ACM SIGPLAN Object-Oriented Programming and Systems (OOPS) Messenger, vol. 7, pp. 36-43, January 1996.

The Validation and Implementation of Real-Time Robotics Systems Using Cleopatra Object-Oriented Physically-Correct Specifications*

Azer Bestavros
(best@cs.bu.edu)
Computer Science Department
Boston University, MA 02215

Abstract

The specification of a real-time system is often the result of a process, whereby a conceptual control system is fleshed out as a computer program. To be accurate, this process must preserve important causal and structural properties of the control system. For example, if the control system has multiple functional components operating concurrently, then the process of mapping these components into a computer program executing on a single processor, must ensure that these components do not interact in ways that are physically impossible. In this paper we review our work on CLEOPATRA, an object oriented specification and programming language that restricts expressiveness in a way that allows the specification of only reactive, spontaneous, and causal computation. Unrealistic systems—possessing properties such as infinite capacities 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—not to mention the elimination of potential hazards that would have gone unnoticed.

1 Introduction

A computing system is *embedded* if it is a component of a larger system whose primary purpose is to monitor and control an environment. The leaping advances in computing technologies that the last few decades have witnessed have resulted in an explosion in the extent and variety of such systems. This trend is expected to continue in the future.

Usually, embedded systems are associated with critical applications, in which human lives or expensive machineries are at stake. Their missions are long-lived and uninterruptible, making maintenance or reconfiguration difficult. Examples include command and control systems, nuclear reactors, process-control plants, robotics, avionics, switching circuits and telephony, data-acquisition systems, and real-time databases, just to name a few. The sustained demands of the environments in which such systems operate pose relatively rigid and urgent performance requirements. Often, these requirements are stated as timing constraints on their behaviors. Wirth [17] singled out this aspect as the one aspect that differentiates real-time from other sequential and parallel systems. This led to a body of research on *real-time computing*, which encompasses issues of specification techniques, validation and prototyping, formal verification, fault-tolerance, safety analysis, programming languages, development tools, scheduling, and operating systems.

The absence of a unified suitable formal framework that addresses the aforementioned issues severely limits the usefulness of these studies. This situation is further exacerbated considering the range of disciplines employed in developing the various components of an embedded application. In this paper we propose CLEOPATRA,[1] a programming environment that recognizes the unique requirements of responsive embedded systems. CLEOPATRA 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. In particular, rather than describing behaviors using *control* structures, it describes behaviors using time-constrained causal structures. CLEOPATRA is object-oriented and compositional, thus advocating modularity and reusability. 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. Our experience confirms CLEOPATRA's suitability as a vehicle for the specification and validation of many embedded and time-critical applications. In particular, we used it to simulate and analyze asynchronous digital circuits, sensory-motor behavior of autonomous creatures, and intelligent controllers [4, 9, 3]. A compiler that allows the execution of CLEOPATRA specifications has been developed [10, 7], and is available via FTP from `cs.bu.edu:/bestavros/cleopatra/`.

CLEOPATRA is based on the Time-constrained Reactive Automata (TRA) formalism [5, 6]. Using the TRA model, an embedded system is viewed as a set of automata (TRAs), each representing an autonomous system entity. TRAs are reactive in that they abide by Lynch's input enabling property [15]; they communicate by signaling events on their output channels and by reacting to events signaled on their input channels. The behavior of a TRA is governed by time-constrained causal relationships between computation-triggering events. Us-

*This work has been partially supported by NSF (grant CCR-9308344).

[1] A C-based Language for the Event-driven Object-oriented Prototyping of Asynchronous Time-constrained Reactive Automata.

ing the TRA formalism, there is no conceptual distinction between a system and a property; both are specified as formal objects. This reduces the verification process to that of establishing correspondences – *preservation* and *implementation* – between such objects. The correspondence between CLEOPATRA and the TRA formalism is straightforward. Every object in CLEOPATRA corresponds to a TRA. In [6], the construction of a TRA, given a CLEOPATRA object, is detailed.

