Technical Report

UCAM-CL-TR-55
ISSN 1476-2986

Number 55

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Executing temporal logic programs

Ben Moszkowski

August 1984

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Executing Temporal Logic Programs[1]

(preliminary version)

Ben Moszkowski

Computer Laboratory, University of Cambridge,
Corn Exchange Street, Cambridge CB2 3QG, England

20 August 1984

Abstract

Over the last few years, temporal logic has been investigated as a tool for reasoning about computer programs, digital circuits and message-passing systems. In the case of programs, the general feeling has been that temporal logic is an adjunct to existing languages. For example, one might use temporal logic to specify and prove properties about a program written in, say, CSP. This leads to the annoyance of having to simultaneously use two separate notations.

In earlier work we proposed that temporal logic itself directly serve as the basis for a programming language. Since then we have implemented an interpreter for such a language called *Tempura*. We are developing Tempura as a tool for directly executing suitable temporal logic specifications of digital circuits and other discrete-time systems. Since every Tempura statement is also a temporal formula, we can use the entire temporal logic formalism for our assertion language and semantics. Tempura has the two seemingly contradictory properties of being a logic programming language and having imperative constructs such as assignment statements.

The presentation given here first describes the syntax and semantics of a first-order temporal logic having the operators ○ (*next*) and □ (*always*). This serves as the basis for the Tempura programming language. The lesser known temporal operator *chop* is subsequently introduced, resulting in *Interval Temporal Logic*. We then show how to incorporate *chop* and related constructs into Tempura.

[1]To appear in the *Proceedings of the NSF/SERC Workshop on the Semantics of Concurrency*, Carnegie-Mellon University, Pittsburgh, Pennsylvania, USA, July 9–11, 1984.

1. Introduction

Temporal logic [12,20] has been recently put forward as a useful tool for reasoning about concurrent programs and hardware. Within temporal logic, one can express logical operators for reasoning about time-dependent concepts such as "*always*" and "*sometimes.*" Consider, for example, the English sentence

"*If the propositions P and Q are always true, then P is always true.*"

This can be represented in temporal logic by the formula

$$\Box(P \wedge Q) \supset \Box P.$$

Here the operator $\Box$ corresponds to the notion "*always.*" Thus, the subformula $\Box(P \wedge Q)$ can be understood as "*P and Q are always true.*"

Typically, temporal logic has been thought of as a tool for specifying and proving properties of programs written in, say, CSP [11] or variants of Pascal with concurrency [10]. This distinction between temporal logic and programming languages has troubled us since it has meant that we must simultaneously use two separate notations. Programming formalisms such as Hoare logic [9], dynamic logic [6,19], and process logic [4,7] also reflect this dichotomy. One way to bridge the gap is to find ways of using temporal logic itself as a tool for programming and simulation. With this in mind, we have developed *Tempura*, an imperative programming language based on subsets of temporal logic. Every Tempura statement is a temporal logic formula. This lets us specify and reason about Tempura programs without the need for two notations.

Another aspect of the current usage of temporal logic is the restriction of temporal constructs to such propositional operators as $\Box$ (*always*), $\bigcirc$ (*next*), $\Diamond$ (*sometimes*) and $\mathcal{U}$ (*until*). In fact, there are quite a few other useful propositional and first-order temporal operators for treating such programming concepts as iteration, assignment and scoping. We explore these constructs within *Interval Temporal Logic*, a formalism having the operators $\Box$, $\bigcirc$ and the lesser known temporal operator *chop*. This also serves as the underlying notation for Tempura programs and their specifications and properties.

1.1. Organization of Paper

We start off by reviewing the syntax and semantics of a temporal logic having the operators $\Box$ (*always*) and $\bigcirc$ (*next*). A number of temporal constructs are then

presented and later used to build legal Tempura statements and expressions. This is followed by a description of an interpreter for executing such statements and a discussion of some of the trade-offs made in implementing the system. Subsequently, we extend the logic to include the temporal operator *chop*. Within the resulting Interval Temporal Logic we derive some Algol-like constructs which are subsequently incorporated into Tempura and the interpreter. We conclude with a look at some related work. The Tempura examples given here have been intentionally kept simple. However, in the full version of this paper we plan to discuss Tempura programs that have been implemented for such tasks as parallel quicksorting and simulation of a hardware multiplier.

2. Basic Features of the Temporal Logic

Before describing Tempura, it is necessary to have an understanding of the underlying temporal logic. Some of the constructs described here will be later used in Tempura programs. Others will facilitate reasoning about program behavior. Rather than presenting the entire logic at once, we will first introduce some basic operators and derive others from them. In a later section, some additional operators will be considered.

2.1. Syntax of the Logic

The initial set of constructs includes conventional logical operators such as $=$ (*equality*) and $\wedge$ (*logical-and*). In addition, there are the two temporal operators $\bigcirc$ (*next*) and $\Box$ (*always*).

2.1.1. Syntax of Expressions

Expressions are built inductively as follows:

- Individual variables: $A, B, C, \ldots$
- Functions: $f(e_1, \ldots, e_k)$, where $k \geq 0$ and $e_1, \ldots, e_k$ are expressions. In practice, we use functions such as $+$ and *mod*. Constants such as 0 and 1 are treated as zero-place functions.
- Next: $\bigcirc e$, where e is an expression.

Here are two examples of syntactically legal expressions:

$$I + (\bigcirc J) + 1, \qquad (\bigcirc I) + J - \bigcirc\bigcirc(I + \bigcirc J).$$

2.1.2. Syntax of Formulas

Formulas are built inductively as follows:

- Predicates: $p(e_1, \ldots, e_k)$, where $k \geq 0$ and $e_1, \ldots, e_k$ are expressions. Predicates include $\leq$ and other basic relations.
- Equality: $e_1{=}e_2$, where e_1 and e_2 are expressions.
- Logical connectives: $\neg w$ and $w_1 \wedge w_2$, where w, w_1 and w_2 are formulas.
- Next: $\bigcirc w$, where w is an formula.
- Always: $\Box w$, where w is an formula.

