On Scalable State-Based Specifications for Real-Time Systems[*]

Alan C. Shaw
Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195
shaw@cs.washington.edu

Technical Report 94-02-03

Abstract

Using our communicating real-time state machine (CRSM) language as a basis, we propose and develop a methodology for specifying requirements and designs for large real-time systems. CRSMs are distributed state machines with novel and general timing facilities and CSP-like synchronous communications. The paper first presents a particular controller-client (CC) architecture for composing CRSMs into larger components and then uses the CC organization to define a number of standard in-the-large paradigms for real-time and other software.

1. Introduction

The general goal is to provide a methodology for the specification of software requirements and designs for *large* real-time systems. Among other features, the approach and techniques should be executable, universal, formal, and scalable. The basis for our work is the communicating real-time state machine (CRSM) notation [Raju 93; Raju & Shaw 92; Shaw 92, 93]. CRSMs are universal state machines with guarded commands as transitions, synchronous IO communications over undirectional channels, and facilities for describing the execution times of transitions and for accessing real-time. CRSMs are distinguished from other state machine models mainly by their explicit timing features.

CRSMs have been tested empirically on a large number of relatively small problems, through paper specifications, computer simulation, monitoring, and (to some extent) verification. Examples include a real-time bounded buffer, calendar timer, mouse clicker recognizer, gate controller for a train crossing, traffic light controller, real-time dining philosophers, and real-time spinning lock algorithms. However, it is evident that some additional methodology is needed to handle larger applications.

This paper makes two contributions towards real-time specifications in-the-large. First, we present a CRSM architecture that permits the construction of larger systems from components. This organizational scheme, called a controller-client architecture, assumes a particular uniform IO channel interface for all CRSMs. Our second contribution is to define a number of standard in-the-large paradigms for real-time and other software. These include conventional control for large components, such as sequential, parallel, guarded selection, and looping; and scalable encapsulated data objects. Also presented are a variety of specific schemes and utilities commonly used mainly in real-time applications. Examples are alarm clock and multicasting utilities, scalable interrupt mechanisms, and organizations for periodic and sporadic activities.

Our approach to scalability has been most influenced by statecharts [Harel 87], with their notions of superstates, interrupts, and series/parallel composition of machines, even though our

[*] This research was supported in part by the National Science Foundation under grant number CCR-9200858.

mechanisms and details are quite different. These differences result from different models (shared memory versus distributed, broadcast communications versus synchronous one-to-one) and our timing facilities. The Modechart notation [Jahanian & Mok 89] is similar to statecharts, but more restricted; they compose nicely but machines are finite state and events that trigger transitions among components cannot have data associated with them. The Requirements State Machine Language (RSML) [Leveson et al. 92], also heavily influenced by statecharts, permits both shared store and distributed interactions among machines, but has no timing features. The Hierarchial Multi-State Machines (HMS) [Gabrielian & Franklin 91] uses aspects of statecharts, Petri-nets, and temporal logic, but seems excessively complex for convenient specification. Another interesting language that provides for hierarchical composition of components is the prototype system description language (PSDL) [Kramer et al. 93], which combines state machine with data flow ideas and includes time; for convenience and simplicity, we prefer a pure state-based notation. Other (non-state-based) models that provide for scalability, for example, those based on programming languages or Petri-nets, are not discussed here.

The next section describes the interfaces and behaviors of machines, and our graphical notation for components and interfaces. Section 3 then gives a brief introduction to CRSMs. The next two sections explain our proposed controller-client architecture for large components and our standard reusable CRSM form. These are used in Sections 6 and 7 which present some standard in-the-large paradigms for compositions and encapsulations and a variety of schemes for real-time software. The last two sections discuss some open problems and issues, and summarize our results.

2. Behaviors, Components, and Interfaces

A real-time system is modeled as a closed world consisting of an external environment and a controlling and monitoring computer system (Figure 2.1). The environment and computer system communicate through objects from a set of inputs *I* and outputs *O*. These input-output (IO) objects are called signals, events, messages, or commands. Elements of *I* are monitored by the computer system and elements of *O* represent control and query messages from the computer.

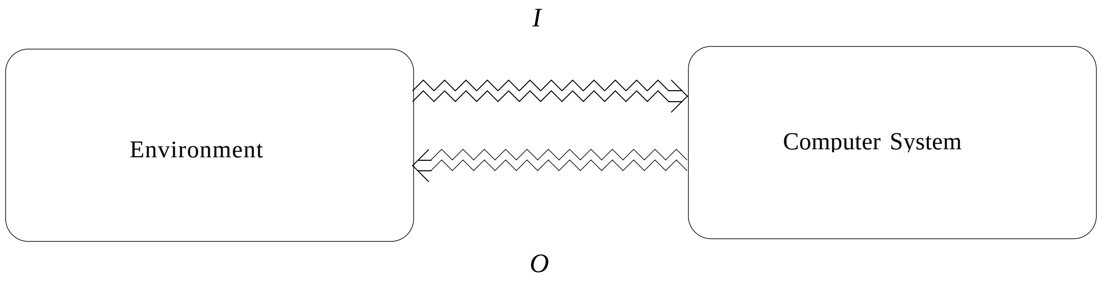


Figure 2.1 Real-Time System Model

The *behavior* of a system is defined as a set *T* of traces over *I* and *O*, where a trace *tr* is a (possibly infinite) sequence of timed IO events:

$$tr = <x_0\ x_1\ \dots\ x_i\ \dots>$$

A timed *IO* event x_i is a triple (e_i, v_i, t_i); e_i and v_i denote an event name and an associated value, respectively, from *I* or *O*, and t_i is the time of the event. The event name can be viewed as a class or as an *IO* channel; the value is the message or data corresponding to the particular instance of the event. For all i, $t_i \le t_{i+1}$ and for each x_i, there exists only a finite number of x_j such that $t_i = t_j$. Generally, the x_i terms from *I* in a trace represent the *given* behavior of the environment over time and the terms from *O* describe the *required* behavior or response of the computer system.

The environment and the computer system are each described by a set of CRSMs and their communicating IO channels. A *closed* system S is given by a pair (M, C), where $M = \{M_1, M_2, \ldots, M_m : m \geq 2, M_i$ a CRSM$\}$[1] and $C=\{C_1, C_2, \ldots, C_k : k \geq 0, C_i$ a channel$\}$. Each channel C_i consists of a name, a type denoting the values of the messages that can be transmitted on the channel, and an ordered pair (M_s, M_r), $s \neq r$, listing the sender and receiver CRSMs. Graphically, a machine "node" is drawn as a named rectangle with rounded corners, a channel as a labeled wavy arrow "edge" directed from the sender to the receiver machine, and a system as a graph of machine nodes connected by channel edges.

