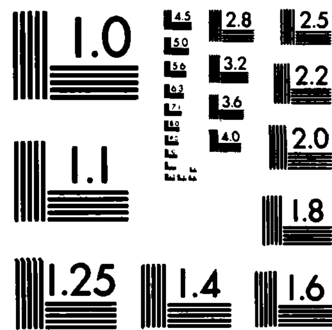
AD-A134 062 REASONING IN INTERVAL TEMPORAL LOGIC(U) STANFORD UNIV CA DEPT OF COMPUTER SCIENCE B MOSZKOWSKI ET AL. JUL 83 STAN-CS-83-969 N00039-82-C-0250 1/1

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

July 1983

Report No. STAN-CS-83-969

AD-A134062

Reasoning in Interval Temporal Logic

by

Ben Moszkowski and Zohar Manna

Department of Computer Science

Stanford University
Stanford, CA 94305

DTIC
OCT 26 1983
A

DTIC FILE COPY

83 09 07 141

Reasoning in Interval Temporal Logic

by

Ben Moszkowski[1,2] and Zohar Manna[1,3]

[1]Department of Computer Science, Stanford University, Stanford, CA 94305, USA
[2]From July, 1983: Computer Lab., Corn Exchange St., Cambridge Univ., England
[3]Applied Mathematics Deparment, Weizmann Institute of Science, Rehovot, Israel

Abstract

Predicate logic is a powerful and general descriptive formalism with a long history of development. However, since the logic's underlying semantics have no notion of time, statements such as "*I increases by 2*" cannot be directly expressed. We discuss *interval temporal logic* (ITL), a formalism that augments standard predicate logic with operators for time-dependent concepts. Our earlier work used ITL to specify and reason about hardware. In this paper we show how ITL can also directly capture various control structures found in conventional programming languages. Constructs are given for treating assignment, iteration, sequential and parallel computations and scoping. The techniques used permit specification and reasoning about such algorithms as concurrent Quicksort. We compare ITL with the logic-based programming languages Lucid and Prolog.

This work was supported in part by the National Science Foundation under a Graduate Fellowship, Grants MCS79-09495, MCS80-06930 and MCS81-11586, by DARPA under Contract N00039-82-C-0250, and by the United States Air Force Office of Scientific Research under Grant AFOSR-81-0014.

This paper will appear in the Proceedings of the ACM/NSF/ONR Workshop on Logics of Programs, June, 1983.

§1 Introduction

As a tool for specification and reasoning, predicate logic has many attractive features. Here are a few:

- Every formula and expression has a simple semantic interpretation
- Concepts such as recursion can be characterized and explored.
- Subsets can be used for programming (e.g., Prolog [5]).
- Theorems about formulas and expressions can themselves be stated and proved within the framework of predicate logic.
- Reasoning in predicate logic can often be reduced to propositional logic.
- Decades of research lie behind the overall formalism.

However, predicate logic has no built-in notion of time and therefore cannot directly express such dynamic actions as

"*I increases by* 2"

or

"*The values of A and B are exchanged.*"

We get around this limitation by using an extension of linear-time temporal logic [6,10] called *interval temporal logic* (ITL). The behavior of programs and hardware devices can often be decomposed into successively smaller periods or intervals of activity. These intervals provide a convenient framework for introducing quantitative timing details. State transitions can be characterized by properties relating the initial and final values of variables over intervals of time.

We originally used ITL to specify and reason about timing-dependent hardware. Moszkowski, Halpern and Manna [4,7,8] give details about ITL's syntax and semantics and also show how to describe hardware ranging from delay elements up to a clocked multiplier and an ALU bit slice. In this paper we show how ITL can also directly capture various control structures found in programming languages. Constructs are given for treating assignment, iteration, sequential and parallel computations and scoping.

§2 Expressing Programming Concepts in ITL

This section will show how ITL can express a variety of useful programming concepts. We assume that the reader is familiar with the syntax and semantics of first-order ITL as described by us in [4] and [8]. Upper-case variables such as A and I are signals and vary over states. Lower-case variables such as b and i are static and thus time-invariant. In general, variables such as A and b can range over all the elements of the underlying data domain. On the other hand, J and n range over natural numbers. The variable X always equals one of the truth values *true* and *false*.

Assignment

The assignment $A \to B$ is true for an interval if the signal B ends up with the signal A's initial value. If desired, we can reverse the direction of the arrow:

$$B \leftarrow A \quad \equiv_{\text{def}} \quad A \to B.$$

Assignment in ITL only affects variables explicitly mentioned; the values of other variables do not necessarily remain fixed. For example, the formulas

$$I \leftarrow (I + 2)$$

and

$$[I \leftarrow (I + 2)] \land [J \leftarrow J]$$

are not equivalent.

