x;\ t := t+1;\ x' \geq x) \textbf{ else } ok$$

is easy.

In practice, neither the limit of approximations nor the (pre)fixed-point definition is usable for programming. Instead, programming is split into partial correctness and termination argument. To prove

$$x \geq 1 \Rightarrow x' = 1 \Leftarrow \textbf{while } x \neq 1 \textbf{ do } x := x \textbf{ div } 2$$

we prove partial correctness, which is

$$x \geq 1 \Rightarrow x' = 1 \Leftarrow \textbf{if } x \neq 1 \textbf{ then } (x := x \textbf{ div } 2;\ x \geq 1 \Rightarrow x' = 1) \textbf{ else } ok$$

For termination we use a “variant” or “bound function” or “well-founded set”. In this example, we show that for $x>1$, x is decreased but not below 0 by the body $x := x \textbf{ div } 2$ of the loop. The bound function is again recursive time in disguise. We are showing that execution time is bounded by x. Then we throw away the bound, retaining only the one bit of information that there is a bound, so there is no incentive to find a tight bound. In the example, showing that x is a variant corresponds to the proof of

$$x \geq 1 \Rightarrow t' - t \leq x \Leftarrow \textbf{if } x \neq 1 \textbf{ then } (x := x \textbf{ div } 2;\ t := t+1;\ x \geq 1 \Rightarrow t' - t \leq x) \textbf{ else } ok$$

This linear time bound is rather loose; for about the same effort, we can prove a logarithmic time bound:

$$x \geq 1 \Rightarrow t' \leq t + \log x \Leftarrow \textbf{if } x \neq 1 \textbf{ then } (x := x \textbf{ div } 2;\ t := t+1;\ x \geq 1 \Rightarrow t' \leq t + \log x) \textbf{ else } ok$$

In any case, we can express the termination proof in exactly the same form as the partial correctness proof (though occasionally time must be measured by a tuple of numbers, rather than just a single number).

A total correctness formalism introduces all the formal machinery necessary to calculate time bounds, but in a disguised and unusable way. A proof of total correctness necessarily requires finding a time bound, then throws it away. Of all the abstractions of time, total correctness gives least benefit for effort. Furthermore, from Gödel and Turing we know that a complete and consistent theory in which termination can be expressed is impossible. Any total correctness theory will therefore be incomplete in its treatment of termination.

Temporal Logic

An interesting abstraction of time is offered by temporal operators such as $\Box$ (always), $\Diamond$ (sometime, eventually), and $\bigcirc$ (next). We want $\Box P$ to mean that P is true at all times during a computation, and $\Diamond P$ to mean that P is true at some time during a computation. Until now, we have assumed that only the initial and final states are observable, but for temporal operators we want intermediate states to be observable too. We keep t and t' for the initial and final execution time, but state variables x , y , ... are now functions of time. The value of x at time t is $x\,t$. An expression such as $x+y$ is also a function of time; its argument is distributed to its variable operands as follows: $(x+y)t = xt+yt$. The programming notations are redefined as follows.

$ok \;=\; t'=t$

$x := e \;=\; t' = t+1 \;\wedge\; x\,t' = e\,t \;\wedge\; y\,t' = y\,t \;\wedge\; \ldots$

$P;Q \;=\; \exists t'' : t \le t'' \le t' \cdot$ (substitute t'' for t' in P) $\wedge$ (substitute t'' for t in Q)

if b **then** P **else** $Q \;=\; bt \wedge P \;\vee\; \neg\, bt \wedge Q$

We can now talk about intermediate states. For example,

$x := x+3;\; x := x+4$

$=$ $\quad t'=t+2 \wedge x(t+1)=xt+3 \wedge x(t+2)=xt+7 \wedge yt=y(t+1)=y(t+2)$

As before, any implementable specification S is a program if a program P is provided such that $S \Leftarrow P$ is a theorem. Recursion is allowed if it is preceded by the passage of time. An assignment is assumed to take time 1 , but that is easily changed if one wants a different measure of time.