Example:

Consider a (simplified) real-time system for controlling traffic lights at the intersection of an avenue and street; in addition to controlling the normal light sequencing, the computer system must respond appropriately to the arrival and departure of an ambulance on one of the thoroughfares. This is a variation of the example presented in [Raju & Shaw 92]. The environment consists of two pairs of traffic lights, one for the avenue and one for the street, and the ambulance. Outputs are commands to turn the lights to their correct colors. Inputs are from the ambulance and comprise a signal indicating the approach of an ambulance on either the street or the avenue and a message that is sent when the ambulance leaves the intersection. The traffic control system *TC* can be specified:

TC = {{Ambulance, Street_Lights, Avenue_Lights, Light_Mode_Control, Light_On/Off_Control}, {Approach, Leave, Avenue, Street, Normal_Mode, Amb_Mode}}

where the channels are:

Channel Name	Message Type	(Sender,Receiver)
Approach	<thoroughfare>	(Ambulance, Light_Mode_Control)
Leave	<null>	(Ambulance, Light_Mode_Control)
Avenue	<color>	(Light_On/Off_Control, Avenue_Lights)
Street	<color>	(Light_On/Off_Control, Street_Lights)
Normal_Mode	<thoroughfare>	(Light_Mode_Control, Light_On/Off_Control)
Amb_Mode	<thoroughfare>	(Light_Mode_Control, Light_On/Off_Control)

The system is illustrated in Figure 2.2

In a closed system $S = \{M, C\}$, there are no dangling or unconnected channels in C: every channel has both a sender and receiver machine in M. Such a system generally consists of a number of open subsystems which in isolation have unconnected channels. In an open system $S = \{M, C\}$, the definition of at least one channel $C_i \in C$ has an ordered pair (M_j, M_k) where exactly one of M_j, M_k is undefined.

An in-the-large component is represented as an open subsystem, with the unconnected channels as its *interfaces*. These higher-level components are also denoted graphically by named rectangles with rounded corners and the interface channels by labeled wavy arrows, as illustrated in Figure 2.3. Components can be combined by connecting channels together, provided that the channels are type-compatible. When combining open subsystems in this manner, each new connecting channel must have a unique name, which may involve renaming the channel and all of its IO commands in the sending or receiving CRSM.

[1] $m \geq 2$ because we assume that there is at least one CRSM representing the environment and at least one for the computer system.

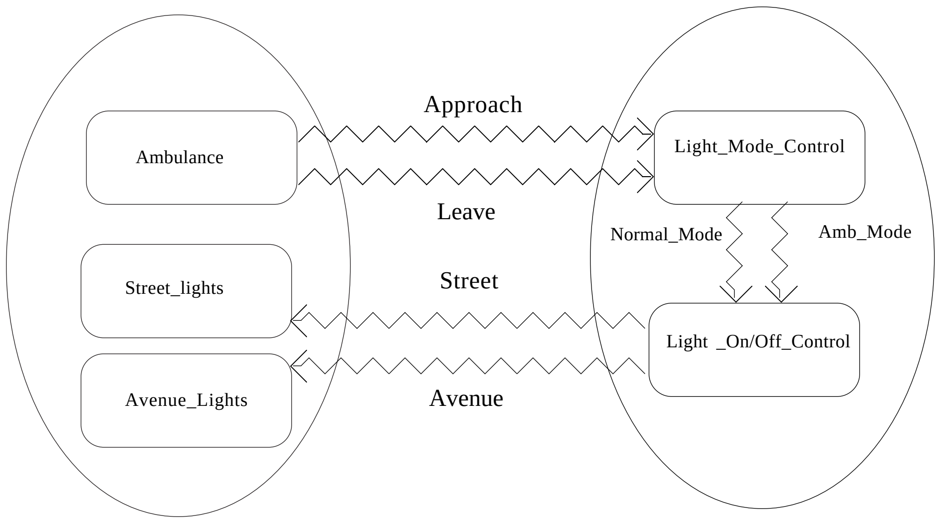


Figure 2.2 Traffic Controller

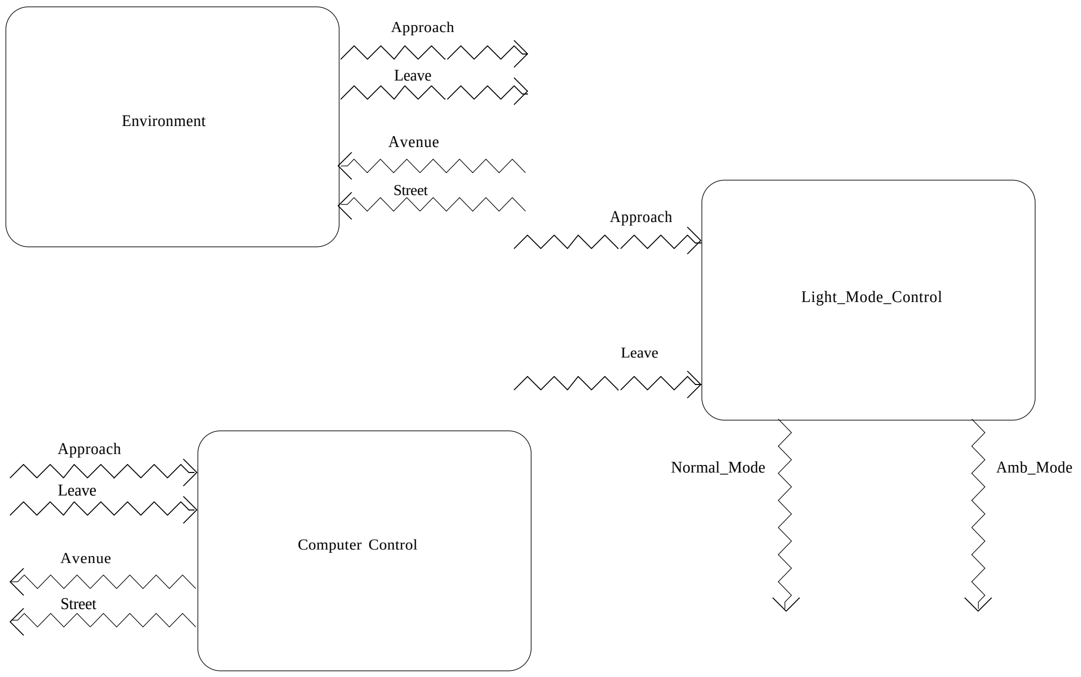


Figure 2.3 Three Open Subsystems From Figure 2.2

3 . Communicating Real-Time State Machines

We present a brief introduction to CRSMs. More formal and detailed treatments appear in [Shaw 92, 93].

A CRSM is a state machine with one designated start state and guarded commands for state transitions. A command can be either an IO command, or an internal one designating a computation or some physical activity. Enabled transitions are fired on an earliest-time-first basis.

Communications between CRSMs is synchronous and occurs over undirectional named channels, in a manner similar to CSP [Hoare 85]. A receiving machine desiring input on a channel *C* may issue an *input* command:

C(x)?

A sender machine may have a corresponding *output* command on the same channel:

C(message)!

When and if communication occurs, the data transmitted by the sender is instantaneously received by the receiver, equivalent to the assignment:

x := message

Both sender and receiver then continue execution.

