In Proceedings of COMPASS'94: The Ninth Annual IEEE Conference on Computer Assurance, Gaithersburg, MD, June 1994.

An Ounce of Prevention is Worth a Pound of Cure

Towards Physically-Correct Specifications of Embedded Real-Time Systems

Azer Bestavros*

Department of Computer Science
Boston University
Boston, MA 02215

Abstract

Predictability – the ability to foretell that an implementation will not violate a set of specified reliability and timeliness 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 a formalism that restricts expressiveness in a way that allows the specification of only reactive, spontaneous, and causal computation. 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.

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 [28] 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.[1]

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. For example, in a simple sensori-motor robotic application [12], algorithms from various disciplines like low-level imaging, active vision, tactile sensing, path planning, compliant motion control, and non-linear dynamics may be utilized [13]. Not only are these disciplines different in their abstractions and programming styles, but also they differ in their computational requirements, which range from single-board dedicated processors to massively parallel general-purpose computers.

In this paper we propose CLEOPATRA,[2] 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 se-

*This research is supported by NSF (grant CCR-9308344).

[1] For comprehensive surveys of recent research in real-time systems, the reader is directed to [22, 10, 26, 27].

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

mantically 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, sensori-motor behavior of autonomous creatures, and intelligent controllers [5, 8, 4]. A compiler that allows the execution of *CLEOPATRA* specifications has been developed [9], and is available via FTP from `cs.bu.edu:/bestavros/cleopatra/`.

CLEOPATRA is based on the Time-constrained Reactive Automata (TRA) formalism [6, 7]. 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 [18]; 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. Using 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.

This paper is organized as follows. In Section 2, we overview the TRA model. We emphasize the TRA operational semantics, which underlies the execution model of *CLEOPATRA*. In Section 3, we describe the *CLEOPATRA* specification/programming language, along with an example that illustrates our "ounce of prevention" thesis. In Section 4, we present a compiler that allows the execution of *CLEOPATRA* specifications. In Section 5, we conclude with current and future research directions.

2 The TRA Model

The TRA model has evolved from our earlier work in [3] extending Lynch's IOA model [18, 17] to suit embedded and time-constrained computation.

2.1 Novelties

Previous studies in modeling real-time computing have focussed on adding the notion of time without regard to physical properties of the modeled systems. This makes it possible to specify systems that do not abide by principles like causality and spontaneity. Using the TRA model, requirements that are physically impossible to guarantee are not possible to express. This preventative approach is likely to spare a lot of time and energy in the development cycle (specification, implementation, and verification) of responsive systems.

The TRA model deals not only with the notion of time, but also with the notion of space. Events occur at uniquely identifiable points in time as well as in state space. Concurrent events are permitted only if they affect disjoint state subspaces. The payoff for this dual treatment of space and time is manifold. In particular, mappings between various levels of abstractions for compilation and verification purposes become more robust as the formalism becomes more structured.

The TRA model does not allow the specification of systems that are not *reactive*. A system is reactive if it cannot block the occurrence of events not under its control. This property is crucial for accurate and realistic modeling of embedded and real-time systems. A sufficient condition for reactivity is the *input enabling* property proposed in [18]. The TRA model is input enabled. It distinguishes clearly between environment-controlled actions, which cannot be restricted or constrained, and locally-controlled actions, which can be scheduled and disabled.

A non-deterministic system is *causal* if given two inputs that are identical up to any point in time, there exist outputs (for the respective inputs) that are also identical up to the same point in time. The TRA model enforces causality by requiring that any local action be produced only as a *result* of an earlier *cause*. We distinguish clearly between causality and dependency. An event occurs as a result of exactly one earlier event but may depend on many others as reflected in the state of the system. This spares our formalism from dealing with clairvoyant and capricious behaviors [24].

Spontaneity is a notion closely related to causality. A system is *spontaneous* if its output actions at any given point in time t cannot depend on actions occuring at or after time t. In particular, if an output occurs simultaneously with (say) an input transition, the same output could have been produced without the simultaneous input transition [21]. Simultaneity is, thus, a mere coincidence; the output event could have occurred spontaneously even if the input transition was delayed. The TRA model enforces spontaneity by requiring that simultaneously occuring events be independent; time has to *necessarily* advance to observe dependencies.

A computing system that maintains perfect timing information cannot be implemented. Nevertheless, formal models (such as the Timed Finite Automata [2] or the Timed Input-Output Automata [16]) allow the specification of perfect clocks. The TRA model does not provide for (or allow the specification of) *perfect clocks*. As a consequence, the only measure of time available for system processes has to be relative to imperfect, locally-maintained clocks. This distinction between real time and perceived time is important when dealing with embedded applications where time properties are stated with respect to real time, but have to be preserved relying on perceived time.

2.2 Basic definitions

We adopt a continuous model of time similar to that used in [1, 15]. We represent any point in time by a nonnegative real $t \in \Re$. Time intervals are defined by specifying their end-points which are drawn from the set of nonnegative rationals $\mathcal{Q} \subset \Re$. A time interval is viewed as a traditional set over nonnegative real numbers. It can be an empty set, in which case it is denoted by ε, it can be a singleton set, in which case it is denoted by the $[t, t]$, $t \in \mathcal{Q}$, or else it can be an infinite set, in which case it is denoted by $[t_l, t_u]$, $(t_l, t_u]$, $[t_l, t_u)$, or (t_l, t_u) – the right-closed, left-closed, and open time intervals, respectively, where $t_l, t_u \in \mathcal{Q}$ and $t_l < t_u$. The set of all such infinite time intervals is denoted by $\mathcal{D}$.