Here are some syntactically legal ITL formulas:

$$(J = 2) \wedge \bigcirc(I = 3), \quad \big(\bigcirc\Box[I = 3]\big) \wedge \neg\big([\bigcirc J] = 4\big), \quad \bigcirc\big(\Box[I = 3] \wedge \bigcirc\bigcirc[J = 4]\big).$$

Note that the operator $\bigcirc$ can be used both for expressions (e.g., $\bigcirc J$) and for formulas (e.g., $\bigcirc(I = 3)$).

2.2. Models

A model is a triple $(\mathcal{D}, \Sigma, \mathcal{M})$ containing a data domain $\mathcal{D}$, a set of states Σ and a interpretation $\mathcal{M}$ giving meaning to every function and predicate symbol. For the time being, we take the data domain $\mathcal{D}$ to be the integers. A state is a function mapping variables to values in $\mathcal{D}$. We let Σ be the set of all such functions. For a state $s \in \Sigma$ and a variable A, we let $s[\![A]\!]$ denote A's value in s. Each k-place function symbol f has an interpretation $\mathcal{M}[\![f]\!]$ which is a function mapping k elements in $\mathcal{D}$ to a single value:

$$\mathcal{M}[\![f]\!] \in (\mathcal{D}^k \rightarrow \mathcal{D}).$$

Interpretations of predicate symbols are similar but map to truth values:

$$\mathcal{M}[\![p]\!] \in (\mathcal{D}^k \rightarrow \{true, false\}).$$

We assume that $\mathcal{M}$ gives standard interpretations to operators such as $+$ and $<$.

The semantics given here keep the interpretations of function and predicate symbols independent of intervals. The semantics can however be extended to allow for functions and predicates that take into account the dynamic behavior of parameters.

Using the states in Σ, we construct *intervals* of time from the set Σ^+. An interval is thus any nonempty, finite sequence of states. If s, t and u are states $\in \Sigma$, then the following are possible intervals:

$$\langle s \rangle, \quad \langle sttsus \rangle, \quad \langle tttt \rangle.$$

Note that an interval always contains at least one state.

We now introduce some basic notation for manipulating intervals. Given an interval σ, we let $|\sigma|$ be the *length* of σ. Our convention is that an interval's length is the number of states *minus one*. Thus the intervals above have respective lengths 0, 5 and 3. The individual states of an interval σ are denoted by $\sigma_0, \sigma_1, \ldots, \sigma_{|\sigma|}$. For instance, the following equality is true iff the variable A has the value 5 in σ's final state:

$$\sigma_{|\sigma|}[\![A]\!] = 5.$$

The model described here views time as being discrete and is not intended to be a realistic representation of the world around us. Nonetheless, it provides a sound basis for reasoning about many interesting dynamic phenomena involving timing-dependent and functional behavior. Furthermore, a discrete-time view of the world often corresponds to our mental model of digital systems and computer programs. In any case, we can always make the granularity of time arbitrarily fine.

2.3. Interpretation of Expressions and Formulas

We now extend the interpretation $\mathcal{M}$ to give meaning to expressions and formulas on intervals. The construct $\mathcal{M}_\sigma[\![e]\!]$ will be defined to equal the value in $\mathcal{D}$ of the expression e on the interval σ. Similarly, $\mathcal{M}_\sigma[\![w]\!]$ will equal the truth value of the formula w on σ.

At first glance, the following definitions may seem somewhat arbitrary. We therefore suggest that an initial reading be rather cursory since the subsequent discussion and examples provide motivation. The definitions can then be referenced as needed.

- $\mathcal{M}_\sigma[\![v]\!] = \sigma_0[\![v]\!]$, where v is a variable.
  Thus, a variable's value on an interval equals the variable's value in the interval's initial state.

- $M_\sigma[\![f(e_1,\ldots,e_k)]\!] = M[\![f]\!](M_\sigma[\![e_1]\!],\ldots,M_\sigma[\![e_k]\!])$.
  The interpretation of the function symbol f is applied to the interpretations of $e_1,\ldots,e_k$.
- $M_\sigma[\![\bigcirc e]\!] = M_{\langle\sigma_1\ldots\sigma_{|\sigma|}\rangle}[\![e]\!]$, if $|\sigma| \geq 1$.
  We leave the value of $\bigcirc e$ unspecified on intervals having length 0.
- $M_\sigma[\![p(e_1,\ldots,e_k)]\!] = M[\![p]\!](M_\sigma[\![e_1]\!],\ldots,M_\sigma[\![e_k]\!])$.
- $M_\sigma[\![e_1{=}e_2]\!] = \mathit{true}$ iff $M_\sigma[\![e_1]\!] = M_\sigma[\![e_2]\!]$.
- $M_\sigma[\![\neg w]\!] = \mathit{true}$ iff $M_\sigma[\![w]\!] = \mathit{false}$.
- $M_\sigma[\![w_1 \wedge w_2]\!] = \mathit{true}$ iff $M_\sigma[\![w_1]\!] = \mathit{true}$ and $M_\sigma[\![w_2]\!] = \mathit{true}$.
- $M_\sigma[\![\bigcirc w]\!] = \mathit{true}$ iff $|\sigma| \geq 1$ and $M_{\langle\sigma_1\ldots\sigma_{|\sigma|}\rangle}[\![w]\!] = \mathit{true}$.
- $M_\sigma[\![\Box w]\!] = \mathit{true}$ iff for all $i \leq |\sigma|$, $M_{\langle\sigma_i\ldots\sigma_{|\sigma|}\rangle}[\![w]\!] = \mathit{true}$.