Guarded commands have the general form:

$g \rightarrow COM[t_1, t_2]$

where *g* is a Boolean guard, *COM* is a command, and $[t_1, t_2]$, $t_1 \leq t_2$, gives a time bound for executing the transition. If *COM* is an internal command and the transition is selected for execution, then its execution time *d* is somewhere in the time-bound window, i.e., $0 \leq t_1 \leq d \leq t_2$. If *COM* is an IO command selected for execution, then t_1 and t_2 give the earliest and latest times, respectively, that the IO can occur, relative to the time that the machine last entered its current state. If IO occurs, it happens at the earliest time that both sender and receiver can execute the command.[2]

Finally, every machine *X* has its own real-time clock machine RTC_X that can send current *global* real-time (denoted *rt*) to *X* on demand. RTC_X has one state with a single transition (to itself) on a clock channel RT_X :

RT_X (rt)! $[0, \infty]$

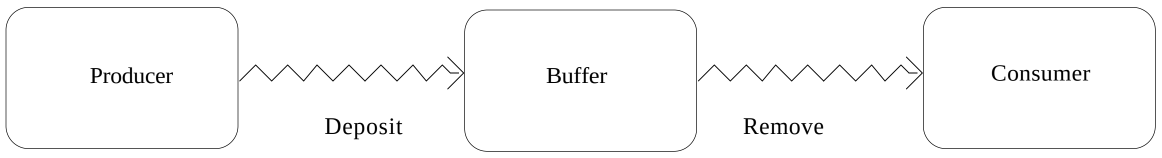
The host machine *X* can obtain current real-time with a "time-out" at time *t* with the command:

RT_X (crt)? [t, t]

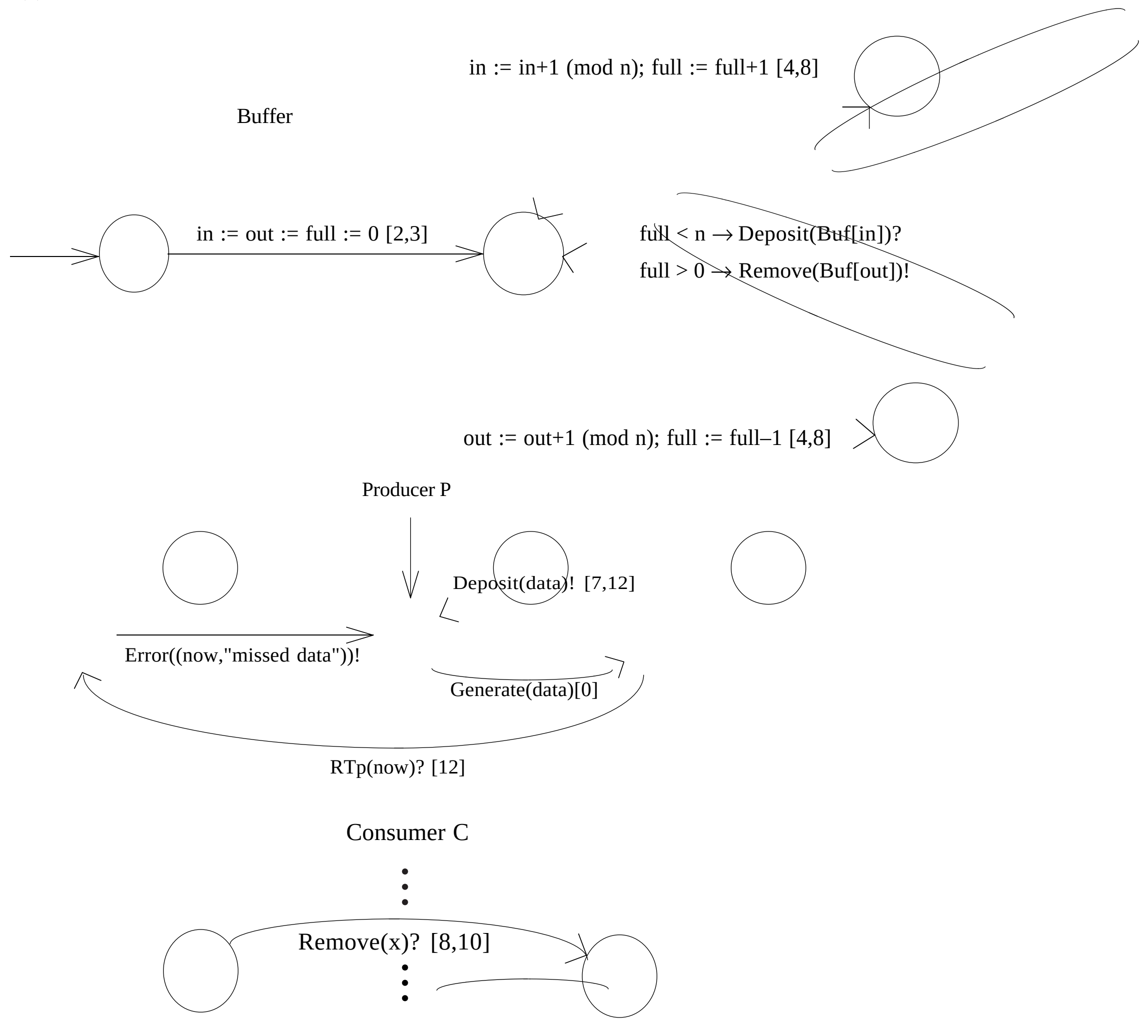
Example: Real-Time Bounded Buffer

Figure 3.1 contains 3 CRSMs implementing a real-time version of a producer and a consumer interacting through a bounded buffer. We assume that the producer is a physical device that generates input signals quasi-periodically in a cycle that ranges between 7 and 12 time units. If the Buffer is unable to accept the data in time, the Producer times-out (RT_P (now)?) and sends a message on the *Error* channel to some fault-handling or monitoring machine (not shown). Abbreviations for two common cases of time-bounds are used: if the bounds are $[0,\infty]$, then the interval is omitted; if the bounds are identical, e.g., [*t*,*t*], then a simple scalar [*t*] is employed. The intervals for the compute transitions in Buffer and P ([2,3], [4,8], and [0]) were chosen arbitrarily for this example.

2 Every interval $[t1, t2]$ is interpreted as $[t1 + \delta, t2 + \delta]$, where δ is a small number representing a minimal amount of time that will always be spent in every state.



(a) Buffer and Clients



Buffer
in := in+1 (mod n); full := full+1 [4,8]
in := out := full := 0 [2,3]
full < n → Deposit(Buf[in])?
full > 0 → Remove(Buf[out])!
out := out+1 (mod n); full := full–1 [4,8]
Producer P
Deposit(data)! [7,12]
Error((now,"missed data"))!
Generate(data)[0]
RTp(now)? [12]
Consumer C
Remove(x)? [8,10]

(b) CRSMs

Figure 3.1 Real-Time Bounded Buffer Example

4. A Standard Reusable CRSM Form

All machines will have a standard interface. Our basic reusable machine contains an input *start* channel for initiating execution, an output *stop* channel for signaling termination, and an arbitrary number of input and output channels as shown in Figure 4.1 (a)[3]. The *start* channel has an optional typed input parameter *x* and the *stop* channel has an optional output *y*. Figure 4.1 (b)

3 Not shown is the clock machine RTC_M that is associated with every machine *M*.

indicates how these channels are used in the rest of the CRSM; guards (not shown) may appear on the *stop* transitions.