Example (*Leaving the elements of a vector unchanged*):

A vector U ends up unchanged iff all of its elements end up unchanged:

$$\models \quad (U \leftarrow U) \equiv \forall\, 0 \le i < |U|.\, (U[i] \leftarrow U[i]).$$

For example, if U has 3 elements, the following formula leaves U unchanged:

$$(U[0] \leftarrow U[0]) \land (U[1] \leftarrow U[1]) \land (U[2] \leftarrow U[2]).$$

This illustrates a simple form of parallel processing.

Example (*Swapping two variables*):

We define the predicate $A \leftrightarrow B$ to be true iff the values of the parameters A and B are swapped:

$$A \leftrightarrow B \quad \equiv_{\text{def}} \quad [(A \leftarrow B) \land (B \leftarrow A)]$$

A variable is swapped with itself iff it ends up unchanged:

$$\models \quad (A \leftrightarrow A) \equiv (A \leftarrow A).$$

If U and V are both vectors of length n then U and V are swapped iff their corresponding elements are swapped:

$$\models \quad (U \leftrightarrow V) \equiv \forall\, 0 \le i < n.\, (U[i] \leftrightarrow V[i]).$$

Example (In-place computation of maximum of two numbers):

Let the function $max(i, j)$ equal the maximum of the two values i and j. We can define $max(i, j)$ by means of a condition expression:

$$max(i, j) \quad =_{\text{def}} \quad \textit{if } i \geq j \textit{ then } i \textit{ else } j.$$

The following two ITL formulas are then semantically equivalent:

$$I \leftarrow max(I, J)$$

$$\textit{if } I \geq J \textit{ then } (I \leftarrow I) \textit{ else } (I \leftarrow J)$$

Example (In-place sorting):

Suppose we have a function $sort(U)$ that given a list U equals U in sorted order. The following predicate $Sort(U)$ then expresses that U is sorted in place:

$$Sort(U) \quad \equiv_{\text{def}} \quad U \leftarrow sort(U).$$

This is not the only way to express *Sort*. Let $bagval(U)$ be a function that gives the bag (multi-set) containing exactly the elements of U and let the predicate $sorted(U)$ be true of U iff the U's elements are sorted. An alternative way to express *Sort* can then be given by the following property:

$$\models \quad Sort(U) \equiv \big([bagval(U) \leftarrow bagval(U)] \wedge fin[sorted(U)]\big).$$

This says that a list is sorted in place iff the list's elements remain unchanged but end up in order.

Example (In-place parallel doubling of the values of a vector's elements):

Here is a formula that represents the parallel doubling of the elements of a vector U:

$$Double(U) \quad \equiv_{\text{def}} \quad \forall\, 0 \leq i < |U|.\, (U[i] \leftarrow 2U[i]).$$

The property below expresses *Double* recursively:

$$\models \quad Double(U) \quad \equiv \quad \textit{if } |U| > 0 \textit{ then}\big[(head(U) \leftarrow 2head(U)) \wedge Double(tail(U))\big].$$

Here the function $head(U)$ always equals U's leftmost element and $tail(U)$ equals the remainder of U:

$$head(U) =_{\text{def}} U[0] \qquad tail(U) =_{\text{def}} U[1 \text{ to } |U| - 1].$$

For example,

$$head(\langle 1, 3, 0\rangle) = 1, \quad tail(\langle 1, 3, 0\rangle) = \langle 3, 0\rangle.$$

The behavior of *head* and *tail* on the empty list $\langle\rangle$ is left unspecified. The logical construct *if* w_1 *then* w_2 is the same as the implication $w_1 \supset w_2$.

Example (*In-place parallel reversal of vector elements*):

The predicate $Rev(U)$ specifies an in-place reversal of the vector U's elements:

$$Rev(U) \quad \equiv_{\text{def}} \quad \forall\, 0 \le i < n.(U[i] \leftarrow U[n-i-1]),$$

where $n = |U|$. If the function $reverse(U)$ equals the reverse of the vector U, we can describe Rev as shown below:

$$\models \quad Rev(U) \equiv [U \leftarrow reverse(U)]$$

The following property expresses $Rev(U)$ by symmetrically exchanging pairs of U's elements in parallel:

$$\models \quad Rev(U) \equiv \forall\, 0 \le i < \lceil n \div 2 \rceil.\, (U[i] \leftrightarrow U[n-i-1])$$