This paper is organized as follows. In Section 2, we describe the CLEOPATRA specification/programming language, along with an example that illustrates our "ounce of prevention" thesis [8]. In Section 3, we present a compiler that allows the execution of CLEOPATRA specifications for simulation/validation purposes. In Section 4, we describe an implementation environment, in which CLEOPATRA was used to program the motion controller for a robotics experiment. In section 5, we conclude with current and future research directions.

2 CLEOPATRA Specifications

In CLEOPATRA, systems are specified as interconnections of TRA objects. Each TRA object has a set of *state variables* and a set of *channels.* Time-constrained causal relationships between events occuring on different channels, and the computations 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 to specify more complex TRAs.

2.1 Classes and Objects

A TRA object specification in CLEOPATRA consists of two components: a header and a body. An object's header specifies its name, the parameters needed for its instantiation, and its signature. An object's body specifies its behavior. In its simplest form, this entails the specification of the TRA's state space and its potentially time-constrained set of reactions to the different events visible to it. More complex behaviors include (among others) the specification of: internal channels, initialization code, and interconnection of local (composed) objects. In CLEOPATRA, TRAs are defined in parametrized *classes.* For example, Figure 1 shows the class of integrators that use trapezoidal approximation, with two parameters: `TICK`, and `TICK_ERROR`.

```
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: Integration using the trapezoidal rule.

The header of a TRA class determines its external signature and signaling range function. For example, any TRA from the class `integrate` specified in Figure 1 has 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`. Start channels do not have to be explicitly included in the header of a TRA-class.

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 Time-constrained Event-driven Transactions 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`[2] 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 TET Specification

In CLEOPATRA, time-constrained causal relationships between events on different channels of a TRA-class, and the computations (state transitions) that they trigger, are specified using *Time-constrained Event-driven Transactions* (TET). A TET describes the reaction of a TRA to a subset of events. Such a reaction might involve responding to triggers and/or firing action(s). Figure 3 explains the relation between the triggering and firing of actions using TETs.

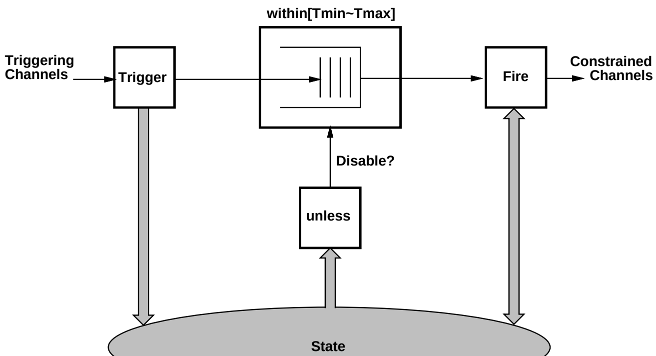

Figure 3: Time-constrained Event-driven Transaction.

[2] The behavior of an object from the `constant` class is to signal the value `VAL` on its only output channel `out` every `TICK` $\pm$ `TICK_ERROR` units of time.

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. It specifies at most one state variable (per triggering channel) where the value of a trigger on that channel is to be recorded. A TET with no triggering section is triggered every time an action is signaled on any channel of the TRA; its trigger set is considered to be the same as the TRA's signature. The fire section specifies the action value to be signaled on the controlled channel as a result of firing the TET. An absent expression means that a random value from the signaling range of the controlled channel is to be 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.

The first TET of the `integrate` class 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`. The second TET of the `integrate` class is an example of a transaction with both a trigger section and a fire section. 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`.

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 boolean expression (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. In other words, a reaction is aborted if at any point in time after its triggering (scheduling), the disabling condition becomes `true`. 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. Open, closed, and semi-closed time intervals can be used provided they specify an interval of time from the set $\mathcal{D}$. 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. We adopt a C-like syntax for the specification of TET methods. Statements in a TET method are executed sequentially. The state transition caused by the execution of a TET method 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 *CLEOPATRA* specification of a finite FIFO. Values fed into that object are delayed for a specified time before being produced as outputs.

The header of the `fifo` TRA-class identifies the channel `in` as input, and the channels `out`, `ack` and `overflow` as outputs. Although not explicitly specified as such, the channel `init` (the start channel) is assumed to be an input channel. 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, each of which underscores a causal relationship between the events triggering its execution and those resulting from its execution. The first TET in the body of the FIFO establishes a causal relationship between events signaled on `init` and and those signaled on `ack`. In particular, firing an action on `init` (the trigger) *causes* the firing of an action on `ack` (the result) after a a delay of at most `DLY_MIN`. The second TET establishes a similar causal relationship between events signaled on `in` and `ack`. The third TET establishes a causal relationship between events signaled on `in` and `out`. In particular, firing an action on `in` *causes* the firing of an action on `out` after a delay of at least `DLY_MIN` and at most `DLY_MAX` elapses, provided that the FIFO did not overflow as of the last initialization. The causal relationship that the fourth TET establishes can be explained similarly.