Before defining the temporal operators, here is a nice way to look at quantifiers. A quantifier is an operator that applies to functions. The quantifiers Σ and Π apply to functions that have a numeric result, and the quantifiers $\forall$ and $\exists$ apply to functions that have a boolean result (a function with a boolean result is called a predicate). If f is a function with numeric result, then Σf is the numeric result of applying f to all its domain elements and adding up all the results. Similarly Πf is the numeric result of applying f to all its domain elements and multiplying all the results. If p is a predicate, then $\forall p$ is the boolean result of applying p to all its domain elements and conjoining all the results. Similarly $\exists p$ is the boolean result of applying p to all its domain elements and disjoining all the results. For the sake of tradition, when a quantifier is applied to a function written as a λ-expression, the λ is omitted. For example, the application of Σ to $\lambda n : nat \cdot 1/2^n$ is written $\Sigma n : nat \cdot 1/2^n$, and the application of $\forall$ to $\lambda r : rat \cdot r<0 \vee r=0 \vee r>0$ is written $\forall r : rat \cdot r<0 \vee r=0 \vee r>0$.

This treatment of quantifiers allows us to write the Generalization and Specialization Laws as follows: if x is in the domain of p , then

$\forall p \implies px \implies \exists p$

A quantification such as $\forall x \cdot lost\ x$ can be written more briefly as $\forall lost$. With composition of operators, we can write deMorgan's Laws this way:

$\neg\forall p \;=\; \exists\neg p$

$\neg\exists p \;=\; \forall\neg p$

or even this way:

$\neg\forall \;=\; \exists\neg$

$\neg\exists \;=\; \forall\neg$

Let S be a specification. The extension of S, written $\underline{S}$ and pronounced "S extended", is defined as follows.

$\underline{S} \;=\; \lambda t'' .\; t \le t'' \le t' \cdot$ (substitute t'' for t in S)

Whatever S may say about time t, $\underline{S}$ extends it to all times from t to t'. Now we define

$\Box S \;=\; \forall \underline{S}$

$\Diamond S \;=\; \exists \underline{S}$

$\bigcirc S \;=\; \underline{S}(t{+}1)$

These definitions give us something close to Interval Temporal Logic [Moszkowski86]. We can prove deMorgan's Laws

$\neg\Box S \;=\; \Diamond \neg S$

$\neg\Diamond S \;=\; \Box \neg S$

and other identities of ITL, such as

$\Diamond S \;=\; \mathit{true}; S$

But we still have a time variable. We can say that x is constant like this: $\Box\, xt{=}xt'$. We can prove

$\bigcirc S \;=\; t'{=}t{+}1;\; S \;=\; t' \ge t{+}1 \wedge$ (substitute $t{+}1$ for t in S)

In Interval Temporal Logic, time is discrete. With the definitions of $\Box$ and $\Diamond$ given above, time can be discrete or continuous. If it is continuous, we might like to strengthen assignment as follows:

$x := e \;=\; t'{=}t{+}1 \;\wedge\; xt'{=}et \;\wedge\; (\Box\, yt'{=}yt) \;\wedge\; \ldots$

so that unaffected variables remain continuously constant, while x is unknown during the assignment and known to have its newly assigned value only at the end. With continuous time, the $\bigcirc$ operator no longer means "next", and is not particularly useful. If time is discrete, we can say that x never decreases like this: $\Box\, x(t{+}1) \ge xt$. If time is discrete and $t'{=}\infty$, we can prove the fixed-point equations

$\Box S \;=\; S \wedge \Box \bigcirc S$

$\Diamond S \;=\; S \vee \Diamond \bigcirc S$

Temporal logic considers time to be too holy to speak its name; it replaces $\forall t$, $\exists t$, and $t{+}1$ by $\Box$, $\Diamond$, and $\bigcirc$. We need quantifiers for many purposes, and we quantify over many things. The temporal operators $\Box$ and $\Diamond$ replace the usual quantifiers $\forall$ and $\exists$ only for quantifications over time, and then only in some cases. We needed $\exists$ over time to define sequential execution (*chop* in Interval Temporal Logic). With two sets of symbols to do similar

jobs, we are burdened with learning two sets of similar laws. By treating quantifiers as operators, and by defining extensions, we make the usual quantifiers just as convenient for time as the temporal operators. On the other hand, if extensions are used only in combination with quantifiers, we might still prefer to write $\Box S$ and $\Diamond S$ than to write $\forall \underline{S}$ and $\exists \underline{S}$. (Underscore is a poor notation anyway.)

