Real-Time Programming With Time-Stamped Event Histories[*]
Technical Report: UW-CSE-96-05-02

Alan Shaw and Douglas Rupp

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350
{shaw,drupp}@cs.washington.edu

Abstract

Assertions on time-stamped event histories (TEHs) have been used in real-time theory for specifying, analyzing, and verifying requirements and designs, and also have been proposed for monitoring implementations. This paper investigates the direct use of TEHs for programming. We define some language and system support mechanisms, show by example the potential benefits of programming with TEHs, and describe an implementation that is an extension of a state-based system. Benefits include cleaner and more readable code, and programs that more closely resemble specifications.

1 Introduction

The behavior of a real-time system can be characterized by a set of traces or histories of its interesting events over time. Each such time-stamped event history (TEH) may be represented by an event sequence:

$$H = < X_1, X_2, \ldots, X_i, \ldots > .$$

The ith event, X_i, is a triple (E_i, V_i, T_i) where E_i is the name of the type or class of the event, V_i is the data, if any, associated with the event, and T_i is the time of the event. Such traces have been used for the assertional specification, reasoning, verifying, and monitoring of real-time systems, mainly because many required and desired properties can be expressed conveniently in terms of TEHs. However, there has been little or no language and system support for dealing directly with TEHs in application programs.

The purpose of our work is to study the use of TEHs for writing real-time programs. Our proposal is to provide support for accessing TEHs in programs, *in addition to* normal language and system facilities; we are not suggesting that traces be used as a universal data object to the exclusion of all others. Our contributions are to demonstrate by examples that there are real benefits in

[*] This research is supported in part by the Army Research Office under grant number DAAH04-94-G-0226 and by the National Science Foundation under grant number CCR-9200858. The work was also supported in part by ENST (Telecom), Paris, when the first author was on sabbatical during Winter and Spring 1995.

programming with TEHs, to define some basic associated programming mechanisms, and to report on some software tools that implement these ideas.

In the next section, we survey some standard applications of TEHs in other areas. Section 3 presents our programming and event notation. Several examples comparing conventional programs with ones using TEHs are discussed in Section 4. The subsequent section describes our supporting software tools and initial experiments. We then describe in Section 6 some open research problems and issues connected with this new approach.

2 Applications of TEHs

For many years TEHs, or related objects such as those that also include states in the traces, have been employed in real-time theory for specifying, analyzing, and verifying requirements and designs. Examples include logics such as Real-Time Logic (RTL) [Jahanian&Mok86] that deal directly with timed events and histories, and also state and programming models such as Timed IO Automata (i.e., [Attiya&Lynch89]) and Timed CSP [Reed&Roscoe86], respectively, where traces are the underlying behavioral descriptions.

A more recent application has been the monitoring or checking of real-time systems [Chodrow_et_al91, Jahanian_et_al94]. In these works, timing constraints expressed in a language based on RTL are checked at run-time by a separate monitoring component of the software. We have used similar ideas for monitoring executable specifications in our Communicating Real-Time State Machine (CRSM) notation [Raju&Shaw94]. Here, event-triggered timing specifications expressed in an RTL variant are checked during simulation runs; the simulation and specification code are each a separate component as in the other works discussed in this paragraph. These efforts within the RTL framework have significantly influenced the research reported in this paper.

We have also been influenced by a new programming model, called time-sensitive objects (TSOs) [Callison95], that develops a data-centered approach to specifying and building real-time systems. The basic idea is to maintain histories of the values of data objects over time; operations are defined for updating and accessing these histories. The TSO model is proposed as an alternative to the conventional process model, for example, time-sensitive periodic objects replacing periodic processes.

Another application area that uses TEH-like objects in an interesting way is parallel debugging systems [McDowell&Helmbold89]. Many of these systems record event histories, sometimes with time-stamps, as the principal debugging data. Parallel discrete event simulation languages also use time-stamped event lists in a central way [Fujimoto90]. One example is the Rapide language which provides an event pattern language for defining triggers for processes [Luckham&Vera95]; the event patterns describe partially ordered sets of events.