```
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 element in *CLEOPATRA*.

Each TET in a TRA-class specifies up to two possible state transitions. Consider, for example, the second TET in the FIFO specification given in Figure 4. In response to a trigger on `in`, the value of the triggering signal is stored in the state variable `y[i]`, thus resulting in a possible state change. Notice that this transition cannot be blocked or delayed; it is an *input transition.* The second state transition, an *output transition*, occurs with the firing of an action on `ack`, resulting in the adjustment of the values of the state variables `i` and `f`.

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. Thus, 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 malignant behaviors, the environment must wait for an acknowledgment `ack()`, or else wait for at least `DLY_MIN` before issuing a new input. Such safety conditions can be verified using TRA-based verification techniques [6].

We argue that any finite implementation of a 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 abiding 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 it spares system developers from trying to implement the impossible.

An indirect result of *CLEOPATRA*'s limited expressivity is to force system specifications to be spelled out at a "lower" level. For example, in *CLEOPATRA* one cannot specify a clock that does not drift. This implies that the consequences of this drift could not be simply discounted as "implementation details". Lowering the level at which specifications are expressed advocates a *functional* specification approach. In contrast to the *black box* approach, the operational approach calls for problem specification by formulating a system to *solve* it. The formulated system is given in terms of implementation-independent structures that, once implemented, would generate the required behavior [18].

3 *CLEOPATRA*-based Validation

We have developed a compiler that transforms *CLEOPATRA* specifications into an event-driven simulator for validation purposes. We have used the *CLEOPATRA* compiler to simulate a variety of systems. In particular, we used it extensively to specify and analyze sensory-motor robotics applications [9] and to simulate complex behaviors of autonomous creatures [4]. Figure 5 shows the different stages involved in the compilation and execution of specifications written in *CLEOPATRA*.

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.

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

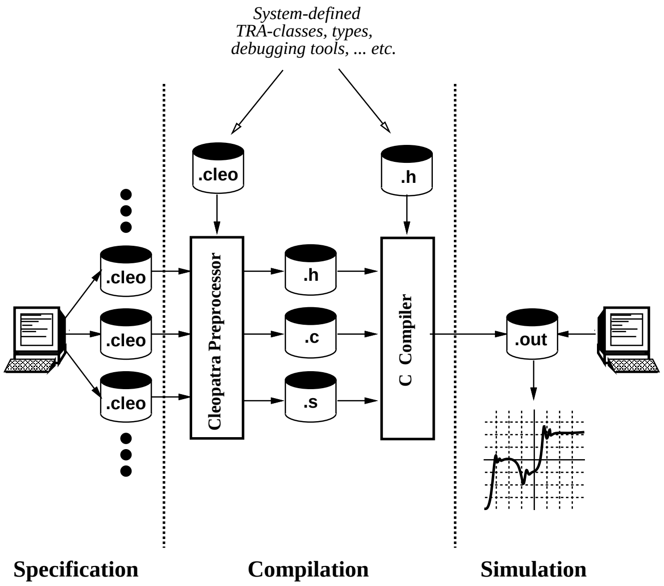

Figure 5: Compilation & simulation of *CLEOPATRA*.