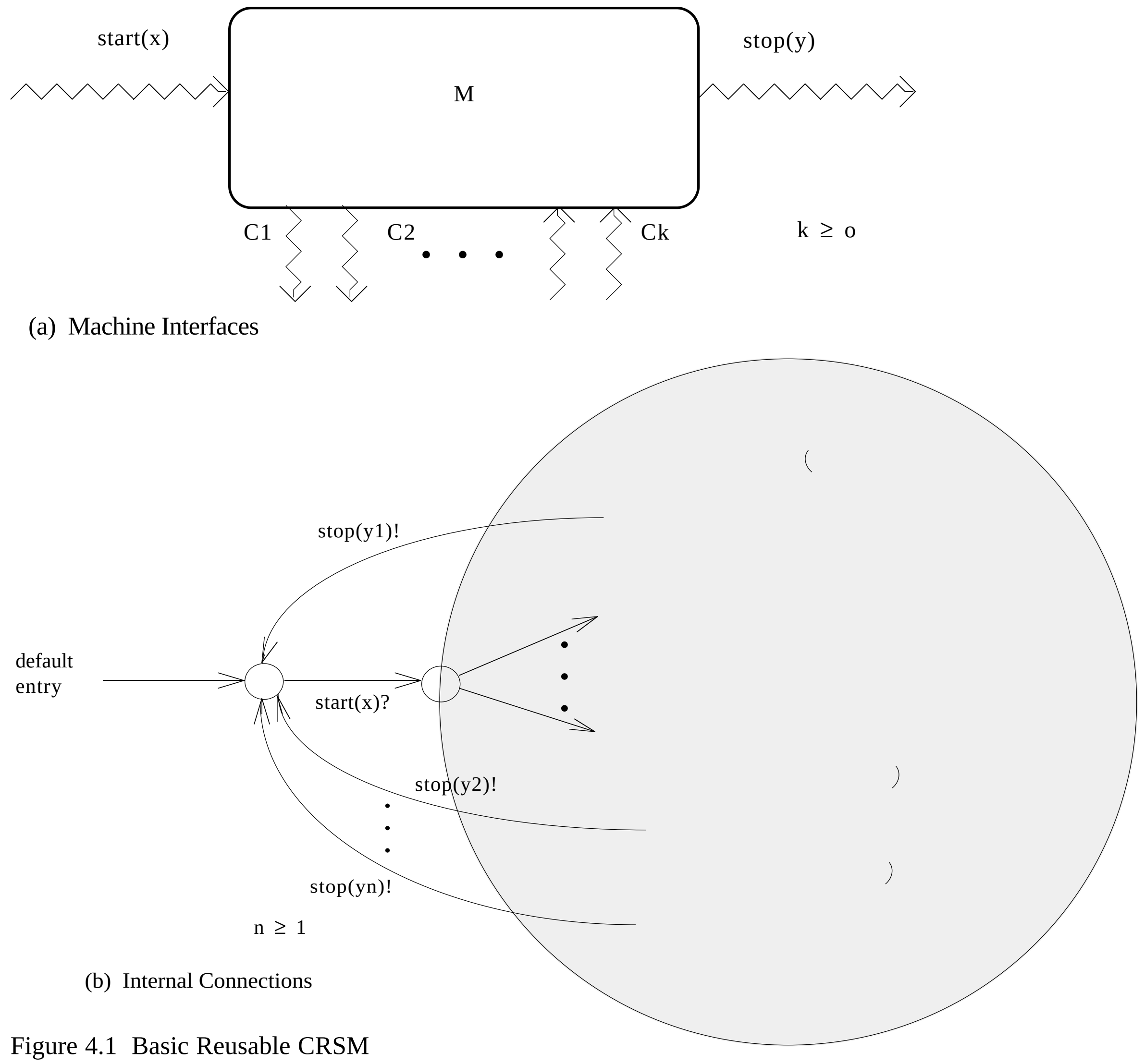


Figure 4.1 Basic Reusable CRSM

To control such a machine, say *M*, another machine would first issue a start command, i.e., send a message over its *start* channel with input, say *a*

M. start(a)!

and then may wait for it to terminate by issuing a stop command

M. stop(b)?

receiving output in *b*. We will use a dot notation, "*M*." above, to distinguish *start*, *stop*, and other identically-named channels on different machines.

An example of a reusable machine is the periodic ticker machine *W* specified in Figure 4.2. *W* issues a *Tick* every *p* units of time after startup, until the current time (*ct)* exceeds *n* which is provided at startup. The *Tick* is ignored unless a user of *W* is requesting it in the time interval [0,eps] after the *Tick* is issued. It is assumed, arbitrarily, that the compute transitions take 1 unit of time; for correct operation, we also assume that *p*>1, ensuring that (*next*-*ct*)>0.

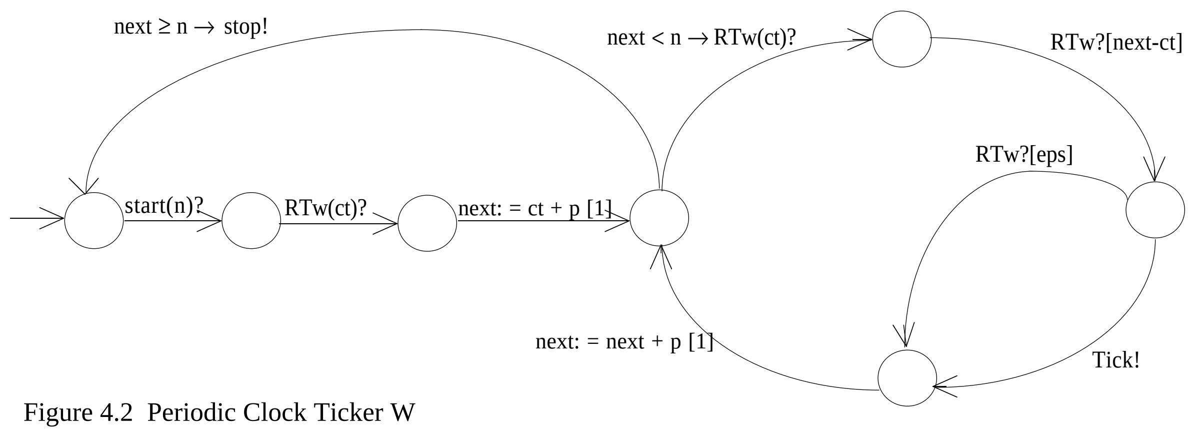


Figure 4.2 Periodic Clock Ticker W

Frequently, a controlling machine may wish to enforce or initiate a termination action on its controllee *M*. In such a case, at least one of *M*'s IO channels is an "interrupt" channel; messages received on this channel cause *M* to (eventually) issue a *stop* request. Alternatively the *stop* channel could act as an interrupt channel. (Section 7.2 presents a finer-grained mechanism for interrupting a CRSM more directly).