Examples

We will now illustrate the use of M by considering the semantics of the sample temporal formulas given earlier. Let s, t and u be states in which the variables I and J have the following values:

	I	J
s	1	2
t	3	4
u	3	1

The formula

$$(J = 2) \wedge \bigcirc(I = 3)$$

is true on an interval σ iff σ has length ≥ 1, the value of J in the state σ_0 is 2 and the value of I in the state σ_1 is 3. Thus, the formula is true on the interval $\langle stu\rangle$. On the other hand, the formula is false on the interval $\langle ttu\rangle$ because J's initial value on this interval is 4 instead of 2.

The formula

$$\big(\bigcirc\Box[I = 3]\big) \wedge \neg\big([\bigcirc J] = 4\big)$$

is true on any interval σ having length ≥ 1 and in which I equals 3 in the states $\sigma_1, \ldots, \sigma_{|\sigma|}$ and J does not equal 4 in σ_1. Thus the formula is true on the interval $\langle sutut\rangle$ but is false on $\langle t\rangle$ and $\langle stutu\rangle$.

The formula

$$\bigcirc\big(\Box[I = 3] \wedge \bigcirc\bigcirc[J = 4]\big)$$

is true on an interval σ having length ≥ 3 and in which the variable I equals 3 in the states $\sigma_1, \ldots, \sigma_{|\sigma|}$ and the variable J equals 4 in the state σ_3. Thus this formula is true of the interval $\langle suutu \rangle$ but is false on $\langle s \rangle$ and $\langle sutuu \rangle$.

2.4. Satisfiability and Validity

A formula w is *satisfied* by an interval σ iff the meaning of w on σ equals *true*:

$$\mathcal{M}_\sigma[\![w]\!] = true.$$

This is denoted as follows:

$$\sigma \models w.$$

If all intervals satisfy w then w is *valid*, written $\models w$.

Example (Validity):

The following formula is true on an interval σ iff $|\sigma| \geq 1$, the variable I always equals 1 and in the state σ_1, I equals 2:

$$\Box(I = 1) \wedge \bigcirc(I = 2).$$

No interval can have all of these characteristics. Therefore the formula is false on all intervals and its negation is always true and hence valid:

$$\models \quad \neg[\Box(I = 1) \wedge \bigcirc(I = 2)].$$

3. Deriving Other Operators

The kinds of interval behavior one can describe with the constructs so far introduced may seem rather limited. In fact, this is not at all the case since we can develop quite a variety of derived operators. We will now present some derived operators that have proved useful in reasoning about simple computations.

3.1. Boolean Operators

The conventional boolean constructs $w_1 \vee w_2$ (logical-or) , $w_1 \supset w_2$ (implication) and $w_1 \equiv w_2$ (equivalence) can be expressed in terms of $\neg$ and $\wedge$. We can define logical-or as shown below:

$$w_1 \vee w_2 \quad \equiv_{\text{def}} \quad \neg(\neg w_1 \wedge \neg w_2).$$

We then express implication and equivalence as follows:

$$w_1 \supset w_2 \quad \equiv_{\text{def}} \quad \neg w_1 \vee w_2, \qquad w_1 \equiv w_2 \quad \equiv_{\text{def}} \quad (w_1 \supset w_2) \wedge (w_2 \supset w_1).$$

The boolean constructs *true* and *false* can also be derived.

Example (Implication):

If in an interval σ, the variable I always equals 1 and in the state σ_1 the variable J equals 2 then it follows that the expression $I + J$ equals 3 in σ_1. This fact can be expressed by the following valid formula:

$$\models \quad \big[\Box(I = 1) \wedge \bigcirc(J = 2)\big] \supset \bigcirc(I + J = 3).$$

Example (Equivalence):

The formula

$$\bigcirc\big([I = 1] \wedge [J = 2]\big)$$

is true on an interval σ iff σ has length ≥ 1 and in the state σ_1, the variable I has the value 1 and the variable J has the value 2. It turns out that the conjunction

$$\bigcirc(I = 1) \wedge \bigcirc(J = 2)$$

has the same meaning. The equivalence of these two formulas is expressible as follows:

$$\bigcirc\big([I = 1] \wedge [J = 2]\big) \;\equiv\; \big[\bigcirc(I = 1) \wedge \bigcirc(J = 2)\big].$$

This formula is true on all intervals and is therefore valid. In general, if two formulas w_1 and w_2 have the same meaning on all intervals, then the equivalence $w_1 \equiv w_2$ is valid.

3.2. The Operator *empty*

The formula *empty* is true on an interval iff the interval has length 0:

$$\sigma \models \mathit{empty} \quad \text{iff} \quad |\sigma| = 0.$$

We can define *empty* as follows:

$$\mathit{empty} \quad \equiv_{\text{def}} \quad \neg \bigcirc \mathit{true}.$$

Example (Testing the length of an interval):

We can use the constructs ○ and *empty* to test the length of an interval. For example, the formula

$$\bigcirc\bigcirc\bigcirc\, empty$$

is true on an interval σ iff σ has length 3.

3.3. The Operators *gets* and *stable*

It is useful to say that over time one expression e_1 equals another expression e_2 but with a one-unit delay. We use the construct e_1 *gets* e_2 to represent this and define it as follows:

$$e_1\ gets\ e_2 \quad \equiv_{\text{def}} \quad \Box\big(\neg empty \supset [(\bigcirc e_1) = e_2]\big).$$

The test $\neg empty$ ensures that we do not "run off" the edge of the interval by erroneously attempting to examine e_1's value in the nonexistent state $\sigma_{|\sigma|+1}$.