In *CLEOPATRA*, any TRA-class with no input channels represents a stand-alone (closed) system whose behavior is independent from the outside world; it is a world of its own. One such TRA-class, namely `main`, is singled out by *CLEOPATRA* to represent the entire system being specified. For embedded systems, a typical `main` TRA-class will simply be the composition of a programmed system, representing the control system, and an external interface, representing the environment. For example, the `main` TRA-class shown in Figure 6 represents the *CLEOPATRA* specification of a closed process control system that models the interaction between a user TRA object and a plant TRA object. The execution of a *CLEOPATRA* stand-alone system is started by instantiating an object from the TRA-class `main` at time 0 and, thereafter, committing only the legal transitions dictated by the system specification and the semantics of the TRA model. Figure 7 shows the values signaled on the `x` and `z` channels over time.

A library of system-defined TRA-classes is available for debugging and performing I/O in *CLEOPATRA*. For example, in the specification of the TRA-class `main` given in Figure 6, the TRA-class `fmonitor` is used to record the action values signaled on the `x` and `z` channels in files `x.dat` and `z.dat` respectively. 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, `fmonitor` 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.

```
#include "sysTRA.cleo"
#define TAU 1
#define DLY 5

TRA-class user(double EPOCH)
  -> x(double)
{
  act:
   init(),x() -> x(random(0,1)):
     within [0.8*EPOCH~1.2*EPOCH]
       ;
}

TRA-class plant(double GAIN)
  y(double) -> z(double)
{
  state:
   double drive = 0, val = 0 ;
  act:
   y(drive) -> :
     ;
   init(), z() -> z(val):
     within [0.9*DLY~1.1*DLY]
       commit {val = val + GAIN*drive ;}
}

TRA-class world()
  y(double) -> x(double), z(double)
{
  include:
   user(300) -> x() ;
   plant(1.5) y() -> z() ;
}

TRA-class control()
  x(double), z(double) -> y(double)
{
  state:
   double s = 0, f = 0;
  act:
   x(s), z(f) -> y(s-f):
     within [0.95*TAU~1.05*TAU]
       ;
}

TRA-class main() ->
{
 internal:
  -> x(double),y(double),z(double)
 include:
  world y() -> x(), z() ;
  control x(), z() -> y() ;
  fmonitor("x.dat") x() -> ;
  fmonitor("z.dat") z() -> ;
}
```

Figure 6: The `main` TRA-class.

4 *CLEOPATRA*-based Implementation

To close the gap between formality and practicality, the development cycle of embedded applications has to be supported in its entirety. This requires that system implementation – and not only specification, validation and verification – be addressed. In this section, we describe on-going and future research in that direction.

For software processes, the distinction between an executable specification and an implementation is vague. This suggests that specification languages can themselves be used as programming languages. For real-time applications, this is true only if programs can be compiled to execute in real-time rather than in simulated time. Currently, we are developing a compiler for real-time programs written in *CLEOPATRA*.