TEHs are *the* objects of convenience or necessity in these other theoretical and practical applications. However, we know of no language or system that supports their *direct* use for programming real-time systems.

3 Event and Programming Notation

As described in the introduction, each event $X_i = (E_i, V_i, T_i)$ is a (*name*, *value*, *time*) triple, for example:

$$X_5 = (\textit{Temperature}, 15^{o}\ \text{C}, 10{:}00\ \text{AM}) .$$

The history is ordered by time, so that $T_i \le T_{i+1}$ for all i. Several events from different classes could occur simultaneously. For events of the same class or name, we require that they be strictly separated in time; e.g., if $E_i = E_j$ and $i \ne j$, then $T_i \ne T_j$.

A time-stamped history that is stored during the execution of a program is of finite length and only defined up to the most recent event that has occurred. If n events have occurred, then the stored history is:

$$H = < X_1, \ldots, X_n > ,$$

where X_n is the most recent event. It is most convenient to reference these events starting from the latest and working backwards. Following earlier works, including our own, we will use negative indices:

$$H = < X_{-n}, \ldots, X_{-1} > ,$$

where for positive i, we have $X_i = X_{i-n-1}$. That is, the most recent event is X_{-1}, the second most recent is X_{-2}, and the first event is X_{-n}.

The identification of the most recent k events is given by the vector form $E() = (C_{-k}, \ldots, C_{-1})$ which means $E_{-i} = C_{-i}$, for i = 1 to k. For example, $E()$ = (*down*, *up*, *timeout*, *down*) is a short form for E_{-4}=*down*, E_{-3}=*up*, E_{-2}=*timeout*, and E_{-1}=*down*. The symbol "~" preceding an event name means that the event has not occurred at the indicated position. For example, $E()$ = (~*Temperature*, *Pressure*) is an abbreviation for $E_{-2} \ne$ *Temperature* and E_{-1}=*Pressure*. Finally, "•" is used as a "don't care" or "wild card" indicator, such as in $E()$ = (a, •, b) which specifies that E_{-2} could be any event.

A TEH H contains all of the events that have occurred. Most often, we will be interested in only a subset of the events in each part of a program. This is accomplished by defining *projections* on H that either include or eliminate designated event classes. The notation H | *expr* will denote the projection or restriction on H as designated by the expression *expr*. In its simplest forms, *expr* could be an event name C or it's "negation" ~C. H | C and H | ~C mean, respectively, to include only the events from C in the history and to exclude the events from C. For example, H | *down* restricts the history to the *down* event class only. We also permit lists in the projections, so that H | (C_1, ... , C_k) restricts the history to only events C_1, ... , C_k , and H | ~(C_1, ... , C_k) excludes these events.

The program examples will use a conventional imperative notation with similarities to Ada and CSP/Occam. This is essentially a textual version of our graphical CRSM language [Raju&Shaw94]. Guarded commands and non-determinacy are used where appropriate. A guarded command is of the form *guard* → *command*. The symbol "[]" is used to separate guarded commands or lists of guarded commands, that are candidates for non-deterministic selection.

All events in our programs, including timeouts (see below), are input-output (IO) communication events following the synchronous CSP model. To send a message *expr* on a channel *C*, a process issues the command:

C(*expr*)! .

A receiving process on the same channel will give a command:

C(*x*)? .

If input-output occurs, it happens simultaneously. The effect at the receiver is the assignment:

x := *expr* .

Both sender and receiver then proceed. The corresponding IO event is the triple: (*C, expr, T*) , where *T* is the time IO occurred.

If events have no message fields, the parentheses are omitted. A possible timeout or delay is described by the syntax:

Timer? [*t*] ,

where *t* is the delay interval. This is viewed as an input from a clock process on the *Timer* channel.

