Timed Common Lisp: The Duration of Deliberation

Scott D. Anderson
Spelman College
Atlanta, GA
anderson@auc.edu

Paul R. Cohen
Experimental Knowledge Systems Laboratory
University of Massachusetts at Amherst
cohen@cs.umass.edu

1 Introduction

The two key features of Deliberation Scheduling and Anytime Algorithms are the duration of the computation and the resulting quality. Clearly, quality can be difficult to define and highly dependent on the domain. Duration, on the other hand, seems straightforward: how long the computation takes. But on what processor? Should the processor matter? What code counts in the computation? How is that code's duration modeled? These questions are addressed in this paper.

Our work on duration modeling stems from our research on simulation systems for real-time planning [1]. One such system is PHOENIX [2], which simulates forest fires burning in Yellowstone National Park. Because the firefighters plan how to put out the fire while the fire is burning, there is time-pressure on their reasoning, and they may need to think about how much planning to do. The mechanism by which PHOENIX integrates the thinking of agents with the discrete-event simulation of the environment is to advance the simulation clock depending on the amount of CPU time used by the agent. For example, the default setting in PHOENIX is that one CPU-second corresponds to five minutes of simulation time.

This CPU-time approach is standard among AI simulators for real-time planning [1]. Unfortunately, there are problems with using CPU time, all of which we have suffered while using PHOENIX:

Variance: Small, random variations in the measurement of CPU time result in random variation in the behavior of the simulation. This can make it difficult to replicate a particular simulation state, whether for debugging, demonstration, or experimentation.

Platform-dependence: The simulation behaves differently from one Lisp platform to another. This exacerbates the variance problem and puts unwanted noise in data from large experiments in which trials are run on many different machines.

Interference: Adding code to record or print data, say for debugging, demonstrations, or to measure quality, affects the CPU time of the code, which in turn affects the behavior of the simulation. This is something like the Heisenberg principle in physics: the act of observing the code affects the code. While the Heisenberg principle may be true in the real world, it is hardly convenient for experimental scientists.

Essentially, all these troubles are "noise" that comes from using CPU time. Consequently, we looked for another way of measuring how much computation an agent has done, one that gives us replicability of simulation states.

2 Duration Modeling

Our basic idea for modeling a computation's duration is to advance the clock by some amount for each "primitive" that is executed. If these increments depend only on the code that is executed and not the Lisp platform, the duration of the code will be invariant. What remains is to decide what a primitive is and how the increments are determined.

2.1 Low-level Models

A "low-level" primitive is a primitive of the Common Lisp language, such as `car`, `+`, or `subst`. By using low-level models, you can retain much of the flavor of the CPU-time approach, because the duration is tied quite tightly to exactly what code executes. We have implemented a language in which every primitive of Common Lisp is shadowed, so that the functions have the same semantics but also advance the clock by a certain amount. (We call this language Timed Common Lisp or TCL.) You program in TCL exactly as in Common Lisp—the two are essentially the same from the programmer's viewpoint. The difference is that TCL primitives advance the clock.

Of course, we will not want to advance the clock by the same amount for each TCL primitive. We will not even want to advance it in the same way. For example, `car` should advance the clock by a small constant; most primitives fall into this category, although the constants are all different. Functions like `+`, on the other hand, should advance the clock depending on the number of arguments they get. (The duration of arithmetic primitives can also depend on the types of the arguments, although the current TCL implementation does not do so.) The duration of a function like `member`, which searches a list for some element, should depend on the length of the list. For a function like `make-array`, the duration should de-

pend on the number of elements in the array.

In implementing TCL, we have defined about two dozen classes of such *duration models*. A duration model is some measure of the amount of work some primitive does. This measure is then multiplied by a coefficient to yield the actual duration of the primitive with that duration model. Duration models are entirely analogous to the "big-O" notation of complexity theory. A constant time function like `car` has a duration model that is $O(1)$, while a function like * has a duration model that is $O(n)$ where n is the arity (number of arguments) of the function. The duration model of `sort` is $O(n \log n)$, where n is the number of arguments to be sorted.

Given these duration models, a "duration database" is then defined. Here are some excerpts:

```
(define-cl-primitives
    append          append-operations 2
    butlast         :length           2
    intersection    set-operations    0
    make-array      array-dims        2
    ninth           :constant         9
    not             :constant         1
    reverse         :length           3
    set-difference set-operations     3)
```

Each line names a primitive of TCL (and Common Lisp), a class of duration model (such as `:constant` for functions like `not` or `ninth`) and the coefficient to be used (such as 1 for `not` or 9 for `ninth`).