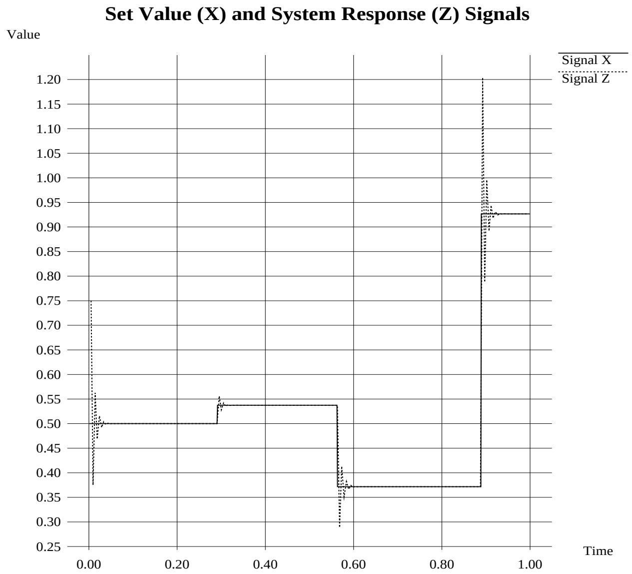

Figure 7: Simulated behavior of an underdamped process control system.

Figure 8 illustrates the various component of a *CLEOPATRA*-based implementation environment. The target machine for the compiler is a distributed VME-based dedicated shared-memory 68030 single-board computers running real-time operating system kernels. *CLEOPATRA* program development, debugging, and monitoring is to be done using a standard Unix-based workstation environment linked to that target.

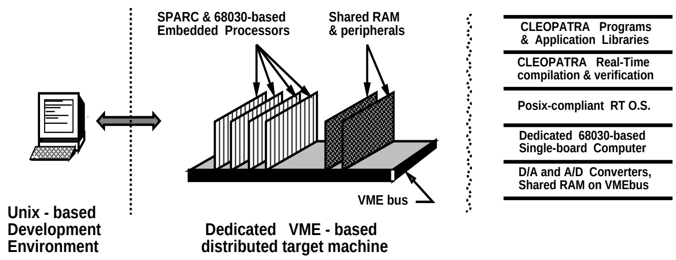

Figure 8: Components of a *CLEOPATRA*-based implementation environment

Compilers for real-time languages like *CLEOPATRA* are complicated by the fact that they are required to verify the *feasibility* of the compilation process. In other words, in addition to checking syntax and semantics, such compilers have to establish that, given a specific hardware configuration, the compiled code will observe all the timing constraints specified in the source code. A simpler approach to address that problem would be to generate code that raises exceptions whenever a violation of a specified time constraint is detected during execution. This is similar to raising exceptions as a result of run-time errors in conventional (non-real-time) languages. Due to its simplicity, we have elected to follow the latter approach in our initial implementation. Meanwhile, we intend to investigate and develop efficient verification algorithms that would potentially lead to the adoption of the former approach.

4.1 A Robotics Testbed

In order for a language to be instrumental in implementing practical systems, it must be geared towards a specific application through the development of appropriate libraries and verification tools. Our intention is for *CLEOPATRA* to target robotic applications. Robotics applications are good representatives for "real" embedded systems. The tasks involved therein are diverse (vision, motion control, high-level planning, ...*etc.*) and make use of very different resources (special purpose image processors, tailor made controllers and drivers, tightly- and loosely-coupled computer networks, massively parallel architectures, ...*etc.*) In addition, the interaction between these tasks is non-trivial and highly time-constrained. Being able to model, and even implement, such complex systems in a single framework is both challenging and attractive.