A real-time system is viewed as a set of interacting mealy automata (TRAs), which communicate with each other through *channels*. A channel is an abstraction for an *ideal* unidirectional communication. The information that a channel carries is called a *signal*, which consists of a sequence of *events*. An event underscores the occurrence of an *action* at a specific point in time. An action is a *value* associated with a channel. For example, let `North`, `South`, `East`, and `West` be the possible values that can be signaled on some channel `MOVE`. `MOVE(East)` is, therefore, a possible action. The instantiation of `MOVE(East)` at time t_1 denotes the event $\langle \texttt{MOVE(East)} : t_1 \rangle$. The sequence $\langle \texttt{MOVE(East)} : t_1 \rangle \langle \texttt{MOVE(North)} : t_2 \rangle \langle \texttt{MOVE(South)} : t_3 \rangle$ ... *etc.* constitutes a signal. A signal cannot convey more than one event simultaneously. That is, for a signal $\langle a_0 : t_0 \rangle \langle a_1 : t_1 \rangle \dots \langle a_k : t_k \rangle \dots$ we require that $t_k < t_{k+1}, k \geq 0$.

At any point in time, a TRA is in a given *state*. The set of all such possible states defines the TRA's *state space*. The state of a TRA is visible and can only be changed by local *computations*. Computations (and thus state transitions) are triggered by actions and might be required to meet specific timing constraints.

2.3 TRA Objects

Definition 1 *A Time-constrained Reactive Automaton is a sextuple* $(\Sigma, \sigma_0, \Pi, \Theta, \Lambda, \Upsilon)$, *where*

- ◇ Σ, *the* TRA *signature, is the set of all the* TRA *channels. It is partitioned into three disjoint sets of input, output, and internal channels. We denote these by* Σ_{in}, Σ_{out}, *and* Σ_{int}, *respectively. The set consisting of both input and output channels is the set of external channels* (Σ_{ext}). *These are the only channels visible from outside the* TRA. *The set consisting of both output and internal channels is the set of local channels* (Σ_{loc}). *These are the locally controlled channels of the* TRA.
- ◇ $\sigma_0 \in \Sigma_{\text{in}}$ *is the start channel.*
- ◇ Π, *the signaling range function, maps each channel in* Σ *to a possibly infinite set of values that can be signaled as actions on that channel. Action sets of different channels are disjoint. The set of all the actions of a* TRA *is given by* $\Pi(\Sigma)$. *The set of input, output, internal, external, and local actions are similarly given by* $\Pi(\Sigma_{\text{in}})$, $\Pi(\Sigma_{\text{out}})$, $\Pi(\Sigma_{\text{int}})$, $\Pi(\Sigma_{\text{ext}})$, *and* $\Pi(\Sigma_{\text{loc}})$, *respectively.*
- ◇ Θ *is a possibly infinite set of states of the* TRA. *The set* Θ *can be expressed as the cross product of a finite number of subspaces* $\Theta = \Phi_1 \times \Phi_2 \times \ldots \times \Phi_p$, *where* $p \geq 1$ *denotes the dimensionality of the state space.*
- ◇ $\Lambda \subseteq \Theta \times \Pi(\Sigma) \times \Theta$ *is a set of possible computational steps of the* TRA. TRA*s are input enabled which means that for every* $\pi \in \Pi(\Sigma_{\texttt{in}})$, *and for every* $\theta \in \Theta$, *there exists at least one step* $(\theta, \pi, \theta') \in \Lambda$, *for some* $\theta' \in \Theta$. *Thus,* Λ *defines a total multi-function* $\Lambda : \Theta \times \Pi(\Sigma_{\texttt{in}}) \rightarrow \Theta$.
- ◇ $\Upsilon \subseteq \Sigma \times \Sigma_{\text{loc}} \times \mathcal{D} \times 2^{\Theta}$ *is a set of time-constrained causal relationships (or simply time constraints) of the* TRA. *A time constraint* $\upsilon_i \in \Upsilon$ *is a quadruple* $(\sigma_i, \sigma_i', \delta_i, \Theta_i)$ *whose interpretation is that: if an action is signaled at time* $t \in \Re$ *on the channel* σ_i, *then a corresponding action must be fired on the channel* σ_i' *at time* $t' > t$, *where* $t' - t \in \delta_i$, *provided that the* TRA *does not enter any of the states in* Θ_i *for the open interval* (t, t').[3] *The channel* $\sigma_i \in \Sigma$ *is called the trigger of the time constraint, whereas* $\sigma_i' \in \Sigma_{\text{loc}}$ *is called the constrained channel.* $\Theta_i \subseteq \Theta$ *defines the set of states that disable the time constraint; once triggered a time constraint becomes and remains active until satisfied or disabled. A time constraint is satisfied by the firing of an action on the channel* σ_i *within the time bounds imposed by the interval* δ_i; *it is disabled if the* TRA *enters in one of the disabling states in* Θ_i *before it is satisfied.*

As an example of a TRA specification, consider the the up/down counter whose state diagram is shown in Figure 1. The counter accepts commands issued on the input channel `cmd` to count up or down and signals the value of the current count (state) on the output channel `cnt`. The counter starts its operation once an action is fired on the `init` channel. The value of the `init` action determines the starting state of the counter. The counter is constrained to produce a count every at least 1.9 and at most 2.1 units of time, once it starts execution. Figure 1 shows the TRA-specification of such a counter.

The first three components of a TRA sextuple can be viewed as defining an interface between the TRA and its environment. In particular, they identify its external signature $\Sigma_{\text{in}} = \{\texttt{init}, \texttt{cmd}\}, \Sigma_{\text{out}} = \{\texttt{cnt}\}$, the identity of the start channel $\sigma_0 = \texttt{init}$, along with the signaling range of all the channels in Σ_{ext}. The last three components of a TRA sextuple identify its

[3] Notice that this condition does not necessitate the existence of a computational step $(\theta, \pi', \theta') \in \Lambda$ for each $\theta \in \Theta - \Theta_i$, where $\pi' \in \Pi(\sigma_i')$ and $\theta' \in \Theta$, since the specification of the TRA might avoid being in θ when σ_i' is scheduled to fire.

behavior. The state space defines the spatial structure of the computation. For the counter of Figure 1, this structure is unidimensionally spanned by θ. The set of computational steps defines the effect of events on the state of the TRA. The set of time-constrained causalities defines the rules governing the *scheduling* of the TRA's local events. For the counter of Figure 1, there are two such rules.

