Appears in *Journal of Symbolic Computation*, vol. 22, no. 6 (Dec.), pp. 730–733

The Programming Language Tempura

Ben Moszkowski
Department of Electrical and Electronic Engineering
University of Newcastle upon Tyne
Newcastle NE1 7RU
Great Britain
Internet: Ben.Moszkowski@ncl.ac.uk

1 Introduction

Tempura [12] is an experimental programming language based on executable subsets of Interval Temporal Logic (ITL) [10, 7, 11]. We developed it in order to narrow the gap between temporal logics and the imperative programs they typically describe. Tempura contains destructive assignment constructs, ";" (sequential composition) and while-loops. It consequently looks much more like a conventional imperative programming language than most other executable temporal logic systems. Unlike most logic-based languages, it is very deterministic and offers no facilities for backtracking. This approach has been adapted primarily for execution efficiency. A number of temporal constructs have been included in Tempura and can be used for hardware simulation and other applications. These include $\Box$ (*always*), $\bigcirc$ (*next*) and a form of projection between different granularities of time. Almost all of Tempura's constructs can be formally defined using conventional first-order logic and the ITL operators *skip*, ";" (*chop*) and "*" (*chop-star*). These three operators respectively provide ways to measure interval length, sequentially combine formulas and iterate a formula. The basic semantics of a Tempura program is readily obtained by considering it to be an ITL formula. Consequently, logical implication can be used to relate an ITL specification with a Tempura program that implements it.

2 Language

Tempura programs typically consist of a logical conjunction of various ITL formulas and are executed in a simulated time consisting of a sequence of one or more discrete states. Here is an example:

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

This executes over 4 units of discrete time (i.e., 5 states). In the initial state, M equals 4 and N equals 1. Over adjacent states, M increases by 1 and N doubles until the computation terminates when M equals 0. The behavior of M and N can be also expressed by the following logically equivalent ITL formula:

$$\begin{array}{c} M = 4 \wedge N = 1 \wedge \bigcirc(M = 3 \wedge N = 2) \wedge \bigcirc\bigcirc(M = 2 \wedge N = 4) \\ \wedge \bigcirc\bigcirc\bigcirc(M = 1 \wedge N = 8) \wedge \bigcirc\bigcirc\bigcirc\bigcirc(M = 0 \wedge N = 16). \end{array} \tag{2}$$

A slight variation of this can be directly executed by the Tempura interpreter.

The following ITL fragment, which operates over 1 unit of time (2 states), decreases M by 1 and doubles N:

$$\mathit{skip} \wedge M \leftarrow M - 1 \wedge N \leftarrow 2N. \tag{3}$$

The *skip* construct specifies that the interval has length 1. The two *temporal assignments* modify in parallel the values of M and N. Unlike convention imperative assignment, temporal assignment lacks *framing* and does not ensure the all nonassigned variables remain unchanged. Instead, this must be explicitly stated by using multiple temporal assignments in parallel or by using the *stable* construct (e.g., *stable* A). The fragment (3) can be included in a while-loop and combined with initialization to obtain the program now shown:

$$M = 4 \wedge N = 1 \wedge \mathit{while}\ M \neq 0\ \mathit{do}\ (\mathit{skip} \wedge M \leftarrow M - 1 \wedge N \leftarrow 2N). \tag{4}$$

This is logically equivalent in ITL to formulas (1) and (2). Note that the conjunction of two temporal assignments performs them in parallel. Therefore, the following exchanges the values of the variables A and B in one unit of time:

$$\mathit{skip} \wedge A \leftarrow B \wedge B \leftarrow A.$$

Universal quantification ($\forall$) provides a way to perform a large number of activities in parallel. Local scoping of variables is typically achieved by enclosing them in existential quantifiers ($\exists$). There are also some constructs for input and output as well as projection between different time granularities.

Timing constraints can be specified. The *len* construct illustrated in the next program offers one way to do this:

$$M = 4 \wedge N = 1 \wedge \mathit{len}(M) \wedge M\ \mathit{gets}\ M + 1 \wedge N\ \mathit{gets}\ 2N. \tag{5}$$

This is logically equivalent in ITL to formulas (1), (2) and (4). The *len* construct can also be used in nested constructs. For example, here is a program which describes a digital signal X ranging over 0 and 1 with stable periods of equal widths over an overall interval of 18 units of time (19 states):

$$X = 0 \wedge \mathit{for}\ 3\ \mathit{times}\ \mathit{do}\ \big((\mathit{len}(5) \wedge \mathit{stable}\ X); (\mathit{skip} \wedge X \leftarrow \neg X)\big).$$

This can be combined with other Tempura fragments to model a circuit containing gates which respond to X's changes.

As with any realistic logic-based programming language, there are major restrictions on permitted Tempura constructs. The language's lack of nondeterminism limits the logical operator $\vee$ (*or*) to being used only in boolean tests (as in most programming languages). The temporal operator $\Diamond$ (*sometimes*) is completely absent from Tempura.