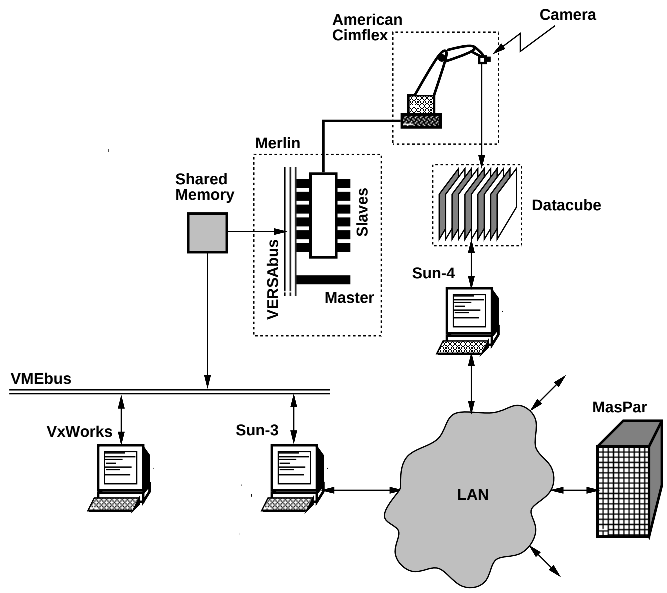

Figure 9: Set-up for a sensori-motor activity coordination experiment

A robot system will typically have associated with it a number of sensing subsystems. If these sensing subsystems are *active*,[4] they will each be issuing motor requests related to the sensory processing algorithm that they are performing. In a general robot, however, the manipulative systems will also be required to perform duties not related to the acquisition of sensory information, but aimed at influencing the robot's environment in a purposive manner. Thus, both of these sensory and manipulatory subsystems, will be competing for a limited resource, that of the positional degrees of freedom of the robot. Such a competition has to be managed.

As an example of competing requests management, consider a robot whose active perception system requires it to move around a block in order to see what it occludes, and whose manipulative system requires it to stand still so that it can grasp a nearby object, which would be, otherwise, out of reach. It can be seen by this simple example that one cannot decouple the motor activities requested by the active perception system from the motor activities requested by the manipulative system. Thus any system that is developed for controlling motor activities in a robot must take into account both the perceptual goals and the manipulative goals of the machine and produce motions which address these goals in an integrated and orderly manner.

Another crucial problem in sensory-motor robotics activities is that of coordination. For example, a vision system might be required to synchronize its sampling with the robot motion. In particular, it might require the robot to remain stand still at a given coordinate for a specific period of time in the future to grab a frame.

One can think of the motor units of the robot as a limited resource that must be shared between the active perception and manipulative systems. The motion control operating system must arbitrate and/or coordinate between the conflicting requirements of these two systems in a way which allows the goals of the two systems to be attained. In [9], we suggested a methodology based on the TRA model that allows one to schedule the motor commands sent to the various actuators in the robot in a manner appropriate to the robot goals.

In [13], an experiment that adopts the TRA framework was proposed. The experiment involves the coordination of motor requests to perform manipulative tasks using directed-vision feedback. The testbed for the experiment (Figure 9) consists of a six-degrees of freedom (American Cimflex) industrial robot connected to a dedicated (Merlin) controller. The controller consists of six parallel MC6809 processors (slaves), each controlling one of the robot's actuators. A single board 68000-based computer VM02 (master) is responsible for driving these processors in real-time. The backbone of the Merlin controller is a VERSAbus which is connected via a Synergist-II VMEbus-VERSAbus translator and a BIT-3 VME-VME adaptor to the bus extender of a SUN-3 workstation (MIPS). On the same bus extender, another 68000-based single-board computer (REAL1) is running VxWorks, a real-time operating system kernel. The Unix-based (MIPS) workstation provides an environment for developing and debugging robotics applications, whereas REAL1 is used to run these applications in real-time. In addition, MIPS acts as a Local Area Network gateway to the other computing facilities in the robotics lab. This includes the MASPAR massively parallel computer, and the data-cube special-purpose array processor for image processing. A video-camera connected to the data-cube is mounted on the American Cimflex robot arm.

[4] Active perception implies usage of the robots manipulative systems to move about and interact with the environment in ways that serve the sensory processes [1].

The VM02 master computer of the Merlin controller runs a High Speed Host Interface (HSHI) program that allows it to receive commands at a rate of up to 250 commands per second – a 4 milliseconds latency – to remotely control the robot from a host computer. In [2] we described an interface that we designed and implemented to allow a SUN workstation to communicate with HSHI via a dual-ported shared memory piggy-backed on the Bit-3 VME-VME adaptor. A drawback of that connection was our reliance on UNIX, a non-real-time operating system. Recently, we have successfully modified our interface to work from REAL1 under the VxWorks real-time operating system. This allowed us to execute time-sensitive tasks safely.