These coefficients are arbitrary. Any set of values will give us the desired noise-free measure of duration. However, in our laboratory, we have a set of coefficients defined to correspond roughly with the times for these functions on the Texas Instruments Explorer,^{tm} so that we can duplicate the behavior of the PHOENIX simulator. Other researchers may choose to duplicate the timing of other platforms on which they are currently measuring CPU time. TCL will also provide tools to help with determining what coefficients to use.

By using these low-level models, TCL can report numbers that look like CPU time, but are noise-free and can be replicated on any Common Lisp platform, since TCL runs on any Common Lisp.

2.2 High-level Models

The fundamental operations of an AI program need not be reduced to the primitives of Common Lisp—we can define duration models at a higher level. For example, a chess-playing program might define "evaluating a board position" or "generating a move" as a fundamental "cognitive primitive." The duration of some computation is then $O(f(n,m))$, where m and n are the number of these higher level primitives; for example, m could be the number of board positions evaluated and n could be the number of moves generated, and f is some arbitrary function of those numbers, determining the duration of a move.

This is a natural approach to modeling the duration of anytime algorithms, since many anytime algorithms use iterative improvement or similar approaches where there is a natural "unit" of duration (and quality). By encapsulating each iteration as a TCL primitive with its own duration model, we can reduce the overhead of TCL (see section 4) and have a program whose duration is simpler and easier to understand. It's hard to look at a CPU time and know that it's "right," but if an iterative improvement algorithm reports a duration of, say, 70, and each iteration takes 5 time units, it's pretty clear what's going on.

Of course, as with the primitives of Common Lisp, we don't want to confine ourselves to constant-time models. TCL allows duration models to be defined as arbitrary functions of the primitive's arguments and the computational state of the system. The model can even be pseudo-random, if that's desirable. To achieve our goal of replicability, the model need only be a deterministic computation.

An agent doing deliberation scheduling needs a simple, declarative representation of how long thinking will take. High-level cognitive primitives can help here, especially since the duration models are stored in a TCL database that is accessible to the agent. If the duration model is not a simple constant, the agent can still try to predict how long the computation will take by guessing at the aspects of the simulation state used by the duration model. For example, it might be reasonable to guess at the number of board positions that will be evaluated during a move. It certainly seems easier to guess at that number than to guess at the amount of CPU time that the move would take. Of course, a historical approach can also be used, where the durations that occurred on previous runs are used for prediction; these historical durations can also be stored in the TCL database.

Using a high-level model also allows for a new class of experiments in which the durations of different cognitive primitives are independently controlled. For example, if "move generation" and "board evaluation" are two cognitive primitives in a chess agent, we can modify the duration model for one primitive independently of the other to see the effect on performance. In principle, one can also alter the duration model for `car` independently of that of `cdr`, but there are no interesting research questions posed by that manipulation. By moving to high-level primitives, one can ask sensible questions about duration/quality tradeoffs.

2.3 Integrating Models

High-level and low-level duration models are not mutually exclusive. In the call-tree of a program, the durations of higher functions can either be determined by the code they call, even down to the lowest Lisp primitives, or they can be determined by independent duration models. This boundary is analogous to the the A|B distinction described by Cooper et al. [3], where the cognitive primitive is above the line (A) while the algorithm is below the line (B)—the line demarcates the boundary between theoretical commitment and implementation detail. TCL makes no distinctions between the levels of primitives and so can easily admit a mixture of both ways to model duration. For example, uninteresting sections of the code can be given deterministic durations by using the built-in TCL durations of low-level primitives, or they can be given simple high-level durations, such as a constant. Meanwhile, interesting sections of the code—code whose behavior is anytime or involves similar tradeoffs between deliberation quality and speed—can be given more exacting duration models.