The CRSMs in the bounded buffer example (Figure 3.1) are not in this reusable form, since they are all missing *start* and *stop* channels. These can be easily added, and in fact, may be necessary if the application is to be completely specified conveniently. In particular, these channels are needed to handle situations such as power failures, and orderly start-up and shutdown.

5. Controller-Client Architecture

Every higher level subsystem will have the same general interface as the reusable primitive CRSM, i.e. *start* and *stop* channels and an arbitrary number of other IO channels (Figure 5.1(a)). An organization, that we call the *controller-client* (*CC*) structure (Figure 5.1(b)), is imposed on these higher level subsystems.

The structure contains a controller machine and $k \geq 1$ client subsystems $M_1, \ldots, M_k$, each of which can be either a basic CRSM or a higher level subsystem. The task of the controller is to initiate the execution of the client subsystems, to control the clients' execution, e.g. with respect to time, and to synchronize the termination of the client machines. Each subsystem set of machines *M* can have an associated clock, namely the clock machine of its controller. The *start* and *stop* channels of *M* are the corresponding channels of *M*'s controller, and will be written as *M*.start and *M*.stop.

Machine composition and refinement are defined in terms of the *CC* architecture. A set of subsystems $\{M_1, \ldots, M_k\}$ are composed by providing an appropriate controller *K*. *K* could consist of more than one CRSM, in general. Conversely, the refinement of a subsystem $M=\{K, M_1, \ldots, M_k\}$ is the set of clients $\{M_1, \ldots, M_k\}$.

The traffic light example described in Section 2 is not in *CC* form. One version of this system in standard form is sketched in Figure 5.2. A side benefit of our conventions, that is not realized in the original, is that this more complete description can provide for orderly startup, shutdown, restarts, and interruptions[4] ·

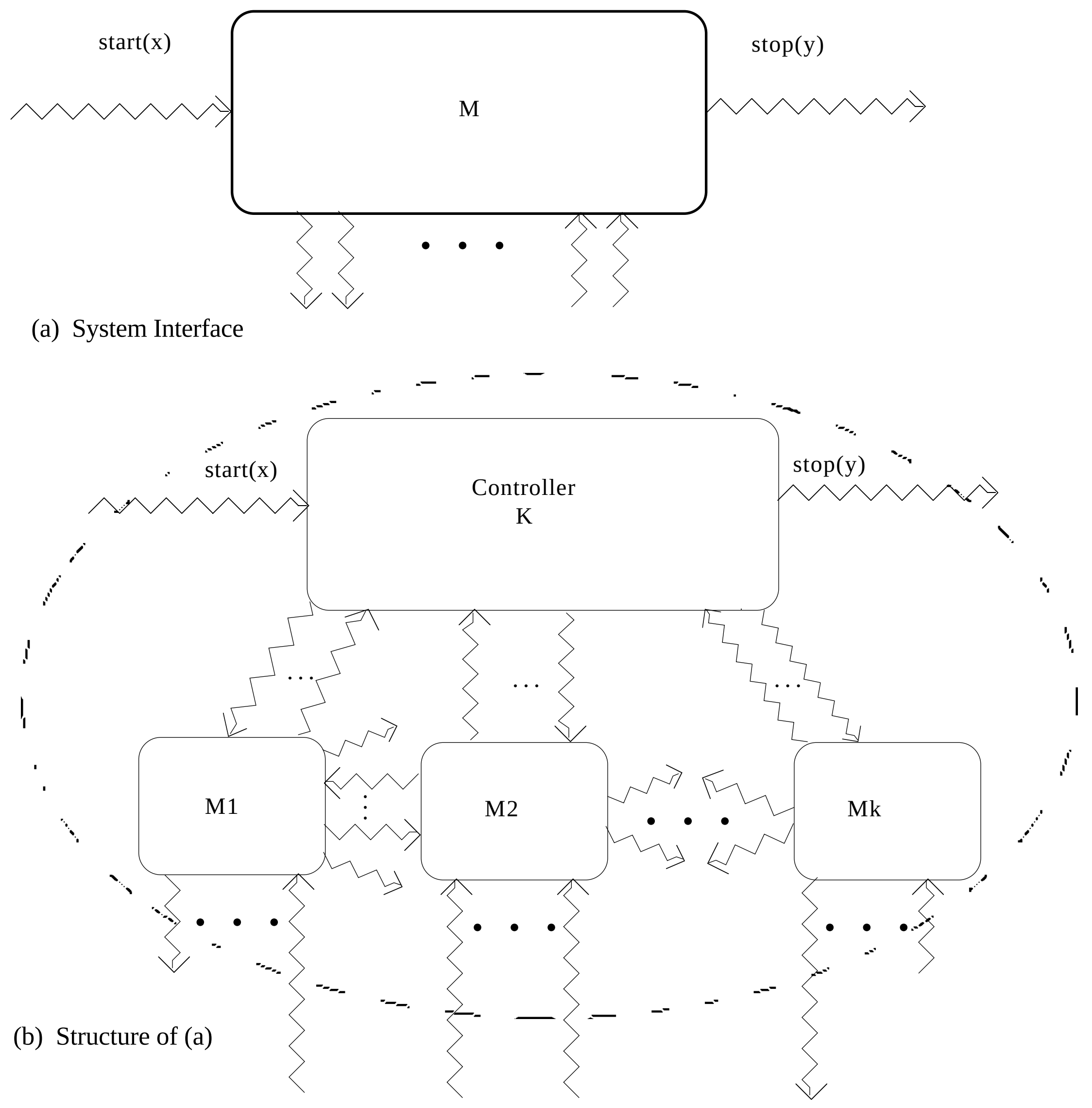


Figure 5.1 CC Architecture for Subsystems

4 There is the issue of how the top level controller in a closed system starts and stops, e.g., the Traffic World Controller in this example. We will assume that some meta-level "super-controller" performs these functions, such as the user of the system.