2.4 Space and Time aspects of TRAs

The behavior of a TRA is generally non-deterministic. Two sources of non-determinism can be singled out. In a given state there may be a number of choices concerning the channel and action to be fired. Each one of these choices results in a different computational step. This gives rise to *control* non-determinism, which presents a *spacial uncertainty* because different computational steps may affect different parts of the TRA state space. The TRA timing constraints specify lower and upper bounds on the delay between causes and effects, thus leaving the TRA with a potentially infinite number of choices concerning the exact delay to be exhibited. This gives rise to *timing* nondeterminism. Considered separately, control and timing nondeterminisms do not violate any physical principles. However, a combination thereof deserves a closer attention because it is related to the notions of space and time.

Two computational steps *conflict* if both of them introduce changes to at least one of the subspaces of the TRA's state space. This is formally defined below.

Definition 2 *Two steps* $(\theta_i, \pi_i, \theta'_i), (\theta_j, \pi_j, \theta'_j) \in \Lambda$ *conflict if and only if for some dimension* k *of* Θ, $\theta_i[k] \neq \theta'_i[k]$ *and* $\theta_j[k] \neq \theta'_j[k]$, *where* $1 \leq k \leq n$.

It is important to realize that the conflict relationship depends not only on a TRA's computational behavior, but also on the structure of its state space. In particular, two TRAs with isomorphic computational steps could have very different conflict relationships depending on their state space characterizations. The notion of conflicting computational steps can be easily extended to actions and channels. This is formally defined below.

Definition 3 *Two actions* π_i *and* π_j *conflict if there exist at least two conflicting computational steps* $(\theta_i, \pi_i, \theta'_i), (\theta_j, \pi_j, \theta'_j) \in \Lambda$. *Two channels* σ_i *and* σ_j *conflict if at least one action from* $\Pi(\sigma_i)$ *and one action from* $\Pi(\sigma_j)$ *conflict.*

The conflict relationship depicts computational dependencies that emerge due to sharing information about state. For two local actions to conflict, their respective channels must be under the control of a single *component* of the TRA. The transitive closure of the conflict relationship, therefore, defines a partition on the locally-controlled channels of a given TRA.

Definition 4 *Two local channels* σ_i *and* σ_j *belongs to the same component (class) if they conflict.*

The partition into classes of the TRA's locally-controlled channels captures some of the structure of the system the automaton is modeling or the set of requirements it is specifying. In particular, each class of channels represents the set of channels locally-controlled by *some* system component. This partitioning retains the basic control structure of the system's primitive components and provides a concrete notion of spatial locality.

To preserve the non-blocking (input-enabled) nature of the TRA model, it is necessary to insure that input actions on different channels do not conflict. A TRA is improper if at least two of its input channels conflict, otherwise it is proper. For the remainder of this paper, we assume that any TRA is *proper*.

The notion of system components we are presenting here is novel and entirely different from that used in untimed models to express fairness [18] by requiring that, in an infinite execution, each of the system's components gets infinitely many chances to perform its locally-controlled actions. In timed systems, the major concern is *safe* and not necessarily *fair* executions [20]. Even if required, fairness can be enforced by treating it as a safety property; liveness properties can be handled in infinite execution by requiring time to grow unboundedly.[4]. This led to the abandoning of the idea of partitioning a system into components in our earlier model proposed in [3]. Lynch and Vaandrager [19] followed suit in their recent modification of the model proposed in [25]. In the TRA model we use system components to represent what can be termed as *spatial locality*. Different actions can be signaled at the same "time" only if they are not signaled from the same "place"; they can be produced at the same "place" only if they do not occur at the same "time".

2.5 TRA Executions and Behaviors

In standard automata theory, there is no distinction between choosing a transition and firing it; they constitute a unique, instantaneous, and atomic activity. In the TRA model a distinction is made whereby choosing (scheduling) a transition and executing (committing) that transition are separate activities. They are *distinct* in that they are separated in time. In fact, a scheduled transition does not have to be committed; it can be abandoned due to unforseeable conditions. The distinction between the two activities is also pronounced in the way the TRA model differentiates between input and local events. Input events are not under the TRA's control; they cannot be blocked or delayed. Local events are under the TRA's control; they are time constrained, and could be disabled.

Consider the time constraint $\upsilon_i = (\sigma_i, \sigma'_i, \delta_i, \Theta_i) \in \Upsilon$, which identifies a time-constrained causal relation-

[4]Such executions were called *admissible* in [19]

ship between the events signaled on σ_i and those signaled on σ'_i. The occurrence of a trigger on σ_i results in an intention to perform an action on σ'_i within the time frame imposed by δ_i. The commitment (abandonment) of such an intention in due time is conditional on the states assumed by the TRA from when the intention is posted until it is committed (abandoned). At any point in time, a TRA might have several outstanding intentions. In particular, the occurrence of a single event might generate a number of intentions, each dictated by a different time constraint. Different outstanding intentions are not necessarily imposed by different time constraints. In particular, the repeated occurrence of a triggering event might generate a number of outstanding intentions, all of which are imposed by the same time constraint.