For example, if the vector U has 5 elements, then $Rev(U)$ is equivalent to the formula

$$(U[0] \leftrightarrow U[4]) \;\wedge\; (U[1] \leftrightarrow U[3]) \;\wedge\; (U[2] \leftrightarrow U[2]).$$

The *gets* Construct

We now introduce the *gets* construct, which is used for repeatedly assigning one expression to another:

$$A \; gets \; B \quad \equiv_{\text{def}} \quad keep(B = \bigcirc A),$$

where the operator *keep* is defined as

$$keep\, w \quad \equiv_{\text{def}} \quad \boxdot(\neg empty \supset w).$$

The effect of *gets* is that B's current value always equals A's next value. The *keep* construct ensures that we don't "run off" the end of the interval. Note that *gets* is semantically equivalent to the unit delay

$$B \; del \; A$$

described by us in [4,7,8]. It is also similar to the operator "*followed by*" in the programming language Lucid [2].

Example (*Synchronous incrementing of a variable*):

The predicate $gen^n\, I$ initializes I to 0 and then repeatedly increments I by 1 until I equals n:

$$gen^n\, I \quad \equiv_{\text{def}} \quad beg(I = 0) \;\wedge\; (I \; gets \; I + 1) \;\wedge\; halt(I = n).$$

Example (*Strength reduction*):

The next formula has J run through $0^2, 1^2, \ldots, n^2$.

$$gen^n\, I \ \wedge\ J \approx I^2.$$

Rather that compute I^2 at each step, we can initialize J to 0 and continuously add $2I+1$ to it:

$$gen^n\, I \ \wedge\ beg(J=0) \ \wedge\ J\ gets\ (J+2I+1).$$

This is semantically equivalent to the previous formula but uses simpler operators. These formulas illustrate a way of treating *strength reduction* in ITL. The expression $2I+1$ can itself be strength-reduced if desired.

Example (*Assigning a list the sequence* $\langle 0, \ldots, 2n-1\rangle$):

The combined formula

$$(gen^{2n}\, I) \ \wedge\ (L = \langle\,\rangle) \ \wedge\ (L\ gets\ [L \,\|\, \langle I\rangle])$$

has the list variable L end up equal to the value $\langle 0, \ldots, 2n-1\rangle$. We use the operator $\|$ to append two lists together. Here is an example:

$$\langle 1,2\rangle \,\|\, \langle 5,0\rangle \quad = \quad \langle 1,2,5,0\rangle.$$

Example (*Simple pipeline*):

If U is a numerical vector of length $m+1$, the following formula sends twice the value of each element of U to the next:

$$\forall 0 \le i < m.\, (U[i+1]\ gets\ 2U[i]).$$

For example, if m equals 3, this is equivalent to the formula

$$(U[1]\ gets\ 2U[0]) \ \wedge\ (U[2]\ gets\ 2U[1]) \ \wedge\ (U[3]\ gets\ 2U[2]).$$

This illustrates a way of expressing highly parallel pipelines in ITL. By using the quantifier $\forall$, we can obtain an arbitrary number of simultaneously executing processes.

Example (*Keeping a variable stable*):

The predicate $stb\, A$ is true if the variable A has a fixed value throughout the entire interval:

$$stb\, A \quad \equiv_{\mathrm{def}} \quad \exists b.\, (A \approx b).$$

We can achieve stb by means of $gets$:

$$\models \quad stb\, A \equiv (A\ gets\ A).$$

Example (*Greatest common divisor*):

The predicate *GetsGcd* computes the greatest common divisor of two numbers:

$$GetsGcd(M,N) \quad \equiv_{\mathrm{def}} \quad M \; gets \; (N \bmod M) \;\wedge\; N \; gets \; M \;\wedge\; halt(M=0),$$

where the construct $halt\, w$ is true for intervals that terminate the first time the formula w is true:

$$halt\, w \quad \equiv_{\mathrm{def}} \quad \boxdot(w \equiv empty).$$

Thus $halt\, w$ can be thought of as a kind of wait-statement. Here is a corresponding correctness property for *GetsGcd*:

$$\models \quad GetsGcd(M,N) \supset [N \leftarrow gcd(M,N)],$$

where the function $gcd(i,j)$ equals the greatest common divisor of i and j. The following property shows that throughout the computation, M and N's gcd remains stable:

$$\models \quad GetsGcd(M,N) \supset stb[gcd(M,N)].$$

Measuring the length of an interval

We can view the formula

$$len = e$$

as an abbreviation for

$$\exists I.\big[beg(I=e) \;\wedge\; (I \; gets \; [I-1]) \;\wedge\; halt(I=0)\big].$$

The formula is true exactly of intervals with length e and illustrates how to localize or "hide" a variable such as I by means of existential quantification. This is similar to a *begin-block* in conventional block-structured programming languages. No conflicts arise when such a formula is combined with others containing variables named I. We use this technique elsewhere in this work.