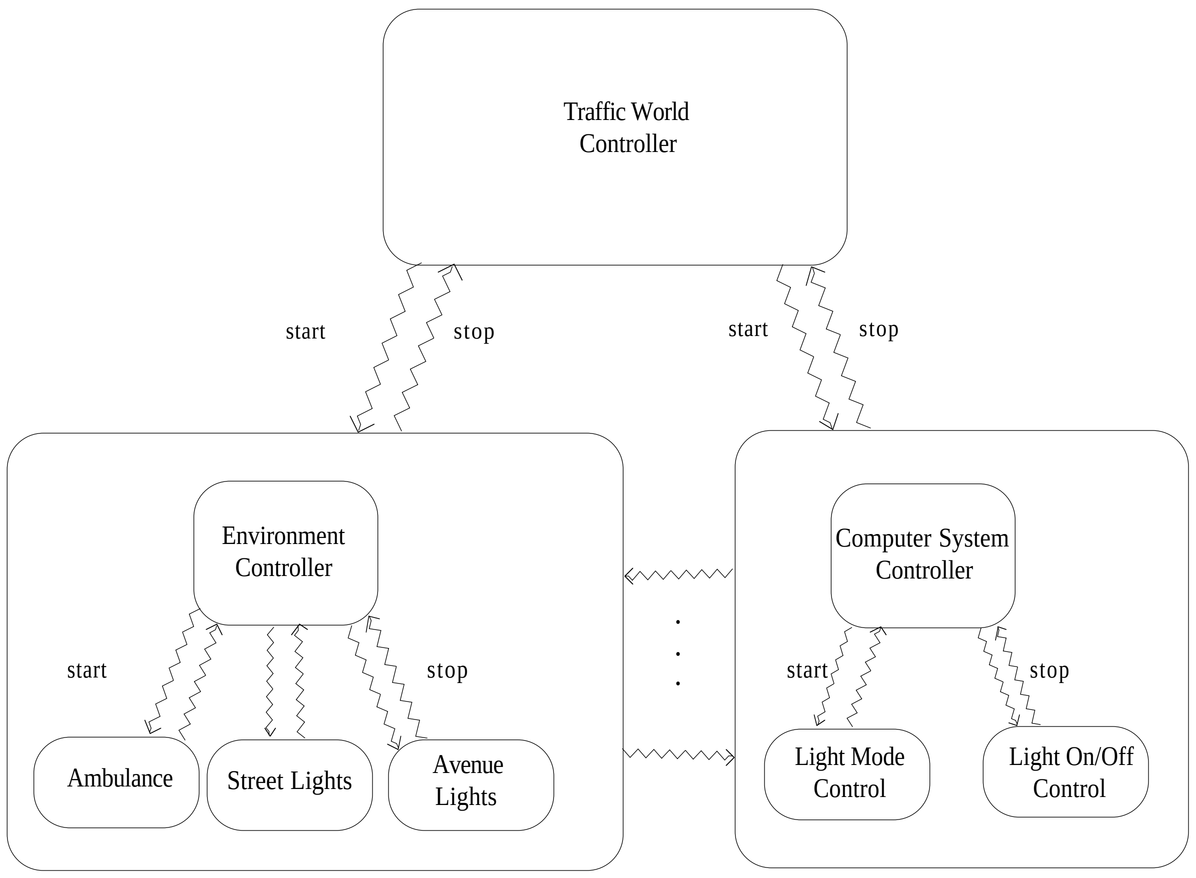


Figure 5.2 CC Structure For Traffic Light System

6. Standard Paradigms for Composition and Encapsulation

Conventional control methods for grouping objects and for encapsulating behaviors are described and implemented within the controller-client framework.

6.1 Conventional Serial and Parallel Machine Compositions

Given two open subsystems M_1 and M_2, each implemented with a *CC* structure, we show how they may be connected sequentially, in parallel, and with guarded selection by providing an appropriate controller K producing a subsystem $M=\{K, M_1, M_2\}$. The methods extend easily to $m \geq 2$ subsystems $M_1, \ldots, M_m$. Cyclic control of a single subsystem is also defined.

For *sequential composition*, the controller K is a small CRSM that sequences through the IO instructions:

M_1. start! M_1. stop? M_2. start! M_2 .stop?

If data is to be passed (not "piped") to M_2 at the termination of M_1, the *stop* and *start* channels of M_1 and M_2 will have parameters and K has the form:

M_1. start! M_1. stop(x)? M_2. start(x)! M_2.stop

In this simplest of cases, the controller could be eliminated completely by identifying the stop channel of M_1 with the start channel of M_2. (Figure 6.1)

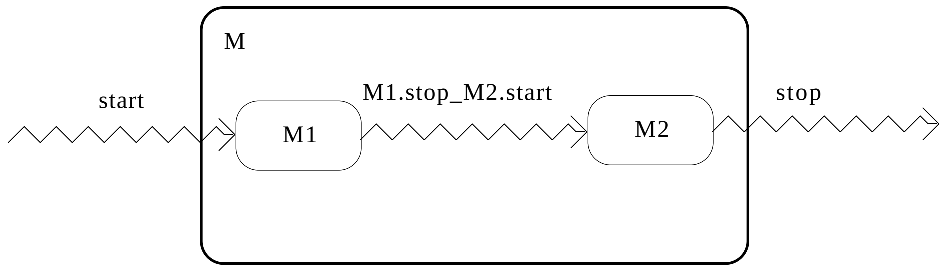


Figure 6.1 Efficient Sequential Composition

A function f from domain X to co-domain Y, i.e., $y = f(x)$ where x in X and y in Y, can be described as a basic CRSM or structure with input on the *start* channel and output on its *stop*. To specify the computation y = f (x) with structure or machine M_f, the "caller" invokes the sequence:

M_f. start(x)! M_f. stop(y)?

Function composition is a particular instance of the sequential case. If the co-domain of a function g is equal to the domain of a function f, their composition $y = f(g(x))$ can be specified by the sequential composition of their machines M_g and M_f.

Parallel composition means that M_1 and M_2 start at (approximately) the same time, perform their functions concurrently, and are synchronized at their termination. It can be implemented using a fork/join form of control. K executes the IO sequence:

M_1. start! M_2.start! M_1.stop? M_2 . stop?

Selection of either M_1 or M_2 depending on the values of Boolean guards g_1 and g_2 is another standard control paradigm. We call this *guarded selection*. The semantics are defined by the controller CRSM in Figure 6.2. Note that because we are working with a distributed model, there are no global variables; the guards $g_1(x)$ and $g_2(x)$ are Boolean functions of the *input* to M obtained though its *start* channel. Guarded selection is non-deterministic - if both g_1 and g_2 are true, one of M_1, or M_2 is selected, non-deterministically.