3 Implementation

Moszkowski [12] presents the design of a Tempura interpreter which executes a program by successively examining one state at a time. The interpreter determines the program's effect on variables in the current state and also extracts from the program those parts which should be postponed until later on. During the processing of a program in a given state, an immediate assignment of the form $v = e$ sets the variable v's current value to be that of the expression e. Sometimes this requires multiple passes over a formula as the following illustrates:

$$J = I + 1 \wedge I = 2.$$

Of course, circular assignments such are $I = I + 1$ are not valid. Some assignments are postponed until the next state. For example, the following equality increases $\bigcirc I$ (I's next value) by 1:

$$(\bigcirc I) = I + 1.$$

As each program piece is analyzed, those parts which do not affect the current state are pushed off until later. For example, the fragment $\Box(K = 1)$, which sets K to 1 in all states, is reduced to the equivalent formula shown below:

$$K = 1 \wedge \textcircled{w}\Box(K = 1).$$

Here $\textcircled{w}$ (*weak-next*) is a weak form of the $\bigcirc$ operator and is trivially true on any interval with only one state. The net effect of the reduction is to assign K the value 1 in the current state and to save the formula $\Box(K = 1)$ for evaluation in the successor state if there is one. A sequential formula of the form $S; S'$ is evaluated by first interpreting the left subformula S until its associated subinterval terminates and then interpreting the right subformula S' from then on until the completion of the overall interval. While-loops and other interative constructs are processed in an analogous way.

We originally implemented a interpreter for Tempura in Lisp. A port to C done by R. Hale and sample programs are available from us by email.

4 Discussion

Tempura provides a framework for investigating the relationship between conventional imperative programming and temporal logics. Although Tempura has never been widely used, it has helped to generate interest in executable temporal logic systems. For example, the Tokio [4] and MetateM [1] systems have both been influenced by it. In addition, various researchers have applied Tempura to hardware simulation and other areas where timing is important [2, 3, 5, 8, 9]. Ruddle [14] favorably rated ITL and Tempura as the basis of a formal method for fast prototyping of real-time C programs.

Some deficiencies in Tempura such as the lack of any kind of framed assignment have stimulated research on extending Interval Temporal Logic [6, 13]. The continued vitality of conventional programming languages suggests that further development on executable imperative subsets of temporal logics is a worthwhile endeavor.

References

[1] H. Barringer, M. Fisher, D. Gabbay et al. MetateM: A framework for programming in temporal logic. In: Proc. REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness, LNCS 430, Springer-Verlag, Berlin, 1989.

[2] R. Dowsing and R. Elliot. A higher level of behaviorial specification: An example in Interval Temporal Logic. In: Proc. EUROMICRO '91, 1991, 517–524.

[3] R. Dowsing, E. Elliot and I. Marshall. Automated technique for high-level circuit synthesis from temporal logic specifications. IEE Proc.-Comput. Digit. Tech., Vol. 141, No. 3, 1994, 145–152.

[4] M. Fujita, S. Kono et al. Tokio: Logic programming language based on temporal logic and its compilation to Prolog, in: Proc. 3rd Int'l. Conf. on Logic Programming, LNCS 225, Springer-Verlag, Berlin, 1986, 695–709.

[5] R. Hale. Temporal logic programming. In: Temporal Logics and Their Applications (A. Galton, ed.), Academic Press, London, 1987, 91–119.
[6] R. Hale. Programming in Temporal Logic. PhD Thesis, University of Cambridge, Cambridge, England, 1988.
[7] J. Halpern, Z. Manna, and B. Moszkowski. A hardware semantics based on temporal intervals, in: ICALP83, LNCS 154, Springer-Verlag, Berlin, 1983, 278-291.
[8] D. Kilis, A. C. Esterline, and J. R. Slagle. Specification and verification of network protocols using executable temporal logic. In: Proc. IFIP Congress '89, North Holland Publishing Co, Amsterdam, 1989, 845–850.
[9] R. W. Lichota. Evaluating Hardware Architectures for Real-Time Parallel Algorithms using Temporal Specifications. PhD Thesis, University of California at Los Angeles, 1988.
[10] B. Moszkowski. Reasoning about Digital Circuits, PhD thesis, Stanford Univ., 1983. (Available as Tech. Rep. STAN–CS–83–970, 1983).
[11] B. Moszkowski. A temporal logic for multilevel reasoning about hardware, Computer, Vol. 18, No. 2, 1985, 10–19.
[12] B. Moszkowski. Executing Temporal Logic Programs, Cambridge Univ. Press, Cambridge, UK, 1986.
[13] B. Moszkowski. Embedding imperative constructs in Interval Temporal Logic. Internal memorandum EE/0895/M1, Dept. of Elec. and Elec. Eng., University of Newcastle, UK, 1995. Available from the author (Ben.Moszkowski@ncl.ac.uk).
[14] A. R. Ruddle. Formal methods in the specification of real-time, safety-critical control systems. In: Proc. Z User Workshop (J. P. Bowen and J. E. Nicholls, eds.), Springer-Verlag, London, 1992, 131–146.