Example (*Constraining the length of a computation*):

By using the construct *len*, we can look at the length of computations. For example, the formula

$$(I \leftarrow I^2) \;\wedge\; (len \le I)$$

specifies that I is squared in at most I steps.

Iteration

An interval can be broken up into an arbitrary number of successive subintervals, each satisfying some formula w. For example, we use the construct w^3 as an abbreviation for

$$w; w; w$$

We can extend ITL to include formulas of the form w^*; this is the Kleene closure of *semicolon*. Other constructs such as while-loops are also expressible within ITL:

$$\textit{while}\, w_1 \,\textit{do}\, w_2 \quad \equiv_{\text{def}} \quad [(\textit{beg}[w_1] \wedge w_2)^* \wedge \textit{fin}(\neg w_1)]$$

ITL can also be augmented with iteration of the form w^e where w is a formula and e is an arithmetic expression. This repeats w for e times in succession.

For-loops are expressible by means of while-loops. For example, the construct

$$\textit{for}\ 0 \leq I < n\ \textit{do}\,(J \leftarrow J + I)$$

can be expanded to

$$\textit{beg}(I = 0) \wedge \textit{while}\,(I < n)\ \textit{do}\,([J \leftarrow J + I] \wedge [I \leftarrow I + 1])$$

Example (*Sequential doubling of the values of a vector's elements*):

The following formula achieves the predicate $\textit{Double}(U)$ by sequentially running through the elements of the vector U and doubling each:

$$\models \quad [\textit{for}\ 0 \leq K < |U|\ \textit{do}\,\textit{Alter}(U, K, 2U[K])] \supset \textit{Double}(U)$$

The predicate $\textit{Alter}(U, i, a)$ sets the i-th element of U to the value a and leaves the other elements unchanged. We can define *Alter* in various ways. Here is one:

$$\textit{Alter}(U, i, a) \quad \equiv_{\text{def}} \quad \forall 0 \leq j < |U|. [\textit{if}\ i = j\ \textit{then}\,(U[j] \leftarrow a)\ \textit{else}\,(U[j] \leftarrow U[j])].$$

Sometimes a formal parameter of a predicate such as *Alter* has behavior that is slightly incompatible with that of the corresponding actual parameter. For example, the formula

$$\textit{Alter}(U, K, 2U[K])$$

contains the signals K and $2U[K]$ where static objects are expected. We therefore view the formula as an abbreviation for

$$\exists i, a. [\textit{beg}(i = K \wedge a = 2U[K]) \wedge \textit{Alter}(U, i, a)].$$

This form of *temporal conversion* corresponds to *call-by-value* in conventional programming languages.

Example (*In-place sequential reversal of a vector*):

The next formula reverses U by serially swapping pairs of U's elements:

$$for\, 0 \leq K < \lfloor |U| \div 2 \rfloor \; do\; Swap(U, K, |U| - K - 1),$$

where the predicate $Swap(U, i, j)$ exchanges the i-th and j-th elements of U, leaving the other elements unchanged. We can define *Swap* in a manner similar to the predicate *Alter* shown above. Note that sequential reversal provides one way to implement the parallel reversal computation discussed earlier.

Example (*Computation of greatest common divisor using while-loop*):

As mentioned previously, we can specify the in-place computation of the greatest common divisor of two variables M and N as follows:

$$N \leftarrow gcd(M, N).$$

The while-loop below implies this:

$$\begin{array}{l} while\,(M \neq 0)\; do \\ \quad if\,(M > N)\; then\,(M \leftrightarrow N)\; else\,([M \leftarrow M] \;\wedge\; [N \leftarrow N - M]). \end{array}$$

Example (*Expressing gets using a loop*):

The construct *gets* can be expressed using iteration:

$$\models \;\; (A\; gets\; B) \;\equiv\; (skip \;\wedge\; [A \leftarrow B])^*.$$

Based on the semantics of while-loops and the predicate *gets*, we can rewrite the while-loop

$$while\,(I \neq 0)\; do\,(skip \;\wedge\; [I \leftarrow I - 1] \;\wedge\; [J \leftarrow J + I])$$

as

$$halt(I = 0) \;\wedge\; (I\; gets\; I - 1) \;\wedge\; (J\; gets\; J + I).$$

This gives us a decentralized, concurrent view of the computation.