4 Example Programs With Comparisons

The purpose is to illustrate programming solutions to several real-time problems, comparing those in a conventional programming language with ones that support TEHs. We choose particularly simple problems, but with realistic features, so that we can expose some of the potential benefits of TEHs.

Example 1: Recognizing Single Clicks from a Mouse Button

Consider a mouse input device with a single button. The computer can detect two events from the button -- a *down* event, named *D*, corresponding to pressing the button and an *up* event *U* that is triggered when the button is released. A single click, denoted *SC*, is recognized if the time between a *U* and its preceding *D* is less than a given interval *tsc*. A single click recognizer will send an *SC* message to a handler whenever this constraint is met.

A solution in a standard programming notation, with title subscripted by *S* for “standard”, is:

```
Single_ClickerS::
    loop
        D?                              // Wait for input D.
        {
            U?   SC!                    // Wait for U and then output SC;
            [] Timer?[tsc]   U?         // or timeout then wait for U.
        }
    end loop
```

The "[]" indicates a selection between the *U* input statement before and the *Timer* input statement after it, based on which occurs first. If a *U* occurs before the *Timer* IO occurs at time *tsc*, then the *SC* signal is generated; otherwise, the program waits for *U* and returns to the top of the loop.

A TEH version, with title subscripted by *H* for "history", is:

```
Single_ClickerH::
        loop
            D?   U?
            if ((T-1 – T-2) < tsc) then SC!
        end loop
```

The program history contains the event classes *D*, *U*, and *SC*. No projections are required in the solution. Now compare *Single_Clicker*$_S$ and *Single_Clicker*$_H$. The TEH version has a direct and straightforward specification of the *SC* property and very simple control structures inside the loop. In the first program, it's not as obvious that the *SC* is generated correctly; a timeout is necessary within a non-deterministic statement -- arguably more complex. (Of course, the TEH program also incurs a cost in storing and accessing histories.)

Example 2: Part of a Traffic Light Controller

An intersection of a street and an avenue has two pairs of traffic lights that cycle through the normal red-green-yellow sequence. If an ambulance approaches the intersection, all lights are turned red, and the street or avenue direction of the ambulance is stored. A conventional state-based program containing the ambulance processing appears in [Raju&Shaw94]. The traffic light events *sr*, *sg*, *sy*, *ar*, *ag*, and *a*y are associated with commands that turn the street and avenue lights red, green, and yellow, respectively. An ambulance sensing event with a *direction* parameter, called *amb_approaching*(), signifies that an ambulance is approaching the intersection. The code with TEH for the ambulance detection and processing, i.e., ensuring that all lights are red, is:

```
AmbulanceH::
        using H | (ag, ay, ar, sg, sy, sr)
        amb_approaching(direction)?
        case E-1 of
            ag:     ay!  Timer?[5]  ar!
            ay:     Timer?[5]  ar!
            sg:     sy!  Timer?[5]  sr!
            sy:     Timer?[5]  sr!
            ar, sr: null
        end case
```

The initial projection in the *using* statement restricts the history to only traffic light events. The *case* construct over the last event generates appropriate sequences to turn the street or avenue lights red. For example, if the last event was *sg* thus turning the street lights green, IO commands

are issued to turn the street light yellow (*sy*!), wait for 5 time units (*Timer*?[5]), and turn the street lights red (*sr*!). The rest of the system is constructed so that, under normal operation, both lights are red after any turn-red event (*ar* and *sr*).

Conventional code for this task also requires knowledge of the state of the traffic lights when the *amb_approaching* event occurs. $Ambulance_H$ obtains this state directly from E_{-1}. Without TEH support, one could distribute pieces of the above code through the traffic controller, as is done in [Raju&Shaw94], encode the state through program variables, essentially storing the history, or allow references to actual labeled states such as provided by the statechart specification language [Harel87]. In the latter, a guard on a state transition could refer explicitly to the state of some other component. Generally, access to TEHs allows a programmer to recreate relevant state.