The *state* of a TRA at an arbitrary point in time is not sufficient to construct its *future behavior*. In addition to the state, the intervals of time where scheduled transitions might fire (due to earlier triggers) have to be recorded. For a given TRA, we define the *intention vector* $I = \vec{\Delta}$ to be a vector of r sets of intentions, where $r = |\Upsilon|$. Each entry in I is associated with one of the TRA's time constraints. If $\upsilon_i = (\sigma_i, \sigma'_i, \delta_i, \Theta_i) \in \Upsilon$ is one of the TRA's time constraints, then $I[\upsilon_i] = \{\delta_{i1}, \delta_{i2}, \ldots, \delta_{ik}, \ldots \delta_{im}\}$ denotes a set of m time intervals during which actions on the channel σ'_i are intended to be fired as a result of earlier triggers on σ_i. Each one of the intervals in Δ_i can be thought of as an independent *activation* of the time constraint υ_i. An empty intentions set, $I[\upsilon_i] = \phi$, indicates the absence of any activations of υ_i. The empty intention vector, I_ϕ, consists of r such empty sets.

Definition 5 *We define the status of a* TRA *at any point in time $t \in \Re$ to be the tuple (θ, I), where θ and I are the* TRA*'s state and intention vector at time t, respectively.*

A TRA changes its status only as a response to the occurrence of an event (input or local). In other words, the change in a TRA's status is necessarily a causal *reaction* to an input event or to an earlier triggering event. Five conditions – namely, legality, spontaneity, safety, causality, and consistency – have to be met for a status succession to occur. These are formally specified below.

Definition 6 *Assume that the status (θ, I) of a* TRA *was entered at time t. Furthermore, assume that at a later time $t' > t$, a set of simultaneous actions $\pi_1 \in \Pi(\sigma_1), \pi_2 \in \Pi(\sigma_2), \ldots, \pi_m \in \Pi(\sigma_m)$ were fired, where $\sigma_j \in \Sigma, 0 \leq j \leq m$. As a result, the* TRA *will assume a new status (θ', I'), where $I' = (I \cup I'_{\text{enabled}}) - (I'_{\text{fired}} \cup I'_{\text{disabled}})$.*

The status (θ', I') is called a valid successor *of the status (θ, I) due to the occurrence of the set of simultaneous events $\langle \pi_1, \pi_2, \ldots, \pi_m : t' \rangle$, if and only if the following conditions hold:*

1. Spontaneity:
The channels $\sigma_1, \sigma_2, \ldots, \sigma_m$ do not conflict; they belong to different TRA *components.*

2. Legality:
There exists some sequence of transitions $(\theta, \pi_1, \theta_1), (\theta, \pi_2, \theta_2), \ldots (\theta, \pi_m, \theta_m) \in \Lambda$, such that $\theta_m = \theta'$.

3 Safety:
For every intention $\delta_{ik} \in I[\upsilon_i]$, $t'' \in \delta_{ik}$ for some $t'' > t'$, $t'' \in \Re$, where $\upsilon_i \in \Upsilon$.

4. Causality:
For all $\sigma_i \in \Sigma_{\text{loc}}$, the following conditions hold

a. If $\sigma_i \neq \sigma_j$ for all $1 \leq j \leq m$ then for every $\upsilon_k = (\sigma_k, \sigma'_k, \delta_k, \Theta_k) \in \Upsilon$ for which $\sigma'_k = \sigma_i$, $I'_{\text{fired}}[\upsilon_k] = \phi$.

b. Otherwise, let $\Upsilon_i \subseteq \Upsilon$ be the set of time constraints with σ_i as the constrained channel, then there must exist exactly one time constraint $\upsilon_r \in \Upsilon_i$ such that:
$\diamond$ $I'_{\text{fired}}[\upsilon_r] = \{\delta_{rk}\}$, where $\delta_{rk} \in I[\upsilon_r]$ and $t' \in \delta_{rk}$, and
$\diamond$ $I'_{\text{fired}}[\upsilon_k] = \phi$, where $\upsilon_k \in \Upsilon_i$ and $\upsilon_k \neq \upsilon_r$.

5. Consistency:
For every time constraint $\upsilon_k = (\sigma_k, \sigma'_k, \delta_k, \Theta_k) \in \Upsilon$, the following conditions hold

a. If $\theta' \in \Theta_k$, then
$\diamond$ $I'_{\text{disabled}}[\upsilon_k] = I[\upsilon_k]$ and
$\diamond$ $I'_{\text{enabled}}[\upsilon_k] = \phi$.

b. Otherwise
$\diamond$ $I'_{\text{disabled}}[\upsilon_k] = \phi$, and
$\diamond$ If $\sigma_k = \sigma_j$ for some $1 \leq j \leq m$, then $I'_{\text{enabled}}[\upsilon_k] = \{(t' + \delta_i)\}$, else $I'_{\text{enabled}}[\upsilon_k] = \phi$.

In the above definition, the *spontaneity* condition allows the occurrence of simultaneous events only if they do not conflict. This guarantees that the transition from θ to θ' is independent of the ordering of concurrent computational steps. The *legality* condition ensures that the state change from θ to θ' is the result of defined computational steps. The *safety* condition guarantees that no active time constraint expires. In other words, outstanding intentions are either committed or abandoned *in due time*. The *causality* condition necessitates that local events be causal; they are signaled only if intended due to an earlier trigger. Thus, the causality condition guarantees that there is exactly one committed intention per local event. In other words, every local event satisfies exactly one intention. The *consistency* condition requires that the intentions in I continue to exist in I' unless otherwise dictated by the occurrence of the set of simultaneous events $\langle \pi_1 : t' \rangle \langle \pi_2 : t' \rangle \ldots \langle \pi_m : t' \rangle$.

We use the notation $(\theta, I) \xmapsto{\langle \pi_1, \pi_2 \ldots \pi_m : t' \rangle} (\theta', I')$ to denote the *direct status succession* from (θ, I) to (θ', I') due to the simultaneous firing of $\langle \pi_1 : t' \rangle$, $\langle \pi_2 : t' \rangle$, ..., $\langle \pi_m : t' \rangle$. Also, we use $(\theta, I) \xmapsto{\alpha} (\theta', I')$ to denote the *status succession* from (θ, I) to (θ', I') due to a number of direct status successions.

