Parallel Programming in Temporal Logic

Roger Hale and Ben Moszkowski
Computer Laboratory, University of Cambridge,
Corn Exchange Street, Cambridge CB2 3QG, England

1 Background

For some time now temporal logic has been an accepted tool for reasoning about concurrent programs. More recently, we have shown that temporal logic specifications may be executed directly [Mos86, Ha187], and have tested our methods on a number of concurrent systems, ranging from hardware devices to high-level algorithms. Out of this work has come the programming language, Tempura, which is a subset of Moszkowski’s Interval Temporal Logic [Mos83].

Having had some experience in using Tempura, we believe that there is a need to compare it with other parallel programming paradigms. For this purpose we have chosen four algorithms from the literature. The first one is a simple scheduling algorithm, the second is for the evaluation of parsed expressions, the third is a graph traversal algorithm, and the final one performs a mergesort on an array of processors. These algorithms take up the central section of this paper, but first there is a description of Interval Temporal Logic (ITL) and a short account of Tempura. Our review of ITL is not intended to be a tutorial, but is sufficient for an understanding of the paper. The paper concludes with a discussion of where ITL stands in relation to some other parallel programming paradigms.

2 Interval Temporal Logic

Like all temporal logics, ITL extends classical first-order logic with operators for relating behaviour at different times. This means that the familiar logical operators, such as “and” and “not”, all appear in ITL, but there are in addition some special temporal operators which stand for notions involving time, like “... and then ...”, “always”, and “next”.

2.1 The Computational Model

A computation, in our view, is a sequence of one or more discrete steps, each step being characterised by the values of some collection of variables. In principle, if it is known what all the variables are and what values they can take, then the (usually infinite) set of all possible computations, Σ, is determined. A particular computation, σ ∈ Σ, is a sequence of steps,

σ = (σ₀, ..., σₙ),

the length of the computation, n, being the number of transitions between states.

The formulae which describe the properties of computations are composed as follows. Syntactically, any formula of classical predicate logic with quantification is also a formula of ITL. Furthermore, any formula p may be prefixed by one of the temporal operators □ (“always”), ◦ (“sometimes”) or ○ (“next”) to form a new property, and any two formulae, p₁ and p₂, may be combined using the so-called chop operator to give a new property p₁ ; p₂ (“p₁ and then p₂”). The semantics of these operators are described informally below and summarised in figure 1.
2.2 First Order Logic and Parallel Composition

First order logic is contained within ITL, and therefore all the classical operators appear in ITL too. For example, if the formulae $p_1$ and $p_2$ describe the behaviours of two processes, then their logical conjunction $p_1 \land p_2$ describes the computation that results when $p_1$ and $p_2$ are run in parallel. If $p_1$ holds the value of $X$ constant, and $p_2$ gives it an initial value of 7, then their conjunction constrains $X$ to have the value 7 on every step.

In a similar way, the disjunction of two process descriptions $p_1 \lor p_2$ describes a computation on which one or the other may occur, and the negation $\neg p$ is true of any process which doesn’t behave like $p$.

2.3 Variables and Quantifiers

To make life easier for ourselves, we distinguish three kinds of variables:

- **Interval variables**, whose values depend on entire computations (they are functions of intervals).
- **State variables**, which depend only on state. To make them into functions of computations, we adopt the convention that the value of a state variable on a sequence of states is its value on the first of those states.
- **Static variables**, which do not change in value during a computation.

Interval variables do not seem very useful for describing deterministic computations. Consequently only state and static variables are used in the following, and these we differentiate by a simple naming convention. Names beginning with a capital letter (like $I$ and $\text{Temp}$) belong to state variables, whereas those starting with a lower case letter (such as $i$ and $\text{temp}$) are static.

Predicate symbols in ITL are analogous to computational procedures; they allow us to parameterise specifications. For example, an array of $n$ identical processors operating in parallel could be specified as follows:

$$p(0) \land \ldots \land p(n - 1).$$

An equivalent, but more concise specification uses *universal quantification*:

$$\forall i < n : p(i)$$

(read “for all $i$ less than $n . . . $”). The dual construction, *existential quantification*, asserts that there is at least one value of each quantified variable for which the associated statement is true. For example, the assertion

$$\exists I : (p_1(I) \land \exists I : p_2(I))$$

can be satisfied if we can find one computation on which variable $I$ behaves according to $p_1(I)$, and another on which $I$’s behaviour is given by $p_2(I)$, provided that both computations agree on the behaviour of every variable except $I$. This is like declaring new variables with local scope.

So far we have only shown how to refer to a computation as a whole. Let us now look at some temporal operators which focus on parts of a computation.

2.4 Chop and Sequential Behaviour

The basic sequencing operator of ITL is *chop* ($;$). The construction $p_1 ; p_2$ is satisfied by a computation which can be split into two parts such that $p_1$ is true on the first, and $p_2$ on the