In Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems, Orlando, FL, June 1994.

CLEOPATRA

Physically-Correct Specifications of Embedded Real-Time Programs

Azer Bestavros*

Computer Science Department
Boston University
Boston, MA 02215

1 Introduction

Predictability – the ability to foretell that an implementation will not violate a set of specified requirements – is a crucial, highly desirable property of responsive embedded systems. This paper overviews a development methodology for responsive systems, which enhances predictability by eliminating potential hazards resulting from physically-unsound specifications.

The backbone of our methodology is the Time-constrained Reactive Automaton (TRA) formalism [3, 4], which adopts a fundamental notion of space and time that restricts expressiveness in a way that allows the specification of only reactive, spontaneous, and causal computation. Using the TRA model, unrealistic systems – possessing properties such as clairvoyance, caprice, infinite capacity, or perfect timing – cannot even be specified. We argue that this "ounce of prevention" at the specification level is likely to spare a lot of time and energy in the development cycle of responsive systems – not to mention the elimination of potential hazards that would have gone, otherwise, unnoticed.

The TRA model is presented to system developers through *CLEOPATRA*.[1] The *CLEOPATRA* language features a C-like imperative syntax for the description of computation, which makes it easier to incorporate in applications already using C. It is event-driven, and thus appropriate for embedded process control applications. *CLEOPATRA* is object-oriented and compositional, thus advocating modularity and reusability. The physical correctness of the TRA formalism complemented with a strong typing system prevents incorrect or unrealistic specifications from being expressed. This saves system developers from having to debug many specification errors during the implementation stage. *CLEOPATRA* is semantically sound; its objects can be transformed, mechanically and unambiguously, into formal automata for verification purposes. Since 1989, an ancestor of *CLEOPATRA* has been in use as a specification and simulation language for embedded time-critical robotic processes. [2, 5, 1].[2]

2 *CLEOPATRA*

In *CLEOPATRA*, systems are specified as interconnections of TRA objects. Each TRA object has a set of *state variables* and a set of *channels*. Any *event* in the system is represented by an *action* (value) signaled on a given channel. The occurence of an event may trigger a change of state. Time-constrained causal relationships between events occuring on different channels, and the computations (state transitions) that they trigger, are specified using *Time-constrained Event-driven Transactions* (TETs). The behavior of a TRA object is described using TETs. TRA objects can be composed together to specify more complex TRAs.

*This research is supported by NSF/CCR-9308344.

[1]A *C*-based *L*anguage for the *E*vent-driven *O*bject-oriented *P*rototyping of *A*synchronous *T*ime-constrained *R*eactive *A*utomata.

[2]A compiler that allows the execution of *CLEOPATRA* specifications has been developed [6], and is available via FTP from `cs.bu.edu:/bestavros/cleopatra/`.

2.1 Classes and Objects

A TRA object specification in *CLEOPATRA* consists of two components: a header and a body. A TRA object header specifies its name, the parameters needed for its instantiation, and its signature (type). An object's body specifies its behavior.

In *CLEOPATRA*, TRA objects are defined in *classes*. For example, Figure 1 shows the *CLEOPATRA* specification of the class of integrators that use trapezoidal approximation.

```
TRA-class integrate(double TICK, TICK_ERROR)
     in(double) -> out(double)
{
 state:
  double x0 = 0, x1 = 0, y = 0;
 act:
  in(x1) -> :
    ;
  init(),out() -> out(y):
    within [TICK-TICK_ERROR~TICK+TICK_ERROR]
      commit { y = y+TICK*(x0+x1)/2; x0 = x1; }
}
```

Figure 1: Class of trapezoidal integrators.