As noted already, the "next" operator is useless for continuous time; in a practical sense, it is also inadequate for discrete time. Arithmetic operations on the time variable are a convenient and familiar way to express specifications concerning quantities of time. To be limited to a successor operator is too constraining.

Concurrency

A computation is sometimes modeled as a sequence of states, or state transitions. The index (or position) of a state (or transition) in a sequence is an abstraction of the time of its occurrence. In some models, an increasing index means increasing time; in others, it means nondecreasing time. A sequence of computations is easily composed into a single computation just by catenation. But composition of parallel computations is not so obvious.

Suppose computation is a sequence of actions, and an increasing index represents nondecreasing time. Then two adjacent actions in the sequence may perhaps occur at the same time. So it seems we have a possibility to represent concurrency. But how do we distinguish concurrent actions from sequential adjacent actions? An answer that has often been given is the following: if a specification allows two actions to occur adjacently in either order, then they are concurrent. This answer has been well criticised for confusing nondeterminacy (disjunction) with concurrency. Saying that two actions occur sequentially in either order is not the same as saying they occur concurrently.

This abstraction of time, as an index in a sequence, leads to another well-known problem. If parallel processes are represented as an interleaving of actions, we stretch time; a longer sequence is needed to represent the same time. Perhaps we do not mind, but many people have been concerned to say that time must not be stretched too far: a finite time (for one process) must not require an infinite sequence. This is the issue of fairness, and it suffers the same criticisms as a total correctness formalism.

Concurrency is basically conjunction. To say that P and Q are concurrent is to say that P and Q both describe the computation. In [Hoare81], parallelism is distinguished from communication between processes; the parallelism is disjoint, and it is exactly conjunction. When memory is shared, there is a problem: conjunction is not always implementable, even when both conjuncts

are. For example, $x := 2 \parallel x := 3$ asks for two, contradictory actions at the same time (it does not ask for two actions sequentially in either order). We may dismiss this example, saying that anyone who asks for the impossible should not expect to get it. But here is a less easily dismissed example: $x := x+1 \parallel y := y+1$. The left process says not only that x is increased, but also that y is unchanged. The right process says that y is increased and x is unchanged. Again, they contradict each other. What we want, of course, is that x and y are increased, and all other variables are unchanged.

In a semantics that does not measure time, and hides intermediate states, such as the partial correctness and total correctness semantics shown earlier, there is another problem. In such a semantics,

$$x := x+1;\ x := x-1 \ = \ ok$$

And so, with no escape,

$$(x := x+1;\ x := x-1) \ \parallel \ y := x$$
$$= \quad ok \parallel y := x$$

According to the first line, it may seem that $y' = x+1$ is a possibility: the right process $y := x$ may be executed in between the two assignments $x := x+1$ and $x := x-1$ in the left process. According to the last line, this does not happen; the final value of y is the initial value of x . No process may see or affect the intermediate states of another process. Intermediate state arises in the definition of sequential execution; it is the means by which information is passed from one program to a sequentially later program. It was not invented for passing information between parallel processes, and cannot be used for that purpose.

Useful concurrency is still possible in a semantics that does not measure time and hides intermediate states. Information can be passed between processes by communication primitives designed for the purpose. For one such definition, see [Hehner93].

For parallel processes to co-operate through shared memory, they must make their intermediate states visible to each other, and make their times explicit. The semantics in the section on temporal logic does exactly that. We need two more auxiliary ideas. First, we define *wait* as an easily implemented specification whose execution takes an arbitrary amount of time, and leaves all other variables unchanged during that time.

$$wait \ = \ t' \geq t \ \wedge \ \square \ xt = xt' \wedge yt = yt' \wedge \ldots$$

As in the definition of assignment, we must know what the state variables are in order to write the right side of this equation. We have been assuming throughout this paper that we always know what our state variables are in any specification. To make it explicit, we can adopt a notation similar to that used in [Morgan90]. Let $\alpha :: P$ be specification P with state variables α and t . For example,

$$x,y :: (x := y+z) \ = \ t' = t+1 \ \wedge \ x\,t' = (y+z)\,t \ \wedge \ y\,t' = y\,t$$

Here, x, y and t are the state variables; z is an ordinary variable, or parameter of the specification. That was the second auxiliary idea. Now we define parallel composition as follows.

$$\alpha::P \parallel \beta::Q \;=\; \alpha::P \wedge \beta::(Q; \mathit{wait}) \;\vee\; \alpha::(P; \mathit{wait}) \wedge \beta::Q$$

If P and Q are programs, we do not need to state α and β because they can be determined syntactically from the variables that appear on the left of assignments. Of course, to make use of $\parallel$ for programming, we must define it for more than just program operands.