3 Non-interfering Code

So far, we've described how the clock advances as each primitive executes. What if we don't want the clock to advance? Suppose, for example, we put in a `print` statement either to debug or demonstrate the program's behavior. We don't want that insertion to affect the behavior of the simulation. With a CPU-time approach, it can be hard to turn off the clock, but with TCL it's trivial. Any code that shouldn't advance the clock is wrapped in a `free` form. For example, the following reports what the agent is thinking about, without affecting its thoughts or their duration:

```
(defun think ()
    ...
    (free (format t "Thinking about ~s~%"
                    current-thought))
    ...)
```

This ability is particularly important in anytime algorithms and deliberation scheduling, since we will want to insert code to measure and report the quality of the result, yet we don't necessarily want that code to affect the algorithm's behavior. For example, measuring the quality of a tour in the TSP (Traveling Salesman Problem) might be a non-trivial computation that is entirely separate from the tour-improvement computation and therefore should be off the clock. Even if we want the quality computation to be on the clock, we may also be saving the (duration,quality) pair to a file or database, for future reference in deliberation scheduling. TCL allows those operations to be done without interfering with the simulation's behavior.

4 Overhead

What are the disadvantages of using TCL? There are no notational disadvantages, since it looks just like Common Lisp and requires no commitment to a particular agent- or cognitive-architecture. The advancing of the clock, however, does entail an inevitable overhead. Quite simply, the code is doing more work. Therefore, there will be some slowdown of the user's code.

It's difficult to make any blanket statements about how much slowdown there will be without knowing the kind of code and duration models. The speed will depend partly on the level of the primitives that the code uses. For example, if the code is "low-level" code that does a lot of operations like `car` and `cdr`, each of those primitives now has an associated increment of the clock. For such simple functions, incrementing the clock is a significant slowdown. On the other hand, a function like `sort` is barely slowed down by measuring the size of its input (n) and incrementing the clock by $cn \log n$, where c is the duration model coefficient. If the user defines cognitive primitives at a higher level, the overhead may be even less. In addition, unfortunately, the speed also depends on the quality of the Lisp compiler—a good compiler can open-code much of the incrementing code using type-specific arithmetic instructions.

We can, however, take timings of standard benchmark programs, to get an idea of how much TCL slows the code down. The data in table 1 were collected using Gabriel's benchmarks [4], which are available by anonymous FTP from the CMU AI archives or by contacting us. The first two columns are raw timings (that is, CPU seconds) for the benchmark programs running normally in Harlequin Lispworks on a DEC Alpha. Note the difference between the entries in the two columns: this is the variation that we want to be rid of by using TCL rather than CPU time to define the duration of thinking. The third column reports the raw times for ordinary TCL code, using primitive duration models incrementing a clock at run-time.[1]

The fourth column is just like the third, except that TCL used a code-walker (CW) to combine duration increments at compile-time. The code-walker looks for "basic blocks" of the program—a basic block is straight-line code, without loops or branches—and tries to compute the total duration of the block. This can be done for the `:constant` and `:arity` duration models, which are very common. For example, three consecutive computations with constant durations of 2, 3 and 5, can be coalesced at compile time to a single increment of 10 time units. Therefore, the times in the fourth column are less than or equal to the corresponding times in the third, repre-

[1]Note that all we needed to do to "port" the benchmark programs to TCL from CL was to load the programs into a different package. The algorithms across a row are exactly the same.

Table 1: Timing studies on a DEC Alpha running Harlequin Lispworks. Columns 1 and 2 are two runs of the benchmarks in ordinary Common Lisp. Columns 3 and 4 are runs of those benchmarks using two versions of TCL, with and without the code walker described in the text. Columns 5 and 6 are columns 3 and 4 divided by the mean of columns 1 and 2, and therefore report the relative slowdown of using TCL.

	raw CL		raw TCL		relative TCL	
	1st run	2nd run	w/o CW	w/ CW	w/o CW	w/ CW
boyer	1.182	1.142	3.033	2.515	2.6	2.2
browse	1.034	1.103	4.045	2.686	3.8	2.5
ctak	0.079	0.072	0.147	0.100	1.9	1.3
dderiv	0.288	0.322	0.359	0.307	1.2	1.0
deriv	0.338	0.335	0.416	0.347	1.2	1.0
destru-mod	0.118	0.120	0.303	0.140	2.5	1.2
destru	0.124	0.120	0.298	0.151	2.4	1.2
div2	0.273	0.308	0.597	0.339	2.1	1.2
fft-mod	1.413	1.326	1.871	1.382	1.4	1.0
fft	1.949	2.143	2.522	1.980	1.2	1.0
frpoly	2.626	2.562	4.050	3.283	1.6	1.3
puzzle-mod	1.474	1.076	1.876	1.357	1.5	1.1
puzzle	1.195	1.089	1.845	1.307	1.6	1.1
stak	0.113	0.112	0.184	0.146	1.6	1.3
tak-mod	0.183	0.186	0.670	0.430	3.6	2.3
tak	0.184	0.188	0.597	0.454	3.2	2.4
takl	0.106	0.093	0.383	0.260	3.8	2.6
takr	0.090	0.092	0.214	0.138	2.4	1.5
traverse	3.674	3.644	8.213	6.683	2.2	1.8
triang-mod	43.718	36.020	53.170	39.892	1.3	1.0
triang	15.741	16.043	33.002	19.452	2.1	1.2

senting the savings due to compile-time code-walking.