In the specification of `integrate` given in Figure 2, the class parameters `TICK` and `TICK_ERROR` have to be specified before *instantiating* an object from that class. Furthermore, the header of `integrate` specifies that objects of that class have a signature consisting of an input channel `in` and an output channel `out`. Both `in` and `out` carry actions whose values are drawn from the set of reals. In *CLEOPATRA*, the start channel of any given TRA-class is called `init`. Upon the instantiation of a given TRA object, an event on `init` is generated by the system. Start channels do not have to be explicitly included in the header of a TRA-class. For example, in the definition of the `integrate` TRA-class given in Figure 1, there is no mention of any `init` channels in the external signature specified in the header, yet, `init` is used later in the body of `integrate`.

The body of a TRA class determines the behavior of objects from that class. Such a behavior can be either *basic* or *composite*. The description of a basic behavior involves the specification of a state space in the `state:` section, the specification of an initialization of that space in the `init:` section, and the specification of a set of TETs in the `act:` section. The behavior of an object belonging to the TRA-class `integrate` shown in Figure 1 is an example of a basic behavior. Composite behaviors, on the other hand, are specified by composing previously defined, simpler TRA-classes together in the `include:` section. For example, in Figure 2, the class `ramp` is defined by composing the `integrate` and `constant` classes together.

```
TRA-class ramp() -> y(double)
{
 internal:
  x(double) -> ;
 include:
  constant -> x() ;
  integrate x() -> y() ;
}
```

Figure 2: A ramp generator in *CLEOPATRA*

2.2 TETs

A TET describes the reaction of a TRA to a subset of events. Such a reaction may involve responding to triggers and/or firing action(s). Figure 3 explains the relation between the triggering and firing of actions using TETs.

The description of a TET consists of two parts: a header and a body. The header of a TET specifies a set of triggering channels (trigger section) and a controlled channel (fire section). The trigger section specifies the effect of the triggering actions on the state of the TRA. A TET with no triggering section is triggered every time an action is signaled on any channel of the TRA. The fire section specifies the action value to be signaled on the controlled channel as a result of firing the TET. This value can be any expression on the state of the TRA. An absent expression means that a random value from the signaling range of the controlled channel is to be

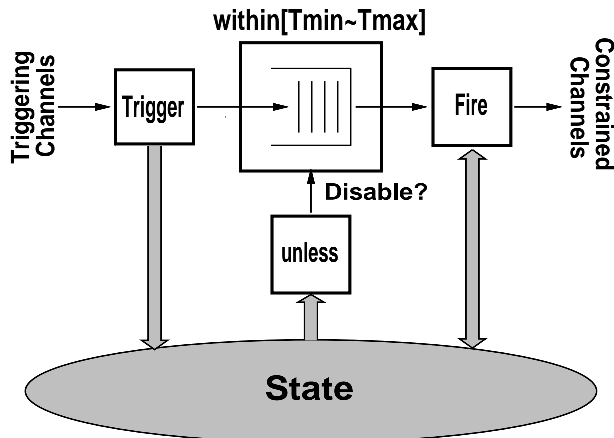

Figure 3: The TET structure

signaled. The body of a TET describes possible reactions to the TET triggers. Each reaction is associated with a disabling condition, a time constraint, and a state transformation schema.

For example, the first TET of the `integrate` class shown in Figure 1 is an example of a transaction with only a trigger section. Every time an action is signaled on the input channel `in`, its value is stored in the state variable `x1`, thus, resulting in a potential input transition. The second TET of the `integrate` class, on the other hand, is an example of a transaction with both a trigger section and a fire section. In particular, every time an action is signaled on one of the triggering channels (`init` or `out`) an output action is fired on `out` after a delay of `TICK` $\pm$ `TICK_ERROR` units of time elapses.