For instance, the formula $K\ gets\ 2K$ is true on an interval σ iff the variable K is repeatedly doubled from each state to its successor:

$$\sigma \models \quad K\ gets\ 2K \quad \text{iff} \quad \text{for all } i < |\sigma|,\ \sigma_{i+1}[\![K]\!] = 2 \cdot \sigma_i[\![K]\!].$$

The construct *stable* e is true iff the value of the expression e remains unchanged. We can readily define *stable* in terms of *gets*:

$$stable\ e \quad \equiv_{\text{def}} \quad e\ gets\ e.$$

Example (Expressing an invariant condition):

The following formula is true on an interval σ in which I and J are both initially 0 and I repeatedly increases by 1 and J repeatedly increases by 2:

$$(I = 0) \wedge (J = 0) \wedge (I\ gets\ I + 1) \wedge (J\ gets\ J + 2).$$

In any interval for which this is true, J always equals $2I$. Below is a valid property that formalizes this:

$$\models \quad \big[(I = 0) \wedge (J = 0) \wedge (I\ gets\ I + 1) \wedge (J\ gets\ J + 2)\big] \supset \Box(J = 2I).$$

This shows how the operator $\Box$ can express an invariant condition.

Example (Stability):

The formula

$$(I = 1) \wedge \mathit{stable}\, I$$

is true iff I initially equals 1 and its value remains unchanged. This is the same as saying that I always equals 1. The following valid property expresses this equivalence:

$$\models \quad [(I = 1) \wedge \mathit{stable}\, I] \equiv \Box(I = 1).$$

3.4. The Operator *halt*

We can specify that a formula w becomes true only at the end of an interval σ by using the formula *halt* w:

$$\mathit{halt}\, w \quad \equiv_{\text{def}} \quad \Box(w \equiv \mathit{empty}).$$

Thus w must be false until the last state at which time w is true. For example, the formula

$$\mathit{halt}(I > 100)$$

is true on σ iff the value of the variable I exceeds 100 in exactly the last state of σ.

Example (Repeatedly doubling a number):

From what we have so far presented, it can be seen that the formula

$$(I = 1) \wedge \mathit{halt}(I > 100) \wedge (I\ \mathit{gets}\ 2I)$$

is true on an interval where the variable I is initially 1 and repeatedly doubles until it exceeds 100. The following valid implication states that intervals on which this formula is true will terminate upon I equalling the value 128:

$$\models \quad [(I = 1) \wedge \mathit{halt}(I > 100) \wedge (I\ \mathit{gets}\ 2I)] \supset \mathit{halt}(I = 128).$$

4. A Temporal Programming Language

Consider now the formula

$$(M = 4) \wedge (N = 1) \wedge \mathit{halt}(M = 0) \wedge (M\ \mathit{gets}\ M - 1) \wedge (N\ \mathit{gets}\ 2N).$$

This holds true of intervals of length 4 in which M successively runs through the values 4, 3, 2, 1 and 0 and N simultaneously runs through the values 1, 2, 4, 8, and 16. Let us now explore how to automate the process of taking such a temporal formula and finding an interval satisfying it. One way to do this is to develop a procedure that analyzes the formula and either terminates with the length of some acceptable interval and values of the relevant variables in all the interval's states or else fails.

We will use another technique which we call *interval generation*. This approach takes the original formula and scans it once for each state of the interval being generated. We introduce the predicates *input* and *output*. In any state where the predicate $input(v)$ is true, the user can input a value for the variable v. Whenever the predicate $output(e)$ is encountered, the expression e is evaluated and its value is displayed to the user. The net effect is that the temporal formula is "executed" with the predicates *input* and *output* providing communication to the user. For example, the formula given below includes a subformula that always outputs the value of N to the user:

$$(M = 4) \wedge (N = 1) \wedge halt(M = 0) \\ \wedge (M \; gets \; M - 1) \wedge (N \; gets \; 2N) \wedge \Box \, output(N). \qquad (1)$$

As the overall formula is processed, the successive values of N are displayed. Figure 1 shows a sample session in which this is executed.

When the following formula is executed, the user is continually asked for the values of I:

$$\Box \, input(I) \wedge halt(I = 0) \wedge (J = 0) \wedge \Box \, output(J) \wedge (J \; gets \; J + I). \qquad (2)$$

These values are summed into J and J itself is displayed. The interval terminates upon I equalling 0. An execution of this is given in figure 2. Numbers in boxes (e.g., $\boxed{6}$) are input by the user.

The general problem of finding an interval that satisfies a temporal formula is unsolvable. However, there are subsets of temporal logic for which the task is managable. We now present *Tempura*, a programming language based on one such subset.

4.1. Syntax of Tempura

The main syntactic categories in Tempura are locations, expressions and statements. Let us look at each of these separately:

```
Output = 1
State #0 ready.

Output = 2
State #1 ready.

Output = 4
State #2 ready.

Output = 8
State #3 ready.

Output = 16
State #4 ready.

Done!  Computation length = 4.
```

Figure 1: Execution of Formula (1)

```
Input = 6
Output = 0
State #0 ready.

Input = 2
Output = 6
State #1 ready.

Input = 5
Output = 8
State #2 ready.

Input = 0
Output = 13
State #3 ready.

Done!  Computation length = 3.
```

Figure 2: Execution of Formula (2)

4.1.1. Locations

A *location* is a place into which values can be stored and later retrieved. Variables such as I, J and K are permissible locations. In addition, if l is a location, so is the temporal construct $\bigcirc l$.

4.1.2. Expressions

Expressions can be either arithmetic or boolean. All numeric constants and variables are legal arithmetic expressions. In addition, if e_1 and e_2 are arithmetic expressions, so are operations such as the following:

$$e_1 + e_2, \quad e_1 - e_2, \quad e_1 \cdot e_2, \quad e_1 \div e_2, \quad e_1 \; mod \; e_2.$$

In addition, if e is an arithmetic expression then so is the temporal construct $\bigcirc e$.

Relations such as $e_1 = e_2$ and $e_1 \geq e_2$ are boolean expressions. If b, b_1 and b_2 are boolean expressions, then so are the following:

$$\neg b, \quad b_1 \wedge b_2, \quad b_1 \vee b_2, \quad b_1 \supset b_2, \quad b_1 \equiv b_2.$$

The constants *true* and *false* and the construct *empty* are boolean expressions as well.