Example (*In-place partitioning of a vector*):

The predicate $Partition(U, I)$ specifies that the vector U of numbers is reorganized in place so that all elements in positions less than the variable I are less than $U[I]$ and the elements in higher positions are at least as large as $U[I]$:

$$Partition(U, I) \quad \equiv_{\text{def}} \quad ([bagval(U) \leftarrow bagval(U)] \;\wedge\; fin[partition(U, I)])$$

where the predicate $partition(u, i)$ is true iff u is partitioned about the i-th element:

$$partition(u,i) \quad \equiv_{\rm def} \quad \forall\, 0 \le j < |u|.\, [(j \ge i) \equiv (u[j] \ge u[i])].$$

The following property shows how to achieve *Partition* in an algorithmic manner:

$$\models \quad [if |U| > 0 \; then \; (Part(U,I); [Swap(U,0,I) \;\wedge\; (I \leftarrow I)])] \supset Partition(U,I).$$

We use *Part* to partition $tail(U)$ into elements $< U[0]$ and $\ge U[0]$:

$$Part(U,I) \quad \equiv_{\rm def} \quad \exists J.\, [beg(I = 1 \;\wedge\; J = |U| - 1) \;\wedge\; while\,(I \le J)\; do\; PartitionStep(U,I,J)].$$

Note that *Part* uses a localized variable J. Each iteration step of the while loop refers to $Partition(U, I, J)$, which either leaves U unchanged or swaps $U[I]$ with $U[J]$:

$$\begin{aligned} &PartitionStep(U,I,J) \quad \equiv_{\rm def} \\ &\quad if\,(U[I] \ge U[0]) \; then \; [(I \leftarrow I) \;\wedge\; (J \leftarrow J - 1) \;\wedge\; Swap(U,I,J)] \\ &\quad else\,[(I \leftarrow I + 1) \;\wedge\; (J \leftarrow J) \;\wedge\; (U \leftarrow U)]. \end{aligned}$$

Example (*Parallel Quicksorting of a vector*):

Using the predicate *Partition*, we can describe an in-place Quicksort algorithm that partitions a vector U and recursively sorts the resulting sections in parallel. The following property of in-place sorting is used.

$$\models \quad (if\; |U| > 0 \; then \; \exists I.\, [Partition(U,I); SortParts(U,I)]) \supset Sort(U),$$

where the predicate *SortParts* recursively sorts the two partitions of U in parallel:

$$SortParts(U,j) \quad \equiv_{\rm def} \quad Sort(U[0 \,{\rm to}\, j-1]) \;\wedge\; Sort(U[j+1 \,{\rm to}\, |U|-1]) \;\wedge\; (U[j] \leftarrow U[j]).$$

Here, for example, the expression $U[0 \,{\rm to}\, j - 1]$ equals the list

$$\langle U[0], \ldots, U[j-1] \rangle.$$

We leave the "pivot" element $U[j]$ unchanged. An actual implementation of this form of sorting might execute more sequentially.

§3 Markers

As mentioned earlier, we can iterate a temporal formula w by means of the construct

$$w^*.$$

A useful variant of this has an explicit Boolean flag X that is true exactly at the end-points of the individual iterative steps:

$$beg\,X \;\wedge\; ([\bigcirc\, halt\,X] \;\wedge\; w)^*.$$

Variables such as X are called *markers* since they mark off the loop's steps. We abbreviate the above form of looping by means of the operator *cycle*:

$$cycle_{w_1}\, w_2 \quad \equiv_{\mathrm{def}} \quad \left[beg\, w_1 \;\wedge\; ([\bigcirc\, halt\, w_1] \;\wedge\; w_2)\right].$$

Here the formula w_1 represents the marker and w_2 gives the individual iterative steps. From the semantics of ITL, every loop has an implicit marker. For example, the formulas

$$(I \leftarrow I+1)^* \qquad \text{and} \qquad \exists X.\; cycle_X(I \leftarrow I+1)^*$$

are semantically equivalent.

When a loop's marker is made explicit, we can sometimes express the loop as smaller mutually synchronized loops that operate in parallel. For example, the loop

$$cycle_X([I \leftarrow I-1] \;\wedge\; [J \leftarrow J+I])$$

can be represented as

$$cycle_X(I \leftarrow I-1) \;\wedge\; cycle_X(J \leftarrow J+I).$$

The individual steps of the loops start and end at the same times. This demonstrates one use of markers since, for instance, the loop

$$([I \leftarrow I-1] \;\wedge\; [J \leftarrow J+I])^*$$

is not readily decomposible without some additional means of synchronization.