A TRA is said to have reached a *stable status* $(\hat{\theta}, \hat{I})$, if all entries of the intention vector are empty ($\hat{I} = I_\phi$). A TRA remains in a stable status until excited by an input event. This follows directly from the causality requirement for a status succession.

To start executing, a TRA $(\Sigma, \sigma_0, \Pi, \Theta, \Lambda, \Upsilon)$ is put in a stable *initial status* (θ_0, I_0), where $I_0 = I_\phi$ and $\theta_0 \in \Theta$. The execution is initiated at time t_0 with the firing of an action π_0 on the start channel σ_0, where $\pi_0 \in \Pi(\sigma_0)$. An execution e of a TRA is a possibly infinite string of alternating statuses and events, which starts with an initial status followed by an initiating event, and which contains an infinite number of status successions (infinite execution), or terminates in a stable status (finite execution).

We follow an approach similar to that adopted in [18] by defining β to be a *behavior* of a TRA $\mathcal{A}$, if it consists of all the *external* events appearing in some execution e of $\mathcal{A}$. We denote the set of all the possible behaviors of a TRA $\mathcal{A}$ by $behs(\mathcal{A})$. Obviously, $behs(\mathcal{A})$ describes all the possible interactions that the TRA $\mathcal{A}$ might be engaged in, and, therefore, constitutes a complete specification of the system that $\mathcal{A}$ models.

A TRA $\mathcal{A}$ is said to *implement* another TRA $\mathcal{B}$ if every behavior of $\mathcal{A}$ is a behavior of $\mathcal{B}$. In other words, all of $\mathcal{A}$'s behaviors (the implementation) are possible behaviors of $\mathcal{B}$ (the specification). The reverse, however, is not true. There might exist behaviors of $\mathcal{B}$ that cannot be generated by $\mathcal{A}$. The notion of a TRA implementing another is used mainly in verification.

2.6 TRA Composition

A basic aspect of the TRA model is its capability to model a complex system by operating on simpler system components. In this section we examine such an operation, namely composition. Other operations (for example hiding and renaming) were presented in [7].

The composition of a countable collection of *compatible* TRAs, $\{\mathcal{A}_i : i \in \mathcal{I}\}$, is a new TRA $\mathcal{A} = \mathcal{A}_0 \times \mathcal{A}_1 \times \ldots \times \mathcal{A}_i \times \ldots = \Pi_{i \in \mathcal{I}} \mathcal{A}_i$. The execution of $\mathcal{A}$ involves the execution of all its components $\mathcal{A}_{i \in \mathcal{I}}$, each starting from an initial status and observing every external event signaled by either the environment (input) or by any TRA in the collection $\{\mathcal{A}_i : i \in \mathcal{I}\}$. The *compatibility* condition for composition insures that, for each channel in the composition, there is at most one writer, a finite number of readers, and that the signaling ranges of readers and writers are compatible.

The input signature of the composed TRA consists of those channels that are inputs to one or more of the component TRAs, and which are not outputs of any of the component TRAs. The output signature of the composed TRA consists of all the outputs of all the component TRAs. Similarily, the internal signature of the composed TRA consists of all the internal channels of all the component TRAs. The start channel of the composed TRA is the start channel of one or more of its component TRAs.[5] The signaling range function of the composed TRA is defined so as to preserve its input-enabled property. In particular, the signaling range of an input channel consists of only those actions that can accepted by all readers of that channel. A computational step of the composed TRA is necessarily a step of one of its components. Similarily the time-constrained causal relationships of the composed TRA are exactly those of the component TRAs.

In [7], the formal construction of the sextuple representation of a composition is given. Also, the relationships between the behaviors and spatial properties of the composed TRA and those of its constituent TRAs are established. In particular, we prove that the sets of proper, spontaneous, and causal TRAs are closed under composition.

The TRA composition operation is more general than those reported in [18, 25, 3] in that it allows the specification of both *parallel* and *sequential* composition. In particular, the introduction of the *start channel* permits the execution of two TRAs to be concurrent if they share the same start channel, or to be serialized if the start channel of one (child) is an output of the other (parent).

3 *CLEOPATRA* Specifications

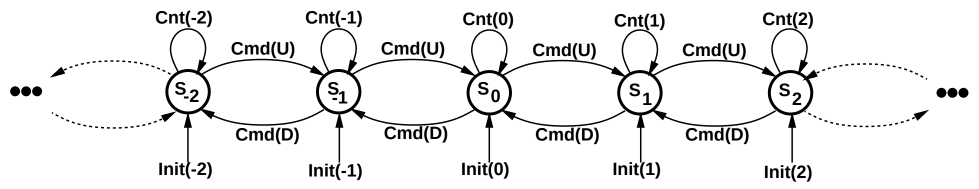
In *CLEOPATRA*, systems are specified as interconnections of TRA objects. Each TRA object has a set of *state variables* and a set of *channels.* Time-constrained causal relationships between events occuring on the different channels, and the computations (state transitions) that they trigger, are specified using *Time-constrained Event-driven Transactions* (TETs). The behavior of a TRA object is described using TETs. TRA objects can be composed to specify more complex TRAs.

The correspondence between *CLEOPATRA* and the TRA formalism is straightforward. Every object in *CLEOPATRA* corresponds to a TRA sextuple. In [7], the construction of a TRA sextuple, given a *CLEOPATRA* object, is detailed.

3.1 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. Figure 2 shows a BNF-like description of a TRA object in *CLEOPATRA*.

[5] Without loss of generality, we assume that TRA to be $\mathcal{A}_0$.