4.1.3. Statements

Certain temporal formulas are legal statements in Tempura. A statement is either *simple* or *compound*. Simple statements are built from the constructs given below. Here l is a location, e an arithmetic expression and b a boolean expression:

$l = e$	(equality)
empty	(terminate)
$\neg$*empty*	(do not terminate)
input(l)	(input into a location)
output(e)	(output an expression).

The statement $l = e$ stores the value of the arithmetic expression e into the location l.

Compound statements are built from the constructs given below. Here w, w_1

and w_2 are themselves statements and b is a boolean expression:

$w_1 \wedge w_2$	(parallel composition)
$b \supset w$	(conditional execution)
$\bigcirc w$	(next)
$\Box w$	(always)

Note that certain temporal formulas can be used as both boolean expressions and statements. Here are three examples:

$$I = 3, \quad (J = 2) \wedge (K = J + 3), \quad (I = 0) \supset \mathit{empty}.$$

On the other hand, the following legal boolean expressions are not Tempura statements even though they are semantically equivalent to the respective formulas given above:

$$3 = I, \quad (2 = J) \wedge (J + 3 = K), \quad \neg(I = 0) \vee \mathit{empty}.$$

4.2. Some Other Statements

Other constructs such as *gets*, *stable* and *halt* can be readily added to Tempura. One way to do this is to expand these to statements already described. Here are some possible equivalences:

$$\begin{aligned} l \mathit{\ gets\ } e &\equiv \Box\big(\neg \mathit{empty} \supset [(\bigcirc l) = e]\big), \\ \mathit{stable}\, l &\equiv l \mathit{\ gets\ } l, \\ \mathit{halt}\, b &\equiv \Box\big([b \supset \mathit{empty}] \wedge [\neg b \supset \neg \mathit{empty}]\big). \end{aligned}$$

Once we include these statements, programs such as the following can be readily processed:

$$\mathit{input}(I) \wedge (J = 1) \wedge \Box\, \mathit{output}(J) \wedge \mathit{halt}(I = 0) \wedge (I \mathit{\ gets\ } I - 1) \wedge (J \mathit{\ gets\ } 2J).$$

This statement initially requests the input of a value for I and then repeatedly outputs the first few powers of 2 until I is decreased to 0.

4.3. An Interpreter for Tempura

We now briefly outline an interpreter for executing Tempura statements. The interpreter takes a statement and generates an acceptable interval by repeatedly

scanning and modifying the statement until the final state of the interval is reached. A flag named *done* is used to indicate termination. Each iteration of the interpreter corresponds to one state of the interval being generated. In addition to the flag *done*, an environment called *env* maintains each variable's current and next values. Over time, as the statement is executed, the entries in the environment are updated to reflect changes to the variables. The full version of this paper will discuss the implementation of the interpreter in more detail.

As we mentioned earlier, Tempura statements are limited to a subset of temporal formulas. So far we have only mentioned syntactic restrictions. Let us consider some limitations that the interpreter itself imposes on Tempura programs. This will give some idea of the design trade-offs we have made.

4.3.1. Determinism

The interpreter expects the user to completely specify the behavior of variables and to indicate when termination should occur. For example, the statement

$$I \textit{ gets } I + 1$$

lacks information on I's initial value and does not specify when to stop. Thus other details must be included for the interpreter to properly operate. Of course, we could be more lenient by using backtracking and related techniques to resolve such omissions. However, for the sake of the simplicity and efficiency of the interpreter, it seems reasonable at the moment to require explicit, unambiguous information on all aspects of variable behavior.

4.3.2. Left-to-right processing

The interpreter scans statements from left to right. Therefore the statement

$$(J = I + 3) \wedge (I = 0)$$

is not properly handled since the value of I is not yet known during the evaluation of the expression $I + 3$. This can be remedied by reordering the two equalities as follows:

$$(I = 0) \wedge (J = I + 3).$$

4.3.3. Restrictions on *empty*

The construct *empty* as implemented by the interpreter is also subject to restrictions. Consider the following statement for running I through the values 10, 9, ..., 0:

$$(I = 10) \wedge (I \textit{ gets } I - 1) \wedge \textit{halt}(I = 0).$$

This does not execute properly since the definition of the *gets* construct involves a test of the value of *empty*, but this in not determined until the *halt* construct is encountered and processed. The solution here is to simply exchange the two operations, thus yielding the following:

$$(I = 10) \wedge \textit{halt}(I = 0) \wedge (I \textit{ gets } I - 1).$$

4.3.4. Restrictions on the operator ○

The environment only maintains the current and next values of variables. Therefore, an attempt to store in a location such as ○○○I does not work properly because this looks too far into the future.

4.3.5. Why these restrictions?

The limitations outlined here could to some extent be avoided by automated static analysis, by repeated scanning of statements during each state and by modifying the environment to store more values for each variable. Nevertheless, the current interpreter seems to be a reasonable compromise. More experience is needed before a firm conclusion is reached on these matters.

5. Further Temporal Constructs

In addition to the constructs already presented, temporal logic contains various useful operators such as existential quantification ($\exists$) and the temporal operator *chop*. Some of these constructs are rather similar to certain kinds of statements found in Algol and related programming languages. We first extend the syntax and semantics of the temporal logic to include $\exists$ and *chop*. The resulting formalism is called Interval Temporal Logic. Within it we define a number of interval-dependent operators. Tempura is subsequently expanded to include some of these.

5.1. Syntax of *chop* and $\exists$

In addition to the constructs previously introduced, we now permit formulas of the following two forms:

- Chop: $w_1; w_2$, where w_1 and w_2 are formulas.
- Existential quantification: $\exists v.\ w$, where v is a variable and w is a formula.