Each reaction in the body of a TET is associated with three pieces of information: A disabling condition, a time constraint, and a state transformation schema. The *disabling condition* (unless clause) is a predicate on the state of the TRA. In order to be committed, a reaction's disabling condition has to remain `false` from when the reaction is triggered until it commits. Thus, an intended reaction is aborted if the disabling condition becomes `true` at any point in time after the reaction is scheduled. The absence of a disabling condition in a reaction implies that, once scheduled, it cannot be disabled. The *time constraint* (within clause), determines a lower and upper bound for the real-time delay between scheduling a reaction and committing it. In accordance with the physically-correct nature of the underlying TRA formalism, this delay must be greater than zero. The absence of a time constraint from a TET specification implies that the causal relationship between the trigger and its effect is unconstrained in time. A lower bound of 0 and an upper bound of ∞ is assumed in such cases. The *state transformation schema* (commit clause) specifies a *method* for computing the next state of the TRA once a reaction is committed. This computation is assumed to be atomic and instantaneous. An absent commit clause implies that committing the reaction does not cause any state changes.

2.3 An Example

Figure 4 shows the specification of a finite FIFO element in *CLEOPATRA*. Values fed into the FIFO element are delayed for some amount of time before being produced as outputs.

```
TRA-class fifo(int N)
   in(float) -> out(float), overflow(), ack()
{
 state:
  float y[N];
  int i, j;
  bool f;
 act:
  init() -> ack():
   before DLY_MIN
    commit { i = 0; j = 0; f = FALSE;}
  in(y[i]) -> ack():
   before DLY_MIN
    commit { i = (i+1)%N ; if (i==j) f = TRUE ;}
  in() -> out(y[j]):
   unless (f)
    within [DLY_MIN~DLY_MAX]
     commit { j = (j+1)%N ; }
  in() -> overflow():
   unless (!f)
    within [DLY_MIN~DLY_MAX]
     ;
}
```

Figure 4: A finite FIFO delay element.

The header of the `fifo` TRA-class identifies the channel `in` as input, and the channels `out`, `ack` and `overflow` as outputs. The signaling range for channels `in` and `out` is the set of floating point numbers, whereas the signaling range for channels `ack` and `overflow` consists of only one value. The body of the `fifo` TRA-class contains two sections. In the `state:` section, the state space of a `fifo` object is described by four state variables: a vector `y[]` of `N` floating point values, two integer values `i` and `j`, and a boolean value `f`. In the `act:` section, the behavior of a `fifo` object is described by four TETs.

The first TET establishes a causal relationship between events signaled on `init` and those signaled on `ack`. Firing an action on `init` (the trigger) *causes* the firing of an action on `ack` (the result) after a delay of at most `DLY_MIN`. The second TET establishes a similar causal relationship between `in` and `ack`. The third TET establishes a causal relationship between events signaled on `in` and `out`. Firing an action action on `in` *causes* the firing of an action on `out` after a delay of at least `DLY_MIN` and at most `DLY_MAX`, provided that the FIFO did not overflow as of the last initialization. The fourth TET can be explained similarly.

Each TET specifies up to two possible state transitions. For example, the second TET of the FIFO in Figure 4 specifies that the value of a triggering event on `in` be stored in the state variable `y[i]`, thus resulting in a possible state change. In accordance with the TRA formalism's *input enabling property* [9], this transition cannot be blocked or delayed; it is an *input transition*. The second state transition induced by this TET occurs with the firing of an action on `ack`, resulting in the adjustment of the values of the state variables `i` and `f`. The value of the action signaled on a local (output or internal) channel does not reflect the state change associated with it. Thus, in the fourth TET, the value signaled on the `out` channel, namely `y[j]`, does not reflect the changes introduced in the `commit` clause, namely advancing the pointer `j`.

2.4 Case and Point!

It is important to realize that `fifo` objects will behave as expected only if inputs from the environment meet certain conditions. In particular, the value of the index `i` is not incremented as a result of an input on the channel `in` until at least `DLY_MIN` units of time elapse following the signaling of that input. It follows that an erroneous behavior will result if two or more events are signaled on the channel `in` in a duration of time shorter than `DLY_MIN`. To avoid such a malignant behavior, the environment must wait for an acknowledgment `ack()` or else, must wait for at least `DLY_MIN` before signaling a new input. Such correctness (safety) conditions can be verified using TRA-based verification techniques [4].