Here is a simple example in variables x and y just to see that it works.

$$\begin{aligned}
&(x{:=}\,2;\ x{:=}\,x{+}y;\ x{:=}\,x{+}y) \parallel (y{:=}\,3;\ y{:=}\,x{+}y)\\
=\ &\quad (t' = t{+}1 \wedge xt'{=}2;\ t'{=}t{+}1 \wedge xt' = xt{+}yt;\ t'{=}t{+}1 \wedge xt' = xt{+}yt)\\
&\wedge (t' = t{+}1 \wedge yt' = 3;\ t' = t{+}1 \wedge yt' = xt{+}yt;\ t'{\geq}t \wedge \boxdot\, yt'{=}yt)\\
&\vee\ (t' = t{+}1 \wedge xt'{=}2;\ t'{=}t{+}1 \wedge xt' = xt{+}yt;\ t'{=}t{+}1 \wedge xt' = xt{+}yt;\ t'{\geq}t \wedge \boxdot\, xt'{=}xt)\\
&\wedge (t' = t{+}1 \wedge yt' = 3;\ t' = t{+}1 \wedge yt' = xt{+}yt)\\
=\ &\quad t' = t{+}3 \wedge x(t{+}1){=}2 \wedge x(t{+}2) = x(t{+}1){+}y(t{+}1) \wedge x(t{+}3) = x(t{+}2){+}y(t{+}2)\\
&\wedge t' \geq t{+}2 \wedge y(t{+}1){=}3 \wedge y(t{+}2) = x(t{+}1){+}y(t{+}1) \wedge (\boxdot\, yt' = y(t{+}2))\\
&\vee\ t' \geq t{+}3 \wedge (\text{other conjuncts})\\
&\wedge t' = t{+}2 \wedge (\text{other conjuncts})\\
=\ &\ t'{=}t{+}3 \wedge x(t{+}1){=}2 \wedge y(t{+}1){=}3 \wedge x(t{+}2){=}5 \wedge y(t{+}2){=}5 \wedge x(t{+}3){=}10 \wedge y(t{+}3){=}5
\end{aligned}$$

In that example, for ease of calculation, we made the unrealistic assumption that every assignment takes exactly one time unit. To be more realistic, we could suppose that assignment time depends on the operators within the assignment's expression, and possibly on the values of the operands. We could also allow the time to be nondeterministic, perhaps with lower and upper bounds, by writing $a \leq t'{-}t < b$. Whatever timing policy we decide on, whether deterministic or nondeterministic, whether discrete or continuous, the definitions and theory remain unchanged. Of course, complicated timing leads quickly to very complicated semantic expressions that describe all possible interactions. If we want to know only something, not everything, about the possible behaviors, we can proceed by implications instead of equations, weakening for the purpose of simplifying. Programming goes the other way: we start with a specification of desired behavior, and strengthen as necessary to obtain a program.

Here are some useful laws that can be proven from the definition of concurrent composition just given. Let b be a boolean expression and let P, Q, R, and S be specifications. Then

$P \parallel Q = Q \parallel P$	symmetry
$P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$	associativity
$P \parallel \mathit{ok} = \mathit{ok} \parallel P = P$	identity
$P \parallel Q \vee R = (P \parallel Q) \vee (P \parallel R)$	distributivity
$P \parallel \textbf{if } b \textbf{ then } Q \textbf{ else } R = \textbf{if } b \textbf{ then } (P \parallel Q) \textbf{ else } (P \parallel R)$	distributivity
$\textbf{if } b \textbf{ then } (P \parallel Q) \textbf{ else } (R \parallel S) = \textbf{if } b \textbf{ then } P \textbf{ else } R \parallel \textbf{if } b \textbf{ then } Q \textbf{ else } S$	distributivity

a Caution concerning Synchronization

In FORTRAN (prior to 1977) we could have a sequential composition of **if**-statements, but we could not have an **if**-statement containing a sequential composition. In ALGOL the syntax was fully recursive; sequential and conditional compositions could be nested, each within the other. Did we learn a lesson? Apparently we did not learn a very general one: we now seem happy to have a parallel composition of sequential compositions, but very reluctant to have a sequential composition of parallel compositions.