The following are two simple formulas:

$$(\mathit{stable}\ I); (\mathit{stable}\ J), \qquad \exists I.\ \Box(J = 2I).$$

5.2. Semantics of *chop* and $\exists$

The semantics of these operators are as follows:

- $M_\sigma[\![w_1; w_2]\!] = \mathit{true}$ iff
  for some $i \leq |\sigma|$, $M_{\langle\sigma_0 \ldots \sigma_i\rangle}[\![w_1]\!] = \mathit{true}$ and $M_{\langle\sigma_i \ldots \sigma_{|\sigma|}\rangle}[\![w_2]\!] = \mathit{true}$.
- $M_\sigma[\![\exists v.\ w]\!] = \mathit{true}$ iff
  for some interval $\sigma' \in \Sigma^+$, $\sigma \sim_v \sigma'$ and $M_{\sigma'}[\![w]\!] = \mathit{true}$. Here the relation $\sigma \sim_v \sigma'$ is defined to be true iff the intervals σ and σ' have the same length and agree on the behavior of all variables except possibly the variable v.

Examples

Consider the following states and their assignments to the variables I and J:

	I	J
s	2	4
t	0	4
u	2	3

We assume that s, t and u agree on assignments to all other variables.

The following formula is true any interval on which I is stable for a while and then J is stable for the remainder of the interval:

$$(\mathit{stable}\ I); (\mathit{stable}\ J).$$

The interval $\langle sustst \rangle$ satisfies the formula since I is always 2 on the subinterval $\langle sus \rangle$ and J is always 4 on $\langle stst \rangle$. The formula is also true on the intervals $\langle s \rangle$ and $\langle uuu \rangle$ but it is false on the interval $\langle stuu \rangle$.

The formula

$$\exists I.\ \Box(J = 2I)$$

is intuitively true on any interval on which we can construct an I such that J always equals $2I$. This is the same as saying that J is always even. For example, the interval $\langle ttt \rangle$ satisfies the formula. From the semantics of $\exists$ given previously it follows that to show this we need to construct an interval σ' for which the relation $\langle ttt \rangle \sim_I \sigma'$ is true and which satisfies the subformula $\Box(J = 2I)$. The interval $\langle sss \rangle$ achieves both of these constraints. Therefore $\langle ttt \rangle$ satisfies the original formula. Other intervals satisfying the formula include $\langle sss \rangle$ itself and $\langle sst \rangle$ but not $\langle u \rangle$ or $\langle stut \rangle$. Existential quantification is a tricky concept and the reader should not necessarily expect to grasp it immediately.

5.3. Discussion of the Operator *chop*

The construct *chop* is rather different from the conventional temporal operators $\Box$ and $\bigcirc$. The later examine an interval's suffix subintervals whereas *chop* splits the interval and tests both parts. This facilitates looking at arbitrary subintervals of time.

Harel, Kozen and Parikh [7] appear to be are the first to mention *chop* as a temporal construct. It is considered in more detail by Chandra, Halpern, Meyer and Parikh [4]. In [5] and [16] we use *chop* to facilitate reasoning about timing-dependent digital hardware. Our subsequent work in [18] and [17] uses *chop* to give specifications and properties of simple algorithms and message-passing systems. In the rest of this section we examine *chop* and other ITL constructs and then extend Tempura to include them.

5.4. The Operator *fin*

The formula *fin w* is true on an interval σ iff w is itself true on the final subinterval $\langle \sigma_{|\sigma|} \rangle$. We express *fin w* as follows:

$$\mathit{fin}\ w \quad \equiv_{\text{def}} \quad \Box(\mathit{empty} \supset w).$$

The formula *fin w* is weaker than *halt w* since *fin w* only looks at the last state but *halt w* tests behavior throughout.

An Example

The following formula is true on an interval σ iff $|\sigma| = 3$ and I is initially 1

and repeatedly doubles:

$$(\bigcirc\bigcirc\bigcirc \mathit{empty}) \wedge (I = 1) \wedge (I \mathit{\ gets\ } 2I).$$

One effect is that I ends up equal to 8. This is expressed by the valid implication given below:

$$\models \quad \big[(\bigcirc\bigcirc\bigcirc \mathit{empty}) \wedge (I = 1) \wedge (I \mathit{\ gets\ } 2I)\big] \supset \mathit{fin}(I = 8).$$

5.5. Assignment

The formula $e_1 \rightarrow e_2$ is true for an interval if the expression e_1's initial value equals the expression e_2's final value. We define this as follows:

$$e_1 \rightarrow e_2 \quad \equiv_{\mathrm{def}} \quad \exists A.\ \big[(\mathit{stable}\ A) \wedge (A = e_1) \wedge \mathit{fin}(e_2 = A)\big],$$

where the variable A does not occur free in either e_1 or e_2. The stability of A is used to compare the values of e_1 and e_2 at different times. We call this construct *temporal assignment*. For example, the formula $I + 1 \rightarrow I$ is true on an interval σ iff the value of $I + 1$ in the initial state σ_0 equals the value of I in the final state $\sigma_{|\sigma|}$. If desired, we can reverse the direction of the arrow:

$$I \leftarrow I + 1.$$

The formula

$$(I \leftarrow I + 1) \wedge (J \leftarrow J + I)$$

is then true in an interval iff I increases by 1 and in parallel J increases by I. Similarly, the following specifies that the values of the variables A and B are exchanged:

$$(A \rightarrow B) \wedge (B \rightarrow A).$$

Unlike assignment in conventional programming languages, temporal assignment only affects variables explicitly mentioned; the values of other variables do not necessarily remain fixed. For example, the formulas

$$I \leftarrow I + 1$$

and

$$(I \leftarrow I + 1) \wedge (J \leftarrow J)$$

are not equivalent since the first formula does not require J's initial and final values to be equal.