We argue that any finite implementation of a `fifo` object (discrete-event delay element) must have a *finite* capacity, which must not be exceeded for a correct behavior. Using *CLEOPATRA*, it is impossible to specify a `fifo` class that behaves correctly *independent* of its environment's behavior. This is a direct result of our abidance by the causality and spontaneity principles, which are preserved by the TRA model. As we mentioned at the outset of this paper, it is our thesis that preventing the specification of physically-impossible objects is desired. At the least is spares system developers from trying to implement the impossible.

2.5 *CLEOPATRA*-based Simulation

We have developed a compiler that transforms *CLEOPATRA* specifications into an event-driven simulator for validation purposes. We have used this system extensively in the specification and analysis of sensori-motor robotics applications [5] and in the behavioral simulation of complex autonomous creatures [2]. Figure 5 shows the different stages involved in the compilation and execution of specifications written in *CLEOPATRA*.

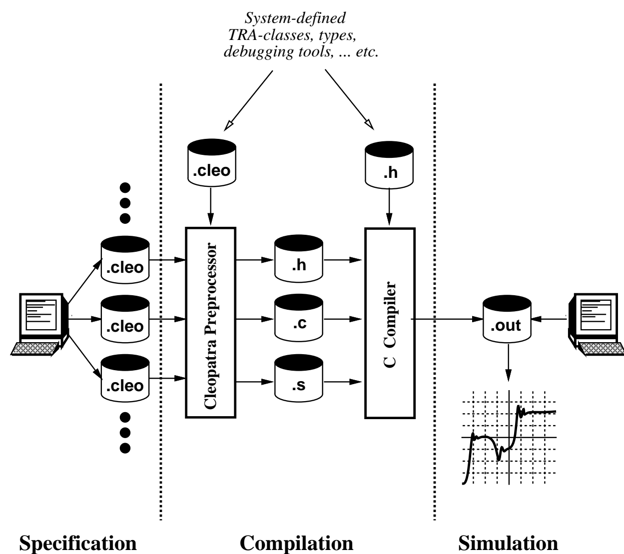

Figure 5: *CLEOPATRA* simulation environment.

At the heart of this process is a one-pass preprocessor, written in C, which parses user-defined *CLEOPATRA* specifications, augmented with system-defined TRA classes,[3] and generates an equivalent C simulator. This C simulator consists of three components. The first is a header (`.h`) file, which includes type definitions for the state space of the various TRA classes in the specification. The second is a schema (`.s`) file, which includes definitions for the state transition functions of the various TETs. The third is the code (`.c`) file, which includes the simulator initialization and control structure along with the instantiation code for the various TRA classes, including `main`. The final step of this process involves the invocation of the C compiler to produce an executable simulator.

A library of system-defined TRA-classes is available for debugging and performing I/O in *CLEOPATRA*. System-defined TRA-classes are themselves specified in *CLEOPATRA*. They are different from user-defined TRA-classes in that they have access to *global* information known only to the simulator. For instance, system-defined objects have access to the simulator's *perfect* clock, `_clk`, whereas user-defined TRA-classes have to maintain their own locally *perceived* clocks, if needed.

C functions can be called from within a *CLEOPATRA* specification. To maintain the semantics of the TRA formalism, however, only functions with no side effects should be used. In other words, C function should be restricted to act as pure operations on the state variables of an object. It should not reach beyond the boundaries of the state space of that object. Also, it should not alter the structure of the state space of the object in any way. Most of the C preprocessor utilities are available in *CLEOPATRA*. This includes simple and parameterized macro definition and invocation, constant definition, and nested file inclusion.