If the marker X is identically *true*, then each step of the loop is reduced to having unit length. Thus, the construct

$$cycle_{true}(I \leftarrow I-1)$$

is equivalent to the formula

$$(skip \;\wedge\; [I \leftarrow I-1])^*$$

and therefore has the same meaning as

$$I\; gets\; (I-1).$$

Markers can also be used with while-loops. We define a while-loop with an explicit marker formula w_1 as follows:

$$while_{w_1}\, w_2\; do\; w_3 \quad \equiv_{\mathrm{def}} \quad beg\, w_1 \;\wedge\; while\, w_2\; do\,([\bigcirc\, halt\, w_1] \;\wedge\; w_3).$$

For instance, the loop

$$while_X\,(I \neq 0)\; do\,([I \leftarrow I-1] \;\wedge\; [J \leftarrow J+I])$$

can be alternatively expressed by means of the following conjunction of three formulas:

$$halt(I=0) \;\wedge\; cycle_X(I \leftarrow I-1) \;\wedge\; cycle_X(J \leftarrow J+I).$$

§4 Data Transmission

In ITL we can use shared variables for communication between different processes. Given an interval and some variable A, it is convenient to speak of the *trace* of A. We define the function $tr(A)$ to be the sequence of A's values in all but the last state of the interval:

$$tr(A) \quad =_{\mathrm{def}} \quad \langle (\bigcirc^j A) : 0 \le j < len \rangle.$$

Thus, in an interval of length 2, the value of $tr(A)$ is the sequence

$$\langle A, \bigcirc A \rangle.$$

Note that in an interval of length 0, $tr(A)$ equals the empty sequence $\langle \rangle$. A variant of $tr(A)$ that doesn't ignore that interval's last state can also be defined.

Example (Transmitting the elements of a list):

The formula $WriteList_I(L)$ outputs the contents of the list L from left to right into the variable I:

$$WriteList_I(L) \quad \equiv_{\mathrm{def}} \quad [tr(I) = L].$$

The next property shows a constructive way to achieve $WriteList$:

$$\models \quad (keep[I = head(L)] \wedge L\ gets\ [tail(L)] \wedge halt[L = \langle \rangle]) \supset WriteList_I(L).$$

The predicate $ReadList_I(L)$ is similar to $WriteList_I(L)$ but requires that L end up with I's trace:

$$ReadList_I(L) \quad \equiv_{\mathrm{def}} \quad [L \leftarrow tr(I)].$$

Example (Writing a set in sorted order):

The predicate $WriteSorted_I(S)$ outputs the elements of the finite set S in sorted order to the variable I:

$$WriteSorted_I(S) \quad \equiv_{\mathrm{def}} \quad WriteList_I(sort(S)).$$

For simplicity, we assume that S contains only numbers. The following formula gives a way to achieve $WriteSorted$:

$$keep(I = \min S) \wedge S\ gets\ (S \sim \{I\}) \wedge halt(S = \{\,\}).$$

The ITL *keep* construct ensures that the variable I always equals the minimum element of S except perhaps in the computation's last state. The *gets* subformula continually deletes I's value from S. As the computation runs, S is reduced to being empty. In the combined

formula

$$WriteSorted_I(S) \wedge ReadList_I(L),$$

the list L ends containing the initial elements of S in sorted order:

$$\models \quad [WriteSorted_I(S) \wedge ReadList_I(L)] \supset [L \leftarrow sort(S)].$$

For example, if S initially equals the set $\{3, 5, 1\}$ then upon termination, L equals $\langle 1, 3, 5\rangle$. Note that bags (multisets) can be used instead of sets if duplicate data values arise.

Example (*Synchronous walk through an S-expression*):

An *S-expression* is either an *atom* or a pair $\langle a, b\rangle$ where a and b are themselves S-expressions. For our purposes, we restrict atoms to being nonnegative integers. Here are some simple S-expressions:

$$3, \quad \langle 4, 1\rangle, \quad \langle\langle 2, 3\rangle, 5\rangle.$$

The predicate $atom(t)$ is true iff the S-expression t is an atom. If t is not atomic then $left(t)$ and $right(t)$ access t's two parts. We can inductively define the *frontier* of an S-expression as follows:

$$frontier(t) \quad \equiv_{\mathrm{def}} \quad \textit{if } atom(t) \textit{ then } \langle t\rangle \textit{ else } \big[frontier(left(t)) \,\|\, frontier(right(t))\big].$$

For instance, the frontiers of the S-expressions given above are respectively

$$\langle 3\rangle, \quad \langle 4, 1\rangle, \quad \langle 2, 3, 5\rangle.$$

The predicate *WriteFrontier* specifies that the frontier of the S-expression T is output to the variable I:

$$WriteFrontier_I(T) \quad \equiv_{\mathrm{def}} \quad WriteList_I(frontier(T)).$$