Example (Sequential composition of assignments):

The formula

$$(K+1 \rightarrow K);(K+2 \rightarrow K)$$

is true on an interval σ iff there is some $i \leq |\sigma|$ such that the subformula $K+1 \rightarrow K$ is true on the subinterval $\langle \sigma_0 \ldots \sigma_i \rangle$ and the subformula $K+2 \rightarrow K$ is true on remaining subinterval $\langle \sigma_i \ldots \sigma_{|\sigma|} \rangle$. The net effect is that K increases by 3. This is expressed by the following property:

$$\models \quad \big[(K+2 \rightarrow K);(K+1 \rightarrow K)\big] \supset (K+3 \rightarrow K).$$

5.6. Conditional Formula

We let the boolean construct *if* w_1 *then* w_2 *else* w_3 be true on an interval if either w_1 and w_2 are true or $\neg w_1$ and w_3 are true. This can be formalized by the following definition:

$$\textit{if } w_1 \textit{ then } w_2 \textit{ else } w_3 \quad \equiv_{\text{def}} \quad (w_1 \supset w_2) \wedge (\neg w_1 \supset w_3).$$

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

The temporal formula

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

is true in any interval where I's value in the final state equals the maximum of the values of I and J in the initial state. This can be seen by case analysis on the test $I \geq J$.

Let the function $\max(i,j)$ equal the maximum of the two values i and j. The following temporal formula also places the maximum of I and J into I:

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

The equivalence of the two approaches is expressed by the following property:

$$\models \quad \big[I \leftarrow \max(I,J)\big] \equiv \big[\textit{if } I \geq J \textit{ then } (I \leftarrow I) \textit{ else } (I \leftarrow J)\big].$$

5.7. While-Loops

Within temporal logic, various kinds of iterative constructs can be introduced as formulas. One of the most important is the *temporal while-loop*. The basic form is similar to that for a while-loop in Algol:

$$\textit{while } w_1 \textit{ do } w_2,$$

where w_1 and w_2 are themselves formulas.

The while-loop obeys the following general expansion property:

$$\textit{while } w_1 \textit{ do } w_2 \quad \equiv \quad \textit{if } w_1 \textit{ then } \big(w_2; [\textit{while } w_1 \textit{ do } w_2]\big) \textit{ else empty}.$$

Thus, if w_1 is true, the body of the loop, w_2, is examined after which the loop is repeated. If w_1 is false, the interval must have length 0. It is possible to express while-loops using operators presented earlier. For-loops and other iterative constructs can also be handled.

Example (Computing the greatest common divisor):

Consider the following assignment which specifies that N's final value equals the initial value of the greatest common divisor of M and N:

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

The formula below implies this:

$$\textit{while } (M \neq 0) \textit{ do } \big([M \leftarrow N \textit{ mod } M] \wedge [N \leftarrow M]\big).$$

5.8. The Construct *skip*

The construct *skip* is true on an interval σ iff σ has length 1. We can express *skip* as follows:

$$\textit{skip} \quad \equiv_{\text{def}} \quad \bigcirc \textit{empty}.$$

Example (Measuring the length of an interval):

An interval's length can be tested using *skip* and *chop*. For example, the formula

$$\textit{skip}; \textit{skip}; \textit{skip}$$

is true on intervals having length 3. Since *chop* is associative, we can omit parentheses without being ambiguous.

Example (Unit-length iterations):

The following while-loop repeatedly decrements I and sums I into J over each unit of time until I equals 0:

$$\mathit{while}\,(I = 0)\;\mathit{do}\,\big(\mathit{skip} \wedge [I \leftarrow I - 1] \wedge [J \leftarrow J + I]\big).$$

The body of the loop contains the *skip* operator in order that the length of each iterative step be 1. The behavior can also be expressed using *halt* and *gets*. Here is a semantically equivalent way of doing this:

$$\mathit{halt}(I = 0) \wedge (I\ \mathit{gets}\ I - 1) \wedge (J\ \mathit{gets}\ J + I).$$

5.9. Incorporating ITL Constructs into Tempura

We now extend Tempura to include statements based on the ITL constructs just introduce. The set of simple statements is now expanded to include the following:

$l \leftarrow e$	(assignment)
skip	(unit interval)

Compound statements are extended to include the following:

$w_1; w_2$	(sequential composition)
if b *then* w_1 *else* w_2	(conditional statement)
while b *do* w	(while-loop)
$\exists v.\ w$	(existential quantification).

Example (Hiding a variable):

The following toy program has two distinct variables both called I:

$$(I = 0) \wedge (I\ \mathit{gets}\ I + 1) \wedge \mathit{halt}(I = 5) \wedge \exists I.\ \big[(I = 1) \wedge (I\ \mathit{gets}\ 3I) \wedge \Box\ \mathit{output}(I)\big].$$

The first I runs from 0 to 5 and in parallel the second I is repeatedly tripled from 1 to 243. The use of existential quantification ($\exists$) keeps the two I's separate and in effect hides the second one. As this example illustrates, we can use $\exists$ to create locally scoped variables.

```
Input = 4
Output = 0
State #0 ready.

Output = 4
State #1 ready.

Output = 7
State #2 ready.

Output = 9
State #3 ready.

Output = 10
State #4 ready.

Done!  Computation length = 4.
```

Figure 3: Execution of Formula (3)

Example (Computing sums):

The following Tempura program uses a while-loop to compute a sum:

$$\begin{aligned} &\mathit{input}(I) \wedge (J = 0) \wedge \Box\, \mathit{output}(J) \\ &\quad \wedge \left[\mathit{while}\ I \neq 0\ \mathit{do}\ \left(\mathit{skip} \wedge [I \leftarrow I - 1] \wedge [J \leftarrow J + I]\right)\right]. \end{aligned} \tag{3}$$

The user inputs a value and the program then determines the sum of the numbers up to that value. Figure 3 shows a sample session in which the user requests the sum of the values up to 4.