Suppose, for example, that we decide to have two processes, as follows.

$$(x := x+y;\ x := x\times y)$$
$$\parallel\quad (y := x-y;\ y := x/y)$$

The first modifies x twice, and the second modifies y twice. Suppose we want to synchronize the two processes at their mid-points, between the two assignments, forcing the faster process to wait for the slower one, and then to allow the two processes to continue with the new, updated values of x and y. The usual solution is to invent synchronization primitives to control the rate of execution of processes. But synchronization is sequencing, and we already have an adequate sequencing primitive. The solution should be

$$(x := x+y \parallel y := x-y);\ (x := x\times y \parallel y := x/y)$$

We just allow a sequential composition of parallel compositions.

an Aside concerning Specification

The specifications in this paper are boolean expressions. Traditionally, the presence or possibility of quantifiers turns a boolean expression into a predicate; in this paper, a predicate is a function with boolean range, and the quantifiers $\forall$ and $\exists$ apply to predicates to produce booleans. Thus $\forall x: int\cdot x\le y$ is a boolean expression.

We could have used predicate expressions rather than boolean expressions for specifications; the difference is language level. A predicate expression takes arguments by position, or address, whereas a boolean expression is supplied values for variables by their names. For example, in the predicate expression $\lambda x, y: int\cdot x\le y$, the names x and y are local (bound) and of no global significance (not free). We can supply 3 as first argument and 5 as second argument, as follows:

$$(\lambda x, y: int\cdot x\le y)\ 3\ 5$$

In the boolean expression $x\le y$, the names x and y are global (free), and we can supply value 3 for x and 5 for y as follows:

$$x := 3;\ y := 5;\ x\le y$$

according to the Substitution Law. For the convenience of using variables' names rather than

addresses, we have used boolean expressions rather than predicates for specifications.

In the well-known theory called Hoare Logic, a specification is a pair of boolean expressions. We specify that variable x is to be increased as follows:

$$\{x = X\}\ S\ \{x > X\}$$

(The parentheses were originally around S, but no matter.) In Dijkstra's theory of weakest preconditions, it is similar:

$$x{=}X \Rightarrow wp\ S\ (x{>}X)$$

There are two problems with these notations: X and S. They do not provide any way of relating the prestate and the poststate, hence the introduction of X. We ought to write $\forall X$, making X local, but customarily the quantifier is omitted. This problem is solved in the Vienna Development Method, in which the same specification is

$$\{true\}\ S\ \{x' > x\}$$

The other problem is that the programming language and specification language are disjoint, hence the introduction of S. Again, S should be local, but the appropriate quantifier is not obvious, and it is customarily omitted. In [Hehner84, Hoare92, Hehner93], the programming language is a sublanguage of the specification language. The specification that x is to be increased is

$$x' > x$$

The same single-expression double-state specifications are used in Z, but refinement is rather complicated. In Z, S is refined by P if and only if

$$\forall\sigma\cdot (\exists\sigma'\cdot S) \Rightarrow (\exists\sigma'\cdot P) \wedge (\forall\sigma'\cdot S \Leftarrow P)$$

where σ is the state variables. In Hoare Logic, $\{P\}\ S\ \{Q\}$ is refined by $\{R\}\ S\ \{U\}$ if and only if

$$\forall\sigma\cdot P \ \Rightarrow\ R \wedge (Q \Leftarrow U)$$

In this paper, S is refined by P if and only if

$$\forall\sigma, \sigma'\cdot S \Leftarrow P$$

Since refinement is what we must prove when programming, it is best to make refinement as simple as possible.

One might suppose that any type of mathematical expression can be used as a specification: whatever works. A specification of something, whether cars or computations, distinguishes those things that satisfy it from those that don't. Observation of something provides values for certain variables, and on the basis of those values we must be able to determine whether the something satisfies the specification. Thus we have a specification, some values for variables, and two possible outcomes. That is exactly the job of a boolean expression: a specification (of anything) really is a boolean expression. If instead we use a pair of predicates, or a function from predicates to predicates, or anything else, we make our specifications in an indirect way, and we make the task of determining satisfaction more difficult.