The following property shows how to recursively use *WriteFrontier* to output the frontier of a static S-expression t:

$$\begin{aligned} \models \quad & WriteFrontier_I(t) \equiv \\ & \quad \textit{if } atom(t) \textit{ then } [beg(I = t) \wedge skip] \\ & \quad \textit{else} \big[WriteFrontier_I(left(t));\ WriteFrontier_I(right(t))\big]. \end{aligned}$$

An S-expression T's frontier can for example be entered into a list L as shown by the property

$$\models \quad [WriteFrontier_I(T) \wedge ReadList_I(L)] \supset [L \leftarrow frontier(T)].$$

§5 Comparison with the Programming Languages Lucid and Prolog

The innovative programming language *Lucid* developed by Ashcroft and Wadge [1,2,3] is similar to parts of ITL. For example, the ITL formula

$$beg(I = 0 \wedge J = 0) \wedge I \; gets \; (I+1) \wedge J \; gets \; (J+I)$$

roughly corresponds to the Lucid program

$$\begin{aligned} I &= 0 \; fby \; (I+1) \\ J &= 0 \; fby \; (J+I). \end{aligned}$$

Not surprisingly, many properties of *gets* involving such concepts as strength reduction can also be handled in Lucid. On the other hand, the Algol-like ITL formula

$$while \, (I \neq 0) \; do \, ([I \leftarrow I - 1] \wedge [J \leftarrow J + I])$$

has no direct analog in Lucid. Lucid's underlying semantics are rather different from ITL's since Lucid uses a three-valued logic and has no notion of global state. Instead, each variable has an infinite sequence of values.

Prolog [5] is based on an interesting subset of predicate logic in which formulas can be interpreted as applicative programs. Because Prolog has no sense of time, ITL formulas cannot in general be directly expressed in it. For example, there is no true analog in Prolog to ITL's while-loops and assigments. In practice, side effects are permitted in Prolog, although the language's core is not really designed to handle them.

§6 Future Research Directions

Let use now consider some aspects of ITL that require further investigation.

Temporal types and higher-order objects

A theory of temporal types needs to be developed. This should provide various ways of constructing and comparing types. Given two predicates p and q, we form the predicate $p \times q$ which is true for any pair whose first element satisfies p and whose second element satisfies q. For example, the formula

$$(nat \times bool)(\langle 3, true \rangle)$$

is true. In general, we write such a test as

$$\langle 3, true \rangle : (nat \times bool).$$

The operator $\times$ extends to n-element tuples:

$$p_1 \times \cdots \times p_n,$$

where $p_1, \ldots, p_n$ are unary predicates. In addition, the construct p^n is equivalent to n repetitions of p. For instance, the test

$$a\colon nat^3$$

is true if a is a triple of natural numbers. The predicate p^* is true for vectors of arbitrary, possibly null length, whose elements all satisfy p. Thus, the type $bool^*$ is true for all vectors of truth values. The type $sig(bool^*)$ is true for any Boolean vector signal with a possibly varying length.

The predicate $struct(X_1\colon p_1, \ldots, X_n\colon p_n)$ checks for tuples whose elements have field names $X_1, \ldots, X_n$ and satisfy the respective types $p_1, \ldots, p_n$. For example, the predicate

$$struct(X\colon nat, Y\colon bool^2)$$

is true for the tuple

$$\langle X\colon 3, Y\colon \langle true, false\rangle\rangle.$$

Note that two types can be semantically equivalent. For example, the types $sig(bool^2)$ and $[sig(bool)]^2$ have the same meaning. On the other hand, the types $sig(bool^*)$ and $[sig(bool)]^*$ are not equivalent. The type $sig(bool^*)$ is true for any object that is always a Boolean vector signal with a possibly varying length. In contrast, the type $[sig(bool)]^*$ requires that the object's length be fixed over time:

$$\models \quad A\colon [sig(bool)]^* \;\equiv\; [A\colon sig(bool^*) \;\wedge\; stb|A|].$$

Type constructs of the form p^* have other uses. For example, we can define the predicate $incr$ to increment a variable I by 1:

$$incr(I) \quad =_{\text{def}} \quad (I \leftarrow I + 1).$$

Then given a vector U, the formula

$$U\colon incr^*$$

specifies that each element of U is incremented by 1 in parallel. This technique is similar to the *mapcar* function of Lisp.