Example 3: General Mouse Clicker Recognizer

Example 1 is generalized here to recognize single clicks, double clicks, and selections from a mouse button. Let a single click (*SC*) be defined as above. A double click (*DC*) is two consecutive single clicks separated by a time interval less than a given *tdc*. Single click signals are not generated if the single click is part of a double click. If the separation between a *D* and the following *U* is greater than or equal to *tsc*, then the *D* corresponds to a selection start (*SS*) and the *U* to a selection end (*SE*); *SS* and *SE* events are generated. A state machine solution to this problem is presented in [Shaw92].

A conventional program implementing the recognizer is:

```
Click_RecognizerS::
    loop
        D?
        {
            Timer?[tsc]  SS!  U?  SE!                    // Timeout after first D
            [] U?                                        // D U has been input
              {
                  Timer?[tdc]  SC!                       // Timeout after D U
                  [] D?                                  // D U D recognized
                    {
                        U?   DC!                         // D U D U means DC
                        [] Timer?[tsc]  SC!  SS!  U?  SE!  // see below *.
                    }
              }
        }
    end loop
```

A graphical state machine representation is easier to follow, because it does not have the deep nesting of the above text. The most complex case is in the innermost clause (*) where the previous

three button inputs, *D U D*, have indicated a possible double click, but the last *U* does not occur in time; in this case an *SC* is sent for the first *D U* pair and then a selection is generated.

A TEH version follows below. In it, we request alarm clock timers *ACsc* and AC*dc* to send a wakeup signal at some future time *t* relative to its call, using the output command:

ACi.Wake_Me(*t*)! ,

where *i* is either *sc* or *dc*. A wakeup signal, *ACi.Wake_Up*, is generated by the alarm clocks after the interval expires. The alarm clocks also respond to *Reset* signals that terminate any outstanding *Wake_Me* requests.

```
Click_RecognizerH::
        using H | ~(Wake_Me, Reset)
        loop
            {
                D?  ACsc.Wake_Me(tsc)!  ACdc.Reset!
                [] U?  ACdc.Wake_Me(tdc)!  ACsc.Reset!
                [] ACsc.Wake_Up?
                [] ACdc.Wake_Up?
            }
            case E() of
                (D, U, D, U):                   DC!
                (D, U, ACdc.Wake_Up):           SC!
                (~D, ~U, D, ACsc.Wake_Up):      SS!  U?  SE!
                (D, U, D, ACsc.Wake_Up):        SC!  SS!  U?  SE!
                others:                         null
            end case
        end loop
```

The projection eliminates the alarm clocks' *Wake_Me* and *Reset* events from consideration. This example also illustrates the use of the vector notation, *E*(), for event lists. The nesting complexity has been replaced by a simpler, but still somewhat difficult, event accepting part that involves generating and resetting timing signals. The *case* statement separates the possible outputs into four clear classes that are not too far removed from a formal specification. In order of appearance, these are: the double click, the simple single click, the selection, and the more complicated single click followed by a selection. For example, if *E*() = (*D, U, D, U*) then a *DC* message is output.

Example 4: Part of a Coolant Controller

This is a toy example in process control taken from [Jaffe_et_al91], that shows the use of all components of the event triple as well as some history. The coolant temperature of a reactor tank is controlled to maintain a temperature of C^{o} Celsius by moving two rods. Whenever the temperature changes by c^{o} Celsius, a sensor sends the temperature to the controller which checks on the validity of the data; the check also ensures that the readings don't arrive too quickly. Let the temperature

sensing event be (*Temperature*, *degrees*, *time*) over the IO channel *Temperature*(). The validity checking and rod moving part of the controller can be coded using TEHs as follows:

```
Coolant_Controller_H::
    using H | Temperature
    loop
            . . .
        Temperature(d)?
        valid := (d > –273) and (d ≤ 500) and (abs(V-1 – V-2) = c)
                and ((T-1 – T-2) ≥ tmin)
        {
            ((d < C) and valid) →       Move_Rod(up)!
            [] ((d ≥ C) and valid) →    Move_Rod(down)!
            [] ¬ valid →                Error!
        }
            . . .
    end loop
```

The projection restricts the history accesses to *Temperature* events only. Note that $V_{-1}=d$ in the program. Without histories, one would typically declare and use variables such as *old_temp*, *new_temp*, *old_time*, and *new_time*, in order to compute the value of *valid*. It would also require the usual amount of error-prone switching of values each time through the loop, for example:

old_time := *new_time* .