- $\Sigma = \Sigma_{\text{in}} \cup \Sigma_{\text{out}} \cup \Sigma_{\text{int}}$, where: $\Sigma_{\text{in}} = \{\texttt{cmd}, \texttt{init}\}$, $\Sigma_{\text{out}} = \{\texttt{cnt}\}$, and $\Sigma_{\text{int}} = \phi$.
- $\texttt{init} \in \Sigma_{\text{in}}$ is the start channel.
- $\Pi(\texttt{init}) = \mathcal{Z}$, $\Pi(\texttt{cmd}) = \{\texttt{Up}, \texttt{Down}\}$, and $\Pi(\texttt{cnt}) = \mathcal{Z}$.
- Θ, the set of states is given by: $\{\theta_i : i \in \mathcal{Z}\}$.
- $\Lambda = (\bigcup_{i,j\in\mathcal{Z}}\{(\theta_i, \texttt{init}(j), \theta_j)\}) \cup (\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{cmd(UP)}, \theta_{i+1})\}) \cup (\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{cmd(Down)}, \theta_{i-1})\}) \cup (\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{cnt}(i), \theta_i)\})$.
- $\Upsilon = \{(\texttt{init}, \texttt{cnt}, [1.9, 2.1], \phi), (\texttt{cnt}, \texttt{cnt}, [1.9, 2.1], \phi)\}$.

Figure 1: TRA-specification of up/down counter.

```
<tra-object> := <tra-header> '{' <tra-body> '}'
<tra-header> := 'TRA-class' <tra-name> {'(' <tra-params-spec> ')'} <signature>
<tra-params-spec> := {<type> <param-id> {';' <tra-params-spec>}}
<signature> := {<ch-list-spec>} '->' {<ch-list-spec>}
<ch-list-spec> := <ch-id> ( <type> ) {',' <ch-list-spec>}
<type> := 'int' | 'double' | 'bool' | ...
<tra-body> := {<declarations>} {<init>} {<transactions>}
<declarations> := {<state>} {<internal>} {<included>}
<state> := 'state:' <state-var-def>
<state-var-def> := <type> <var-list-def> ';' {<statevar-def>}
<var-list-def> := <var-id> {'=' <constant-exp>} {',' <var-list-def>}
<internal> := 'internal:' <signature>
<included> := 'included:' <included-objects>
<included-objects> := <tra-instantiation> ';' {<included-objects>}
<tra-instantiation> := <tra-name> {'(' <actual-param-list> ')'} <ext-binding>
<actual-param-list> := <constant-exp> {',' <actual-param-list>}
<ext-binding> := {<ch-list>} '->' {<ch-list>}
<ch-list> := <ch-id> {',' <ch-list>}
<init> := <code>
<transactions> := {<xact> {<transactions>}}
<xact> := <xact-header> ':' <xact-body>
<xact-header> := {<trigger-list>} '->' <out-sig-spec>
<trigger-list> := <in-sig-spec> {',' <trigger-list>}
<in-sig-spec> := <ch-id> '(' {<var-id>} ')'
<out-sig-spec> := <ch-id> '(' {<exp>} ')'
<xact-body> := <act> | '{' <acts> '}'
<acts> := <act> {<acts>}
<act> := <computation> | {<condframe>} <fire-acts> | {<timeframe>} <fire-acts>
<computation> := 'commit' '{' <code> '}' | 'do' '{' <code> '}'
<condframe> := 'unless' '('<cond>')' | 'while' '('<cond>')'
<timeframe> := <closed-timeframe> | <open-timeframe>
<closed-timeframe> := 'within' '['<constant-exp>'~'<constant-exp>']'
<open-timeframe> := 'before' <constant-exp> | 'after' <constant-exp>
```

Figure 2: Partial Syntax of a TRA specification in *CLEOPATRA*

In *CLEOPATRA*, TRAs are defined in *classes*. For example, Figure 3 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 3: Integration using the trapezoidal rule.

TRA classes are parametrized. For instance, the specification of `integrate` given in Figure 4 includes the parameters `TICK`, and `TICK_ERROR`, which have to be specified before *instantiating* an object from that class.

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 3 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. For example, in the definition of the `integrate` TRA-class given in Figure 3, 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 Time-constrained Event-driven Transactions in the `act:` section. The behavior of an object belonging to the TRA-class `integrate` shown in Figure 3 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 4, the class `ramp` is defined by composing the `integrate` and `constant`[6] classes together.

[6]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.

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

Figure 4: *CLEOPATRA* specification of a ramp generator.

3.2 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 5 explains the relation between the triggering and firing of actions using TETs.

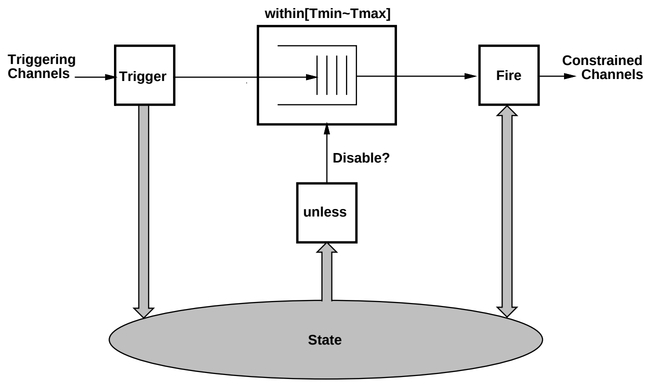


Figure 5: Time-constrained Event-driven Transaction.

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 shown in Figure 3 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` 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 boolean expression (predicate) on the state of the TRA.[7] 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, an intended 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. Only constant expressions are allowed to be used in the specification of time bounds. Open, closed, and semi-closed time intervals can be used provided they specify an interval of time from the set $\mathcal{D}$.[8] 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.

3.3 An Example

Figure 6 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.