One might suppose that any boolean expression can be used to specify any computer behavior: whatever correspondence works. In Z, the expression *true* is used to specify (describe) terminating computations, and *false* is used to specify (describe) nonterminating computations. The reasoning is something like this: *false* is the specification for which there is no satisfactory final state; an infinite computation is behavior for which there is no final state; hence *false* represents infinite computation. Although we cannot observe a "final" state of an infinite computation, we can observe, simply by waiting 10 time units, that it satisfies $t' \geq t+10$, and it does not satisfy $t' < t+10$. Thus it ought to satisfy any specification implied by $t' \geq t+10$, including *true*, and it ought not to satisfy any specification that implies $t' < t+10$, including *false*. Since *false* is not true of anything, it does not (truly) describe anything. A specification is a description, and *false* is not satisfiable, not even by nonterminating computations. Since *true* is true of everything, it (truly) describes everything, even nonterminating computations. To say that P refines Q is to say that all behavior satisfying P also satisfies Q, which is just implication. The correspondence between specifications and computer behavior is not arbitrary.

Conclusions

The most striking conclusion of this paper is that a total correctness semantics is not worth its trouble. It is a considerable complication over a partial correcness semantics in order to gain one bit of information of dubious value (since nontermination cannot be observed, a promise of termination without a time bound is worthless). Partial correctness, with a time variable, provides more information at less cost. (The pejorative term "partial correctness" should not be used, and was not used in [Hoare69].)

Another contribution of this paper is a new semantics for Interval Temporal Logic, based on a boolean semantics of visible intermediate states, and using extensions. This semantics allows arbitrary arithmetic on a time variable, using the temporal operators as convenient quantifications.

And finally, a compositional semantics of concurrency with shared variables is presented.

Acknowledgments

The first usable theory of programming is due to C.A.R.Hoare. That work is so important that no-one ever needs to look in the references to see what paper [Hoare69] might be, and all work on the subject since then builds on that work. The kind of semantics used in the present paper, a single boolean expression, was developed in discussion with Tony Hoare during a fruitful term I spent at Oxford in 1981, and presented in [Hehner&Hoare83]. For those and many other inspirations, I thank Tony Hoare. I also thank Theo Norvell, Andrew Malton, and IFIP Working Group 2.3 for many good ideas and helpful discussion.

References

E.C.R.Hehner, C.A.R.Hoare: "a More Complete Model of Communicating Processes", *Theoretical Computer Science* v26 p105-120, 1983

E.C.R.Hehner: "Predicative Programming", *CACM* v27 n2 p134-151, 1984 February. See p142.

E.C.R.Hehner: *a Practical Theory of Programming*, Springer, New York, 1993

C.A.R.Hoare: "an Axiomatic Basis for Computer Programming", *CACM* v12 n10 p567-580, 583, 1969 October

C.A.R.Hoare: "a Calculus of Total Correctness of Communicating Processes", *the Science of Computer Programming* v1 n1-2 p49-72, 1981 October, and in C.A.R.Hoare, C.B.Jones: *Essays in Computing Science*, p289-314, Prentice-Hall International Series in Computer Science, London, 1989

C.A.R.Hoare: "Programs are Predicates", in C.A.R.Hoare, J.C.Shepherdson: *Mathematical Logic and Programming Languages*, p141-154, Prentice-Hall International, London, 1985, and in C.A.R.Hoare, C.B.Jones: *Essays in Computing Science*, p333-349, Prentice-Hall International, London, 1989

C.A.R.Hoare: lectures given at the NATO Advanced Study Institute, International Summer School on Program Design Calculi, Marktobedorf, 1992 July 28-August 9

C.C.Morgan: *Programming from Specifications*, Prentice-Hall International, London, 1990

B.C.Moszkowski: *Executing Temporal Logic Programs*, Cambridge University Press, 1986

X.Nicolin, J.Richier, J.Sifakis, J.Voiron: "ATP: an Algebra for Timed Processes", *Programming Concepts and Methods*, p414-443, Galilee, 1990