5 Supporting Software Tools

In order to permit some early experimentation, we have extended our communicating real-time state machine (CRSM) software [Raju&Shaw94] with TEH support. The resulting system, called CRSM+H, consists of a graphical editor and a simulator. The editor permits the creation of general state machines, where the transitions are guarded commands with time durations. A command can be a CSP-like send or receive, or any C++ program. In addition, there is a novel set of timing facilities that allow the specification of timing durations for transitions and provide access to clocks. The TEH extensions implement mechanisms for storing and accessing event histories as discussed below. Some of the TEH features and a small part of the code was borrowed from the assertion checker component of the CRSM tools. The simulator executes a system of machines over simulated time.

Each IO communication is an event. Global history is stored in an array of event records, so that an event $X(i)$ is the record $(X(i).E, X(i).V, X(i).T)$. For $i \geq 0$, the reference is to the ith instance; if $i < 0$, the reference is to the ith most recent instance as defined earlier. A finite size window of the q most recent events on all IO channels is maintained in the array X. In our experiments to date, a q of size 20 has been adequate. There is also an implementation of the vector $E()$, defined in Section 3, as a Boolean function $E(C_{-k}, \ldots , C_{-1})$, which returns True if $E_{-i} = C_{-i}$, for $i = 1$ to k, and False

otherwise. The projection $H \mid (C_1, \ldots, C_k)$ and the elimination projection $H \mid \sim(C_1, \ldots, C_k)$ are supported by global initialization functions *H_Use* $(C_1, \ldots, C_k)$ and *H_Not* $(C_1, \ldots, C_k)$, respectively. $T(i)$ and $V(i)$ return the time and value of event i on channels in the modified or projected event history.

Several examples have been run, including the general click recognizer of Example 3 in the last section ($Click\_Recognizer_H$). The sensor and integrator tasks presented as Ada programs in [Corbett94] were also translated into both pure CRSM and CRSM+H machines and then simulated. The TEH (CRSM+H) versions of the sensor and integrator seemed simpler than both the Ada and pure CRSM versions, requiring fewer timeouts and following the specifications in a more straightforward fashion.

6 Research and Issues

This work started in 1995 and is still in its early stages. However, it seems clear from our computer and paper experiments that TEHs offer some real benefits for programming real-time applications. What is not clear yet though is how to recognize those algorithms that can be beneficially reformulated and how to do so. Our examples in Section 4 required a surprising amount of experimentation before we achieved satisfactory results, perhaps because the ideas were new to us. Part of this is discovering or inventing an appropriate programming style. One approach to this issue that we are pursuing is to develop solutions to common problems in a variety of different event-based real-time languages, such as Ada95, Esterel, statecharts, and CRSMs, and compare these with solutions with TEHs. In fact, the idea for TEHs came to the first author while comparing programs in the above languages for the general click recognizer problem.

There are a number of open issues concerning programming and systems support for histories. Various projections on histories are needed, but it's not yet evident whether the simple restrictions given here are sufficient. For example, one could imagine something similar to regular expressions (REs) or REs plus negation for defining the event classes of interest. However some care must be taken to ensure that the complexity of computing the projections remains reasonable.

A related problem is the "scope" of histories and events. As presented, histories are global; it may be desirable to have histories also at various local levels without resorting to projections, for example, because of clock errors (see below). Both the experiments with assertion checking and those reported here use a small window covering the most recent events; there may be useful applications that require larger windows or that require some of the initial events in a history. In any case, we have not investigated in detail either the performance or predictabilty of the functions for updating and accessing histories.