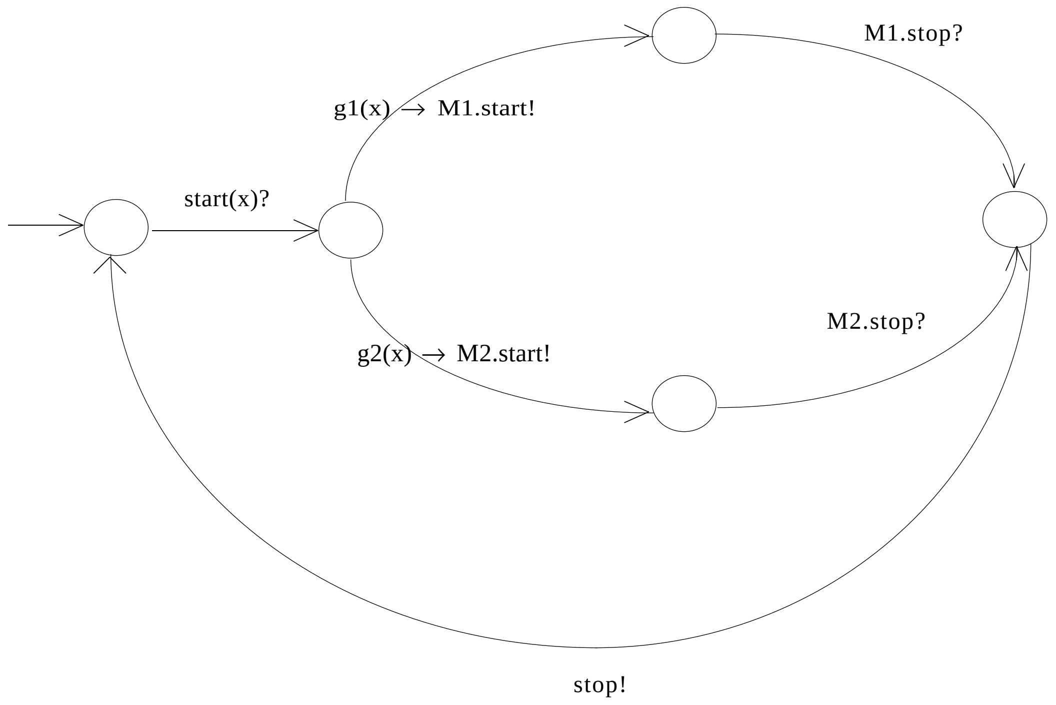
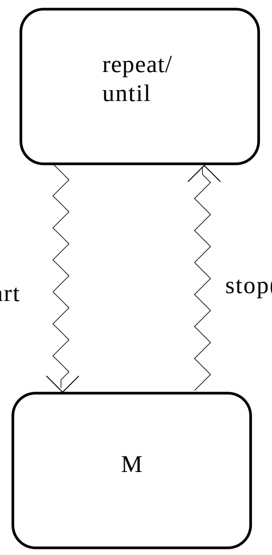
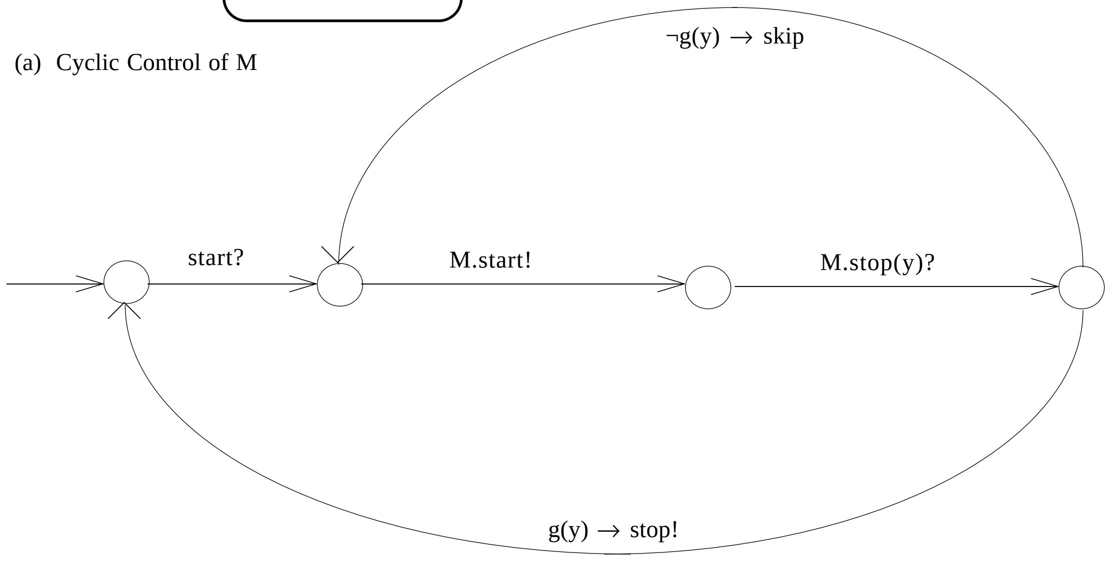


Figure 6.2 Controller for Guarded Selection

Several different forms of *cyclic* control are also useful. For example, a subsystem *M* is to execute repeatedly until a Boolean guard becomes true; this is an in-the-large *repeat*/*until* loop. The controller is given in Figure 6.3.



(a) Cyclic Control of M

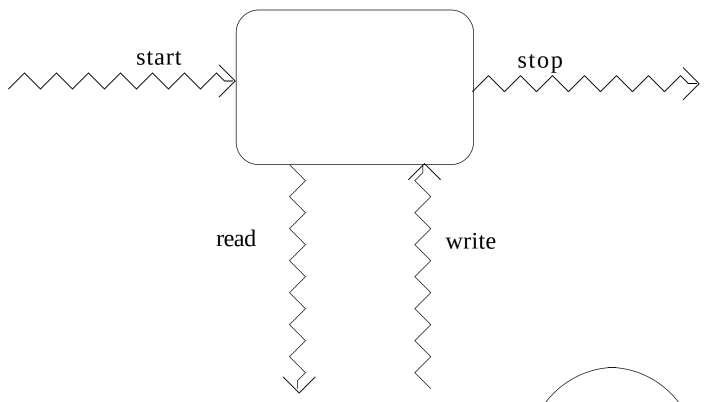


(b) repeat/until Controller CRSM

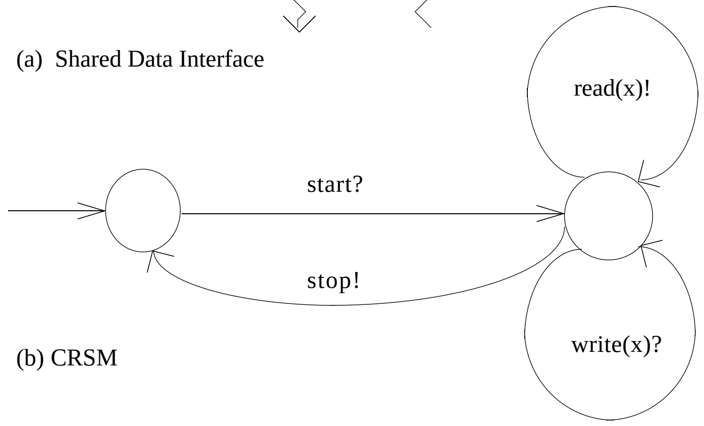
Figure 6.3 Repeat/Until Looping

6.2 Data Encapsulations

The suggested approach for handling *shared* data in our distributed model is through abstract data types (ADTs) that maintain the data state and provide remote procedure call interfaces (methods or operations) for user access and manipulation. A shared data server that implements *read* and *write* operations on a database *x* is specified by the basic CRSM in Figure 6.4.



(a) Shared Data Interface



(b) CRSM

Figure 6.4 Shared Data Server

Of course, if several users wish to read and write the database, we need unique *read* and *write* channels for each user. The standard way to accomplish this is by defining arrays of channels; e.g. read.k , write.k , $k = 1 .. n$ for $n \geq 1$ users.

If access and update at a finer granularity than the entire database is desired, then it is necessary to transmit names or addresses of data at the interface. For example, if data is stored in an array $X(i)$, $i = 1,...,m$, then the data server could execute the following sequences:

Read: read(i)? return(X(i))!

Write: write(i,y)? X(i) := y

A user would call the server with:

Read call: read(i)! return(x)?

Write call: write(i,y)!

The array index *i* serves as the data address.