The third pair of columns is the speed of TCL, with and without the code-walking, relative to the mean CL time. For example, the 2.2 in the upper right of the table is the speed of TCL with the code-walker (2.515) divided by the mean of 1.182 and 1.142. This data is from just six runs, and, because of the variance in measuring CPU times, we would have to collect much more data to get very precise estimates. Nevertheless, we can conclude that (1) variance in measuring CPU time is indeed a problem, and (2) the cost of using TCL appears to be 20 to 120 percent, depending on the benchmark program. Naturally, we hope that realistic AI programs will tend more towards the lower end of the range.

5 MESS

We have re-implemented PHOENIX using a simulation substrate called MESS (Multiple Event Stream Simulator), of which TCL is a part. We describe MESS as a simulation substrate because it makes no domain commitment. Instead, it works with abstractions called "events," "event streams" and "activities," among others. One builds a simulation environment in MESS by defining the events that happen, thereby changing the state of the world, and defining the event streams that produce those events. The MESS substrate takes care of synchronizing all the events so that the simulation unfolds in the correct way, with processes interacting as they should.

The implementer of a simulation uses the built-in event classes and event streams of MESS by inheriting and defining methods for CLOS (Common Lisp Object System) generic functions. For example, the `realize` method implements the semantics of the occurrence of an event. A user typically must define that method for events that are unique to the new simulation. Another example is the pop method of event streams, which cause them to yield a new event upon demand; the user can extend or modify the implementation of event streams by defining that method on a new event-stream class.

One kind of event stream is a *thinking* event stream, which are used to implement deliberative agents. It is an event stream in which the interval between events is determined by the amount of thinking done by the agent. TCL allows us to define thinking event streams that have deterministic behavior. MESS allows deliberative agents to be smoothly integrated into a discrete event simulation, with the agent events (sensor and effector actions) to be correctly interleaved with other events, even those from other agents.

Another feature of MESS defines *activities*, which occur over an interval of time, and allows these activities to be *interrupted* by events that occur during that time interval. For example, an agent's movement can be modeled as an activity, which might get interrupted by changes in the environment. An agent's thoughts can also be modeled as an interruptible activity. This feature is particularly interesting to deliberation scheduling and anytime algorithm researchers, because the thinking can be interrupted by, for example, a timer going off or an event in the world that indicates a change in the value of further deliberation.

6 Conclusion

Using the Timed Common Lisp language frees a researcher from worrying about noise in measuring CPU time, from worrying that a new release of the Lisp compiler will cause a change in an agent's behavior or an algorithm's time/quality curves, from worrying that the system's behavior on an DEC Alpha won't be the same as on a SUN SPARCstation, and from worrying that adding instrumentation code to collect statistics will change the behavior of the simulation. This noise may not even be very great; PHOENIX's variance isn't very much, but it is noticeable, and it is enough to prohibit experiments in which we replicate particular simulation states. Therefore, TCL gives us a significant advantage over a CPU-time approach. Furthermore, the TCL approach allows us to declaratively represent duration using high-level primitives, so that deliberation scheduling becomes easier and allows us more control over time/quality tradeoffs.

Because TCL is integrated with MESS, it can be easily used to build deterministic simulation of multiple agents acting under time pressure in a complex environment.

Acknowledgments

This work is supported by ARPA/Rome Laboratory under contract F30602-93-C-0100 and by NTT Data Communications Systems Corporation. The U. S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notice contained hereon.

References

[1] Scott D. Anderson. *A Simulation Substrate for Real-Time Planning.* PhD thesis, University of Massachusetts at Amherst, February 1995. Also available as Computer Science Department Technical Report 95–80.

[2] Paul R. Cohen, Michael L. Greenberg, David M. Hart, and Adele E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3):32–48, Fall 1989.

[3] Richard Cooper, John Fox, Jonathan Farringdon, and Tim Shallice. Towards a systematic methodology for cognitive modeling. Technical Report UCL-PSY-ADREM-TR5, University College London, November 1992.

[4] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985.