The simulator has proven to be quite efficient. This is due primarily to the causal and compositional nature of the TRA model, which tend to localize the computation triggered by the occurrence of an event within the boundaries of few TETs. The number of simulated events per second (seps) depends on a number of factors: the average channel fan-out, the average number of TETs per TRA, and the complexity of the event-driven computation. It does not depend, however, on the size of the state space or on the amount of TRA nesting. For an application with a fan-out of 1 and an average of 2.4 TETs per TRA, and an O(1) event-driven computational complexity, the compiled *CLEOPATRA* specifications executed at a rate of almost 19,500 seps.[4] The performance of a simulator for the same application hand coded directly in C performed only slightly better. Namely, it executed at a rate of almost 20,000 seps. The performance of the simulator degrades considerably when extensive I/O and tracing operations are performed.

[3]System-defined TRA classes are mainly for i/o and debugging purposes.

[4]All simulations were performed on a SPARCstation SLC™ workstation.

3 Conclusion

Predictability can be *enhanced* in a variety of ways. It can be enhanced by restricting expressiveness as was done in Real-Time Euclid [8], by sacrificing accuracy as was done in the Flex system [7], or by abstracting segmented resources as was done in the Spring kernel [10]. The TRA-development methodology we are advocating here introduces one more way of improving predictability, that of allowing only physically-sound specifications. Pursuing the ideas presented in this paper will undoubtedly provide us with one more handle in our persistent quest for predictable systems. An interesting question to be addressed in the future would be whether this and other handles can be combined in any useful way to *guarantee* predictability.

Our experience with the TRA development methodology in the design, simulation, and analysis of asynchronous digital circuits, sensori-motor autonomous systems, and intelligent controllers confirms its suitability for the specification, verification, and validation of many embedded and time-critical applications. Its usefulness in the implementation of such systems, although promising, is yet to be established.

A fruitful direction for future research would be to automate the process of transforming TRA-based physically-sound time-critical specifications into provably-correct implementations given appropriate resources. Such research will have two complementary – experimental and theoretical – components. The experimental component would involve the development of a compiler to transform *CLEOPATRA* specifications into predictable real-time programs, given a dedicated computing platform. The theoretical component would aim at devising efficient verification algorithms that can be automated and incorporated in the *CLEOPATRA* compiler.

References

[1] Azer Bestavros. TRA-based real-time executable specification using CLEOPATRA. In *Proceedings of the 10th Annual Rochester Forth Conference on Embedded Systems*, Rochester, NY, June 1990. (revised May 1991).

[2] Azer Bestavros. Planning for embedded systems: A real-time prospective. In *Proceedings of AIRTC-91: The 3rd IFAC Workshop on Artificial Intelligence in Real Time Control*, Napa/Sonoma Region, CA, September 1991.

[3] Azer Bestavros. Specification and verification or real-time embedded systems using the Time-constrained Reactive Automata. In *Proceedings of the 12th IEEE Real-time Systems Symposium*, pages 244–253, San Antonio, Texas, December 1991.

[4] Azer Bestavros. *Time-constrained Reactive Automata: A novel development methodology for embedded real-time systems.* PhD thesis, Harvard University, Division of Applied Sciences (Department of Computer Science), Cambridge, Massachusetts, September 1991.

[5] Azer Bestavros, James Clark, and Nicola Ferrier. Management of sensori-motor activity in mobile robots. In *Proceedings of the 1990 IEEE International Conference on Robotics & Automation*, Cincinati, Ohio, May 1990. IEEE Computer Society Press.

[6] Azer Bestavros, Devora Reich, and Robert Popp. Cleopatra compiler design and implementation. Technical Report TR-92-019, Computer Science Department, Boston University, Boston, MA, August 1992.

[7] Jen-Yao Chung, Jane Liu, and Kwei-Jay Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transaction on Computers*, 19(9):1156–1173, September 1990.

[8] Eugene Kligerman and Alexander Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–949, September 1986.

[9] Nancy Lynch and Mark Tuttle. An introduction to Input/Output Automata. Technical Report MIT/LCS/TM-373, MIT, Cambridge, Massachusetts, November 1988.

[10] John Stankovic and Krithi Ramamritham. The Spring Kernel: A new paradigm for real-time operating systems. *ACM Operating Systems Review*, 23(3):54–71, July 1989.