It would be interesting to have a semantics of higher-order temporal objects such as time-dependent functionals. Perhaps a suitable variant of proposition ITL can facilitate some sort of Gödelization by representing all values as temporal formulas. Alternatively, an encoding like that used by Scott [11,12] in developing a model of the typeless lambda calculus might work. However, we wish to strongly resist the introduction of partial values. One concession we make in this direction is to not require that every function have a fixed point.

Projection

Sometimes it is desirable to examine a computation at certain points in time and ignore all intermediate states. This can be done using the *temporal projection* construct $w_1 \,\Pi\, w_2$. For example, the formula

$$X \,\Pi\, (I \text{ gets } [I+1])$$

is true if I increments by 1 over each successive pair of the states where X is true. Variables like X serve as markers for measuring time and facilitate different levels of atomicity. If two parts of a system are active at different times or are running at different rates, markers can be constructed to project away the asynchrony.

Other definitions of projection are also possible. For example, a *synchronous* form can be defined as follows:

$$w_1 \; sim \; w_2 \quad \equiv_{\text{def}} \quad (beg\, w_1 \;\wedge\; fin\, w_1 \;\wedge\; [w_1 \,\Pi\, w_2]).$$

This forces the marker formula w_1 to be true in an interval's initial and final states. We can view this construct as *simulating* the formula w_2 at at rate given by w_1; hence the name "*sim.*" For example, the formula

$$X \; sim \; [ReadList_I(L)]$$

reads the variable I into the list L at the rate indicated by X.

In section 3, we showed how to express iterative constructs by means of markers: For example, the following loop has X as a marker:

$$cycle_X([I \text{ gets } I+1] \;\wedge\; [J \text{ gets } J+I]).$$

All loops have implicit markers that are accessible through existential quantification. This provides a general means for identifying the end points of the iteration steps and extracting them using projection. We feel that markers and projection provide a way to decoupled low-level computational details from high-level ones.

Tempura, a prototype programming language based on ITL

Moszkowski and Manna [9] present a prototype programming language called *Tempura* that is based on ITL. Along with the programming languages Lucid and Prolog, Tempura has the property of having a semantics based on logic. Much work remains ahead in exploring this temporal approach to language design and developing practical techniques for specifying, executing, transforming, synthesizing and verifying Tempura programs. Perhaps the state sequences of temporal logic can also be used as a convenient basis for logics of, say, formal languages, typesetting and music. More generally, temporal logic may provide a semantics of both time and space.

§7 Conclusions

Interval temporal logic has constructs for dealing with such programming concepts as assignment, iteration and computation length. Because ITL is a logic, programs and properties can be stated in the same formalism. Unlike conventional first-order logic, ITL can directly express computations requiring a notion of change. Moszkowski, Halpern and Manna [4,7,8] have shown that ITL also provides a basis for describing timing-dependent hardware involving clocking and propagation delay. ITL-based programming languages such as Tempura [9] will be able to take advantage of this versatility. Thus, ITL appears to have a wide range of application.

Acknowledgements

We wish to thank Martin Abadi, Joe Halpern, John Hobby and Yoni Malachi for stimulating conversations and suggestions.

References

1. E. A. Ashcroft and W. W. Wadge. "Lucid: A formal system for writing and proving programs." *SIAM Journal of Computing 5*, 3 (Sept. 1976), 336–354.

2. E. A. Ashcroft and W. W. Wadge. "Lucid, a nonprocedural language with iteration." *Communications of the ACM 20*, 7 (July 1977), 519–526.

3. E. A. Ashcroft and W. W. Wadge. *Lucid, the Data Flow Programming Language*. To be published.

4. J. Halpern, Z. Manna and B. Moszkowski. A hardware semantics based on temporal intervals. Proceedings of the 10-th International Colloquium on Automata, Languages and Programming, Barcelona, Spain, July, 1983.

5. R. Kowalski. *Logic for Problem Solving*. Elsevier North Holland, Inc., New York, 1979.

6. Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, pages 215–273, Academic Press, New York, 1981.

7. B. Moszkowski. A temporal logic for multi-level reasoning about hardware. Proceedings of the 6-th International Symposium on Computer Hardware Description Languages, Pittsburgh, Pennsylvania, May, 1983, pages 79–90.

8. B. Moszkowski. *Reasoning about Digital Circuits*. PhD Thesis, Department of Computer Science, Stanford University, 1983.

9. B. Moszkowski and Z. Manna. Temporal logic as a programming language. In preparation.

10. N. Rescher and A. Urquart. *Temporal Logic*. Springer-Verlag, New York, 1971.

11. D. Scott. "Data types as lattices." *SIAM Journal of Computing 5*, 3 (Sept. 1976), 522–587.

12. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, Masachusetts, 1977.

END

FILMED

11-83

DTIC