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.[9]

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 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 6: *CLEOPATRA* specification of a finite FIFO delay element.

[7]No side effects are permitted in the evaluation of this condition.

[8]Current *CLEOPATRA* processors accept only dense intervals of three forms: $(0, T_u)$, (T_l, ∞), or $[T_l, T_u]$, where $T_u > T_l \geq 0$. These are introduced using the `before`, `after`, and `within` clauses, respectively.

[9]In other words, between input and output transitions.

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 6. 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`. Notice that the value of the action signaled on a local (output or internal) channel does not reflect the state change associated with it. For instance, in the fourth TET of Figure 6, 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`.

3.4 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()`[10], or else wait for at least `DLY_MIN` before issuing a new input. Such safety conditions can be verified using TRA-based verification techniques [7].

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 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 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 [29].

[10]An `ack()` event is signaled after the input is processed.

4 *CLEOPATRA* Simulation

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 sensori-motor robotics applications [8] and to simulate complex behaviors of autonomous creatures [5]. Figure 7 shows the different stages involved in the compilation and execution of specifications written in *CLEOPATRA*.

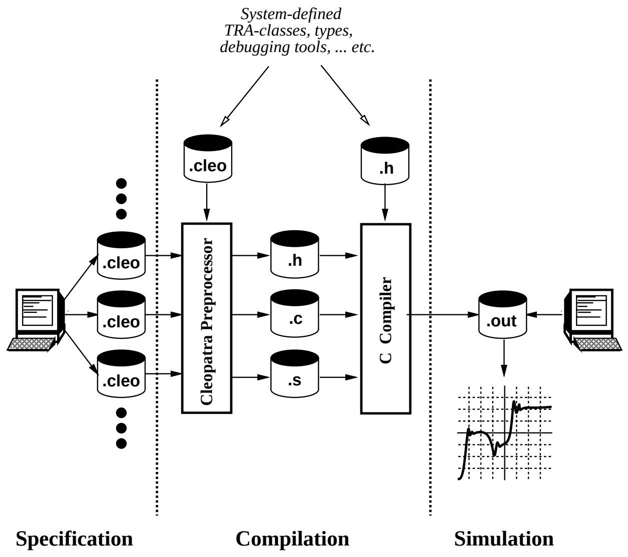


Figure 7: Compilation & simulation of *CLEOPATRA*.

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,[11] 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. Figure 10 illustrates a typical session, in which the *CLEOPATRA* compiler `ccleo` is invoked to process the file `process-ctrl.cleo` containing the specification of the stand-alone process control system shown in Figures 8 and 9.

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

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 9 represents the *CLEOPATRA* specification of the closed process control system shown in Figure 8. The execution of a *CLEOPATRA* stand-alone system is started by instantiating an object from the TRA-class `main` at time[12] 0 and, thereafter, committing only the legal transitions dictated by the system specification and the semantics of the TRA model. Figure 11 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 9, 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.

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. An example of the use of a C-function is illustrated in the description of the `user` TRA-class of Figure 9 where the function `random()` is called periodically to generate a random set value.

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.[13] For example, in the *CLEOPATRA* specification of the stand-alone process control system shown in Figure 9, system-defined TRA classes are included using the `#include` directive, and constants are defined using the `#define` directive.

The simulator has proven to be quite efficient. This is due primarily to the causal and compositional nature of the TRA model, which tends 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.[14] 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.[15]

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 [11], or by abstracting segmented resources as was done in the Spring kernel [23]. 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.

[12] The start time of the simulation can be explicitly specified.

[13] Current *CLEOPATRA* processors do not admit conditional compilation.

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

[15] This is the case in the simulation shown in Figure 10, where an almost 5-fold decrease in efficiency can be attributed to the use of the `fmonitor` TRA-class.

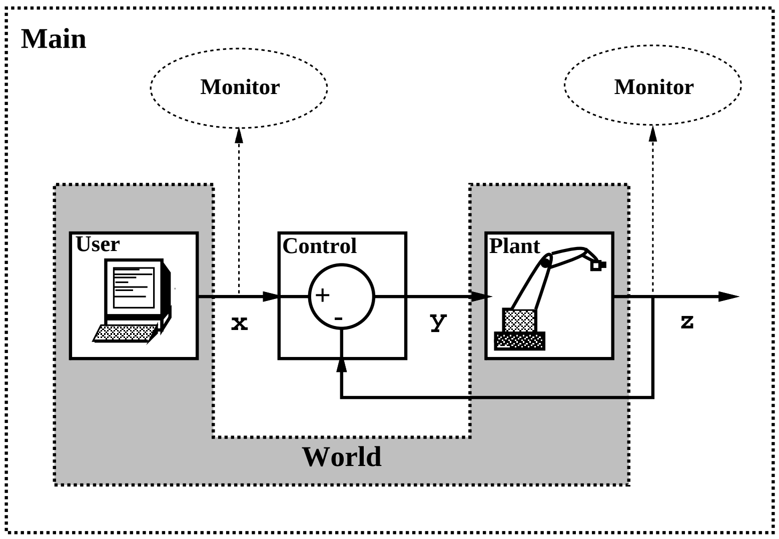


Figure 8: A stand-alone process control system.

```
#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 9: The `main` TRA-class.

```
% ccleo process-ctrl
TRA-class fmonitor(string FILENAME)
  init(unit), signal(double) -> ;
TRA-class user(double EPOCH)
  init(unit) -> x(double) ;
TRA-class plant(double GAIN)
  init(unit), y(double) -> z(double) ;
TRA-class world()
  init(unit), y(double) -> x(double), z(double) ;
TRA-class control()
  init(unit), x(double), z(double) -> y(double) ;
TRA-class main()
  init(unit) -> 'z(double)', 'y(double)', 'x(double)' ;

Cleopatra preprocessing completed.
C compilation completed.