Example (Computing powers):

Consider the problem of finding the value of the expression I^J and placing this in another variable K. We can specify this using the temporal assignment

$$K \leftarrow I^J.$$

The following Tempura program achieves this by looking at J's binary structure:

$$(K = 1) \wedge \left[\mathit{while}\ (J > 0)\ \mathit{do}\ (\mathit{skip} \wedge w)\right],$$

where the statement w has the form

$$\begin{aligned} &\mathit{if}\ (J\ \mathit{mod}\ 2 = 0) \\ &\quad \mathit{then}\ \left[(I \leftarrow I \cdot I) \wedge (J \leftarrow J \div 2) \wedge (K \leftarrow K)\right] \\ &\quad \mathit{else}\ \left[(I \leftarrow I) \wedge (J \leftarrow J - 1) \wedge (K \leftarrow K \cdot I)\right]. \end{aligned}$$

6. Experience and Further Work

Based on the ideas discussed here, we have implemented a Tempura interpreter in Lisp. Tempura programs for such tasks as parallel quicksorting, fast fourier transforms, and simulation of a hardware multiplier have been written and successfully run. We intend to discuss some of this work in the full version of this paper. Together with Roger Hale, a PhD student, we are experimenting with the system and are trying out further programming examples. Another Tempura interpreter has been developed as a programming project by Nigel Beckwith.

Tempura contains various other features including for-loops, arrays and procedures. All of these can be fitted into the framework of temporal logic (see [5] and [16] for more details). The main change to the interpreter is the addition of a memory for maintaining the values of variables and array elements. The environment entries no longer store values themselves but instead point to cells in the memory. This approach is very similar to that used in executing conventional block-structured languages.

7. Some Related Work

The functional programming language *Lucid* [2,3] developed by Ashcroft and Wadge is similar to parts of Tempura. For example, the Lucid program

$$I = 0 \ fby \ (I + 1); \quad J = 0 \ fby \ (J + I)$$

roughly corresponds to the temporal formula

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

This illustrates how the operator *gets* can be handled in Lucid. On the other hand, Algol-like temporal constructs such as $\leftarrow$, *chop* and *while* do not have direct analogs in Lucid. Thus, a Tempura construct such as

$$while \ (M \neq 0) \ do \ \big(skip \wedge [M \leftarrow M - 1] \wedge [N \leftarrow 2N]\big)$$

cannot be readily translated. In [1], a calculus is developed for reasoning about Lucid programs.

Hehner [8] views programs as logical predicates over input-output behavior of variables. Various Algol-based constructs such as assignment ("$:=$"), sequencing

(";") and while-loops are treated in this manner. Time is introduced by means of an explicit clock variable. Hehner then goes on to introduce concurrency through CSP-like channels. It seems too early to compare this approach with temporal logic.

Manna and Wolper [14] investigate techniques for synthesizing CSP synchronization code from temporal logic specifications. Examples include a reader-writer system and the dining philosphers' problem. Mishra and Clarke [15] use temporal logic to describe asynchronous digital circuits and then generate corresponding finite-state automata. Tang [21] and Manna and Pneuli [13] discuss ways of translating conventional programming constructs into transition systems described in temporal logic. The resulting temporal descriptions can then be used to reasoning about the original programs. In contrast to these approaches, we have worked on developing a set of temporal operators that facilitate programming directly in temporal logic. This bypasses the need for two notations and omits any synthesis from specifications.

8. Conclusions

The present work has investigated Tempura, a program language based on Interval Temporal Logic. The ITL formalism provides a way to treat such programming concepts as assignment and loops as formulas about intervals of time. Therefore, Tempura programs, their specifications and their properties can all be expressed in the same formalism. In the future, we hope to use Tempura for simulating various hardware devices and message-passing systems. In addition, we plan to explore the feasibility of using it as a general-purpose programming language.

Acknowledgements

We wish to thank Mike Gordon, Roger Hale and Edmund Ronald for stimulating conversations and suggestions. Funding from the British Science and Engineering Research Council is gratefully acknowledged.

9. References

[1] E. A. Ashcroft and W. W. Wadge. *Lucid: A formal system for writing and proving programs. SIAM Journal of Computing 5*, 3 (September 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] A. Chandra, J. Halpern, A. Meyer, and R. Parikh. Equations between regular terms and an application to process logic. *Proceedings of the 13-th Annual ACM Symposium on Theory of Computing*, Milwaukee, Wisconsin, May, 1981, pages 384–390.

[5] 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.

[6] D. Harel. *First-Order Dynamic Logic.* Number 68 in the series *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1979.

[7] D. Harel, D. Kozen, and R. Parikh. "Process logic: Expressiveness, decidability, completeness." *Journal of Computer and System Sciences 25*, 2 (October 1982), pages 144–170.

[8] E. C. R. Hehner. "Predicative programming (parts I and II)." *Communications of the ACM 27*, 2 (February 1984), pages 134–151.

[9] C. A. R. Hoare. "An axiomatic basis for computer programming." *Communications of the ACM 12*, 10 (October 1969), pages 576–580, 583.

[10] C. A. R. Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*, pages 61–71. Academic Press, London, 1972.

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

[12] 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.

[13] Z. Manna and A. Pnueli. How to cook your favorite programming language in temporal logic. *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas, January, 1983, pages 141–154.

[14] Z. Manna and P. L. Wolper. "Synthesis of computing processes from temporal logic specifications." *ACM Transactions on Programming Languages and Systems 6*, 1 (January 1984), pages 68–93.

[15] B. Mishra and E. M. Clarke, Automatic and hierarchical verification of asynchronous circuits using temporal logic. Technical report CMU-CS-83-155, Department of Computer Science, Carnegie-Mellon University, September, 1983.

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

[17] B. Moszkowski. A temporal analysis of some concurrent systems. To appear in the proceedings of the STL Workshop on Concurrency, Cambridge, England, September, 1983.

[18] B. Moszkowski and Z. Manna. Reasoning in interval temporal logic. Technical report STAN-CS-83-969, Department of Computer Science, Stanford University, July, 1983.

[19] V. R. Pratt, Semantical considerations on Floyd-Hoare logic. *Proceedings of the 17-th Annual IEEE Symposium on Foundations of Computer Science*, Houston, Texas, October, 1976, pages 109–121.

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

[21] C. Tang. Toward a unified logic basis for programming languages. *Proceedings of IFIP Congress 83*, Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1983, pages 425–429.