Time-stamped events are potentially very general objects. We have concentrated on IO events only and within a particular communication model. One might wish to declare new event classes, and to selectively enable, disable, generate, and reset events. Finally, an accurate time-stamp based on a shared global time has been implicitly assumed. This assumption is often not easily met, especially for distributed systems, since it requires that clocks be synchronized.

Our software tools need to be developed further, perhaps moving in the direction of a textual implementation. One appealing approach is to take a language with object capabilities, such as Ada95, and construct classes that provide TEH mechanisms.

7 Summary and Conclusions

Time-stamped event histories have been used in many areas of computing, but not directly in programming. Our goals are to study their application for real-time programming, and to define appropriate language and system support. We have outlined a programming and event notation for accessing TEHs, shown by example their potential benefits, and described an implementation which is an extension of a state-based simulation. Current and future work includes refining and extending the notation, support and applications, as well as developing a programming style for TEHs. For those programs and parts of programs that are applicable, the potential benefits include cleaner and more readable code, and code that more closely resembles the required specifications.

Acknowledgments

We are grateful to Ken Hines and Ross Ortega for comments on an earlier draft.

References

[Attiya&Lynch89] H.Attiya and N.Lynch, "Time bounds for real-time process control in the presence of uncertainty," *Proc. IEEE Real-Time Systems Symp.,* Dec. 1989, pp.268-284.

[Callison95] H.Callison, "A time-sensitive object model for real-time systems," *ACM Trans. on Software Engineering and Methodology*, vol.4, no.3 (July 1995), pp.287-317.

[Chodrow_et_al91] S.Chodrow, F.Jahanian, and M.Donner, "Run-time monitoring of real-time systems," *Proc. IEEE Real-Time Systems Symp.*, Dec.1991, pp.74-83.

[Corbett94] J.Corbett, "Modeling and analysis of real-time Ada tasking programs," *Proc. IEEE Real-Time Systems Symp.*, Dec.1994, pp.132-141.

[Fujimoto90] R.Fujimoto, "Parallel discrete event simulation," *Comm. of ACM*, vol.33, no.10 (Oct.1990), pp.30-53.

[Harel87] D.Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol.8, (1987), pp.231-274.

[Jaffe_et_al91] M.Jaffe, N.Leveson, M.Heimdahl, and B.Melhart, "Software requirements analysis for real-time process control systems," *IEEE Trans. on Software Engineering*, vol.17, no.3 (March 1991), pp.241-258.

[Jahanian&Mok86] F.Jahanian and A.Mok, "Safety analysis of timing properties in real-time systems," *IEEE Trans. on Software Engineering* , vol.SE-12, no.9 (Sept. 1986), pp.890-904.

[Jahanian_et_al94] F.Jahanian, R.Rajkumar, and S.Raju, "Runtime monitoring of timing constraints in distributed real-time systems," *Real-Time Systems Journal*, vol.7, no.3, 1994, pp.247-273.

[Luckham&Vera95] D.Luckham and J.Vera, "An event-based architecture definition language," *IEEE Trans. on Software Engineering*, vol.21, no.9 (Sept.1995), pp.717-734.

[McDowell&Helmbold89] C.McDowell and D.Helmbold, "Debugging concurrent programs," *ACM Computing Surveys*, vol.21, no.4 (Dec. 1989), pp.593-612.

[Raju&Shaw94] S.C.V.Raju and A.Shaw, "A prototyping environment for specifying, executing, and checking communicating real-time state machines," *Software-Practice and Experience* , vol.24, no.2 (Feb. 1994), pp.175-195.

[Reed&Roscoe86] G.Reed and A.Roscoe, "A timed model for communicating sequential processes," *Proc. ICALP '86*, Springer-Verlag LCNS 226, 1986, pp.314-323.

[Shaw92] A.Shaw, "Communicating real-time state machines," *IEEE Trans. on Software Engineering*, vol.18, no.9 (Sept. 1992), pp.805-816.