% process-ctrl
CPU time = 1366612 usec   # of events = 5486   SEPS = 4014.3069
```

Figure 10: A typical *CLEOPATRA* compilation and execution session.

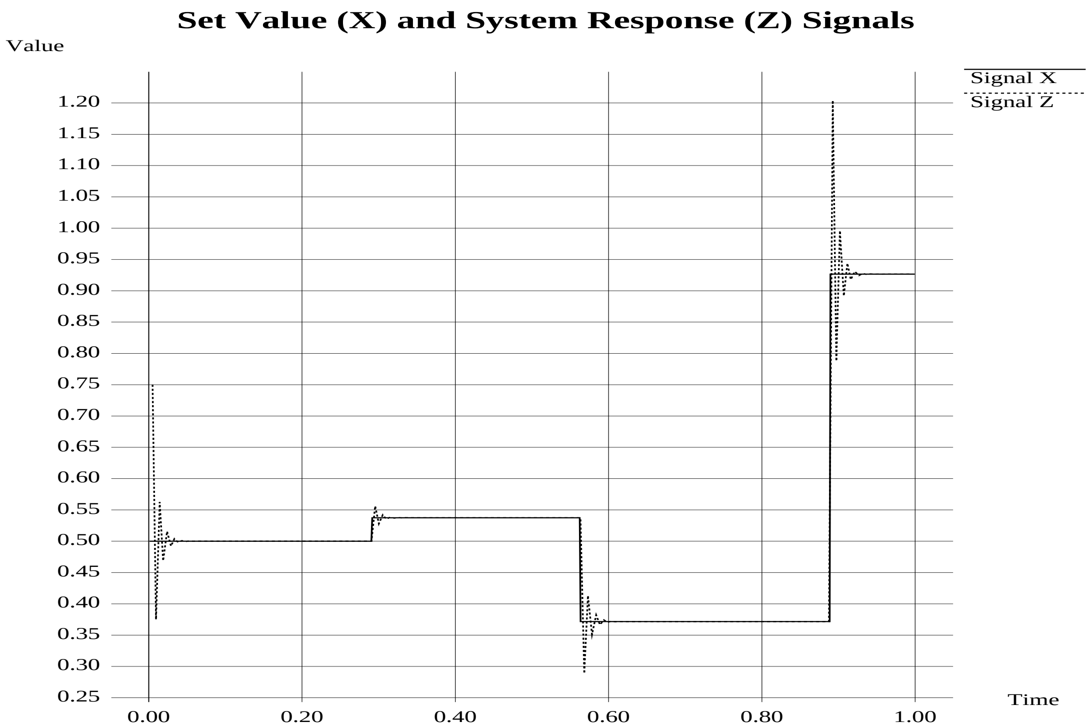


Figure 11: Simulated behavior of an underdamped process control system.

References

[1] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proceedings of the 5th annual IEEE Symposium on Logic in Computer Science*, Philadelphia, Pensylvania, June 1990. IEEE Computer Society Press.

[2] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *Proceedings of TAU'90: The 1990 ACM International Workshop on Timing issues in the Specification and Synthesis of Digital Systems*, Vancouver, Canada, August 1990.

[3] Azer Bestavros. The IOTA: A model for real-time parallel computation. In *Proceedings of TAU'90: The 1990 ACM International Workshop on Timing issues in the Specification and Synthesis of Digital Systems*, Vancouver, Canada, August 1990.

[4] 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).

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Alan Burns and Andy Wellings. *Real-time systems and their programming languages.* Addison Wesley Co. (International Computer Science Series), 1990.

[11] 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.

[12] 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.

[13] K. S. Fu, R. C. Gonzalez, and C. S. G. Lee. *Robotics: Control, sensing, vision, and intelligence.* McGraw-Hill Book Company, 1987.

[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] Harry Lewis. A logic of concrete time intervals. In *Proceedings of the 5th annual IEEE Symposium on Logic in Computer Science*, Philadelphia, PA, June 1990. IEEE Computer Society Press.

[16] Nancy Lynch and Hagit Attiya. Using mappings to prove timing properties. Technical Report MIT/LCS/TM-412.b, MIT, Cambridge, Massachusetts, December 1989. Also in *Proceedings of the 1990 ACM Symposium on Principles of Distributed Computing*, pp. 265-280.

[17] Nancy Lynch and Kenneth Goldman. 6.852 distributed algorithms lecture notes: The I/O Automata. Technical report, Laboratory of Computer Science, MIT, Cambridge, MA, Fall 1988.

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

[19] Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. Unpublished notes, Massachusetts Institute of Technology Laboratory for Computer Science, August 1991.

[20] Fred Schneider. Critical (of) issues in real-time systems: A position paper. Technical Report 88-914, Department of Computer Science, Cornell University, Ithaca, NY, May 1988.

[21] Ramavarapu Sreenivas. *Towards a system theory for interconnected Condition/Event systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, September 1990.

[22] John Stankovic and Krithi Ramamritham, editors. *Hard Real-Time Systems.* IEEE Computer Society Press, 1988.

[23] 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.

[24] D.A. Stuart and P.C. Clements. Clairvoyance, capricious timing faults, causality, and real-time specifications. In *Proceedings of the 12th IEEE Real-time Systems Symposium*, pages 254–263, San Antonio, Texas, December 1991.

[25] Mark Tuttle, Michael Meritt, and Francesmary Modugno. Time constrained automata. MIT/LCS, November 1988.

[26] André M. van Tilborg and Gary M. Koob, editors. *Foundations of Real-Time Computing: Formal Specifications and Methods.* Kluwer Academic Publishers, 1991.

[27] André M. van Tilborg and Gary M. Koob, editors. *Foundations of Real-Time Computing: Scheduling and resource management.* Kluwer Academic Publishers, 1991.

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

[29] Pamela Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, 8(3), May 1982.