4.2 A Simple Motion Controller

We used *CLEOPATRA* to write a motion control algorithm for Merlin (the six-degrees of freedom American Cimflex industrial robot). The algorithm allowed the robot arm to mimic the motion of a human arm as seen through the vision system in a real-time manner. A simplified version of this algorithm is shown in figure 10 and is explained below.

The controller accepts 3 inputs from the environment over 3 channels: `i_pos()`, `c_pos()`, and `ctrl_ack()`. The `i_pos()` channel is the interface between the vision subsystem and the controller. An action on the `i_pos()` channel carries a vector of 6 numbers representing the perceived position (X, Y, and Z coordinates) and orientation (roll, pitch, and yaw angles) of the human arm. The `c_pos()` channel is the feedback signal from the Merlin controller. An action on the `i_pos()` channel carries a vector of 6 numbers representing the current position and orientation of the robot arm. The `ctrl_ack()` channel carries acknowledgements from the robot arm actuator. An action on the `ctrl_ack()` channel implies that the actuator is done with the previous motion request and is ready for the next one.

The controller produces 2 outputs to the environment over 2 channels: `ctrl()` and `alarm()`. The `ctrl()` channel carries the command signal to the robot arm actuator. An action on the `ctrl()` channel carries a vector of 6 numbers representing the next (requested) position and orientation of the robot arm. The `alarm()` channel provides an exception signal to higher-level controllers in the system. An action on the `alarm()` channel signifies the failure of the controller to meet a timing constraint (alarm condition 1, 2, and 3) or to keep the error between the position/orientation of the robot and human arms within a *safe* margin (alarm condition 0).

In addition to its external input and output channels, the controller has an internal channel, `safety_check()`, which is used to trigger periodically a test on the error between the position/orientation of the robot arm and the human arm. This periodic behavior illustrates how synchronous reactions could be specified in *CLEOPATRA*.

The reaction of the controller to its inputs is simply to latch the latest requested positions signaled on the `i_pos()` channel and the latest feedback signaled on the `c_pos()`. If these actions are not processed within `LATCH_DLY` and `FEED_DLY` units of time, respectively, then appropriate exceptions are raised on the `alarm` channel.

The output behavior of the controller is such as to produce an action on the `ctrl()` channel and then wait for an acknowledgement on the `ctrl_ack()` channel be-

```
TRA-class mimic(double CHECK_DLY, CNTRL_DLY,
                       LATCH_DLY, FEED_DLY,
                       ACK_DLY, TICK)
  i_pos(double[6]), c_pos(double[6]), ctrl_ack()
     -> ctrl(double[6]), alarm(int)
{
 state:
  double old_pos[6], req_pos[6] ;
  bool ready = FALSE ;

 internal:
  -> safety_check() ;

 act:
  init() -> safety_check(), ctrl(req_pos) :
   commit {
     initialize(req_pos,old_pos) ;
     ready = TRUE ;
     toggle = 0 ;
   }

  safety_check() -> safety_check():
   within [CHECK_DLY~CHECK_DLY+TICK]
     ;

  safety_check() -> alarm(0):
   unless (safe(req_pos, old_pos))
    before CHECK_DLY
     ;

  i_pos(req_pos) -> alarm(1):
   unless (ready)
    before LATCH_DLY
     commit {
       ready = FALSE ;
     }

  c_pos(old_pos) -> :
   before TICK
    commit {
      feed_toggle = 1 - feed_toggle ;
    }
     ;

  c_pos() -> alarm(2):
   within [FEED_DLY~FEED_DLY+TICK]
    unless(! stable(feed_toggle))
     ;

  ctrl_ack() -> ctrl(req_pos):
   before TICK
    commit {
      ready = TRUE ;
      ack_toggle = 1 - ack_toggle ;
    }

  ctrl_ack() -> alarm(2):
   within [ACK_DLY~ACK_DLY+TICK]
    unless(! stable(ack_toggle))
     ;
}
```

Figure 10: Simple controller for American Cimflex.

fore issuing a new `ctrl` action. If the `ctrl_ack()` action is not received within `ACK_DLY` units of time, then an `alarm()` exception is signaled. The relationship between actions on the `ctrl()`, `ctrl_ack()` channels illustrates how asynchronous reactions could be specified in *CLEOPATRA*.

Notice that the controller in figure 10 continues to operate even after an exception is raised by firing an action on the `alarm` channel. Our implementation enabled more complex behaviors (specified in *CLEOPATRA* and running concurrently) to deal with these exceptions. This methodology is in line with the subsumption architecture [11], which empowers a module from a higher layer to overwrite the output of a module from a lower layer. The higher layer is called a dominant behavior, whereas the lower layer is called an inferior behavior. Subsumption allows control systems to be patched up by allowing smarter (or higher priority) behaviors to take over from default behaviors whenever appropriate.

5 Conclusion

Predictability can be *enhanced* in a variety of ways. It can be enhanced by restricting expressiveness as was done in Real-Time Euclid [14], by sacrificing accuracy as was done in the Flex system [12], or by abstracting segmented resources as was done in the Spring kernel [16]. The TRA-development methodology we are advocating in this paper 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.

In this paper, we have portrayed *CLEOPATRA* as a language suitable for the validation and implementation of embedded real-time systems. In that respect, *CLEOPATRA* possesses a number of features that make it attractive. It features a C-like imperative syntax for the description of computation, which makes it easier to incorporate in real applications already using C. It is object-oriented, thus advocating modularity, reusability, and off-the-shelf hierarchical programming of embedded systems. Finally, it is event-driven and, as such, distinguishes clearly between causality and dependency. Our experience with *CLEOPATRA* in the design, simulation, and analysis of asynchronous digital circuits, sensory-motor autonomous systems, and intelligent controllers has confirmed the value of these features.

References

[1] R. Bajcsy. Perception with feedback. In *Proceedings of the 1988 Darpa Image Understanding Workshop*, 1988.

[2] Azer Bestavros. *The Michael - Merlin Connection: Programming tools for the remote control of the American Cimflex robot.* Robotics Laboratory, Harvard University, Cambridge, MA, September 1988.

[3] 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).

[4] 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.

[5] Azer Bestavros. Specification and verification or real-time embedded systems using the Time-constrained Reactive Automata. In *Proceedings of RTSS'91: The 12^{th} IEEE Real-time Systems Symposium*, pages 244–253, San Antonio, Texas, December 1991.

[6] 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.

[7] Azer Bestavros. Cleopatra: Physically-correct specifications of embedded real-time programs. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, Orlando, FL, June 1994.

[8] Azer Bestavros. An ounce of prevention is worth a pound of cure: Towards physically-correct specifications of embedded real-time systems. In *Proceedings of COMPASS'94: The Ninth Annual IEEE Conference on Computer Assurance*, Gaithersburg, MD, June 1994.

[9] 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.

[10] 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.

[11] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, RA-2:14–23, April 1986.

[12] 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.

[13] James Clark, Nicola Ferrier, and Lei Wang. A robotics system for manipulation using directed vision feedback. Internal report, Robotics laboratory, Harvard University, Cambridge, MA, 1991.

[14] 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.

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

[16] 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.

[17] Niklaus Wirth. Toward a discipline of real-time programming. *Communications of the ACM*, 20(8), August 1977.

[18] Pamela Zave. The operational versus the conventional approach to software development. *Communications of the ACM*, 27(2):104–118, February 1984.