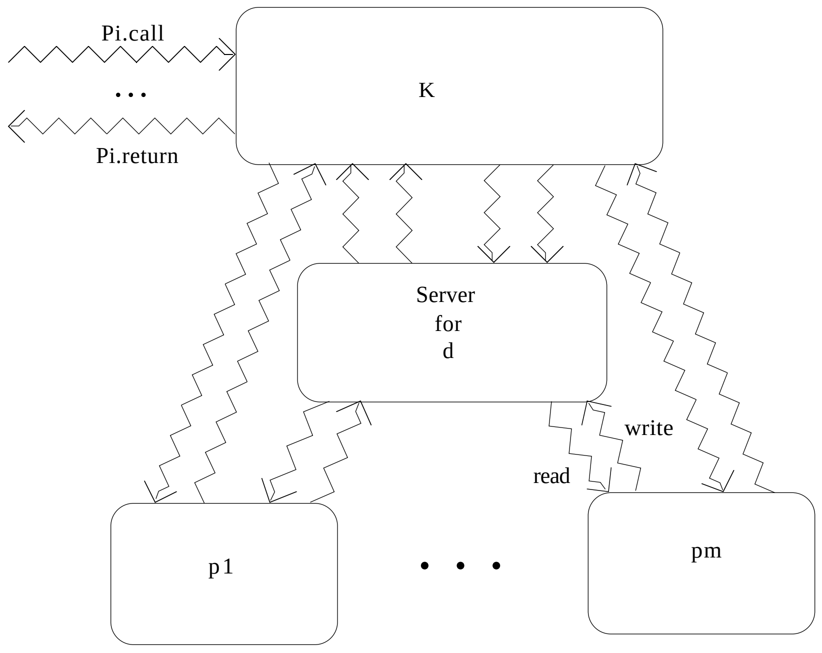
For higher-level ADT servers that implement more general operations, a send/receive protocol over the IO channels can be used to implement the remote procedure call interface. A server *S* that offers operations p_1, p_2, ..., p_m, $m \geq 1$, on shared data *d* can be constructed with the *CC* structure of Figure 6.5. A user invokes a service p_i with the IO calls:

$S.p_i$.call(x)! $S.p_i$.return(y)?

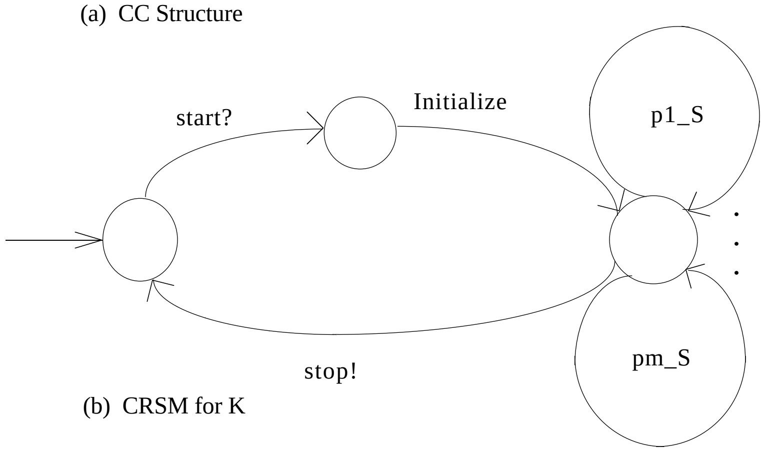
x is the input parameter to *S* and *y* is the output returned from *S*. The p_i sequence (p_i_s) in the controller *K* has the general form

p_i. call(x)? Perform_Requested_Service p_i.return(y)!

A guard may be part of the first command. Also, we have not shown separate channels for each user, as in the other examples above.



(a) CC Structure



(b) CRSM for K

Figure 6.5 General ADT Server

7. Real-Time Paradigms

7.1 Some Time-Constrained Utilities

We describe two different kinds of utilities that are useful in real-time specifications. One family defines alarm clocks for both absolute and relative time. The second provides an alternative communications method, in particular a multicast mechanism for broadcasting messages to a given subset of the components of a system. Both classes have interesting and non-obvious time constraints.

The generic alarm clock has the interface depicted in Figure 7.1. The message in the *Wake_Me* channel gives a non-negative absolute or relative time, say *t* or *Δt*, respectively, at which a *Wakeup* message is to be sent by the clock. If *now* is the time at which the *Wake_Me* communication occurs, then at time t'= max(*now*, *t*) (absolute time) or *t'*= *now* + *t* (relative time), the *Wakeup* message will be enabled on its channel. The *Wakeup* message will remain enabled until either it has been received or the time has reached *t'* + *ring_time*, whichever occurs

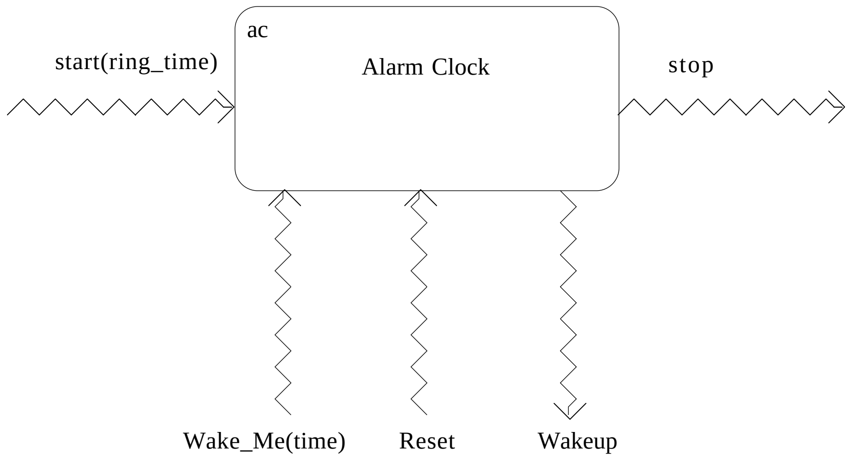


Figure 7.1 Alarm Clock Interfaces

earliest[5] . A (null) message on the *Reset* channel will reset the alarm clock so that it is ready to receive another *Wake_Me* or to terminate. Figures 7.2 and 7.3 contain CRSMs for relative and absolute time alarm clocks, respectively. In these versions, a *Reset* signal is not handled once an alarm clock starts "ringing"; this could be added easily if desired.

The multicast facility allows a signal to be broadcast for a given time interval. During the broadcast interval, any of a set of designated receivers can elect to receive the signal. The interface for a generic multicast system appears in Figure 7.4. The message or signal to be broadcast is sent to the facility over the *multicast* channel. The message will be made available to each receiver over the channels $Receiver_i$, *i*=1, ..., *r*, for the time interval [*now*, *t'*], where *now* is the time at which the multicast message is sent to and received by the multicast facility (assumed identical here) and *t'* = *now* + *transmit_time*.

[5] The minimal transition time δ is being ignored here. The maximum enabling time is really *t'* + *ring_time* + $k\delta$, where *k* = 2 or 3 depending on the alarm clock (Figure 7.2 or 7.3).

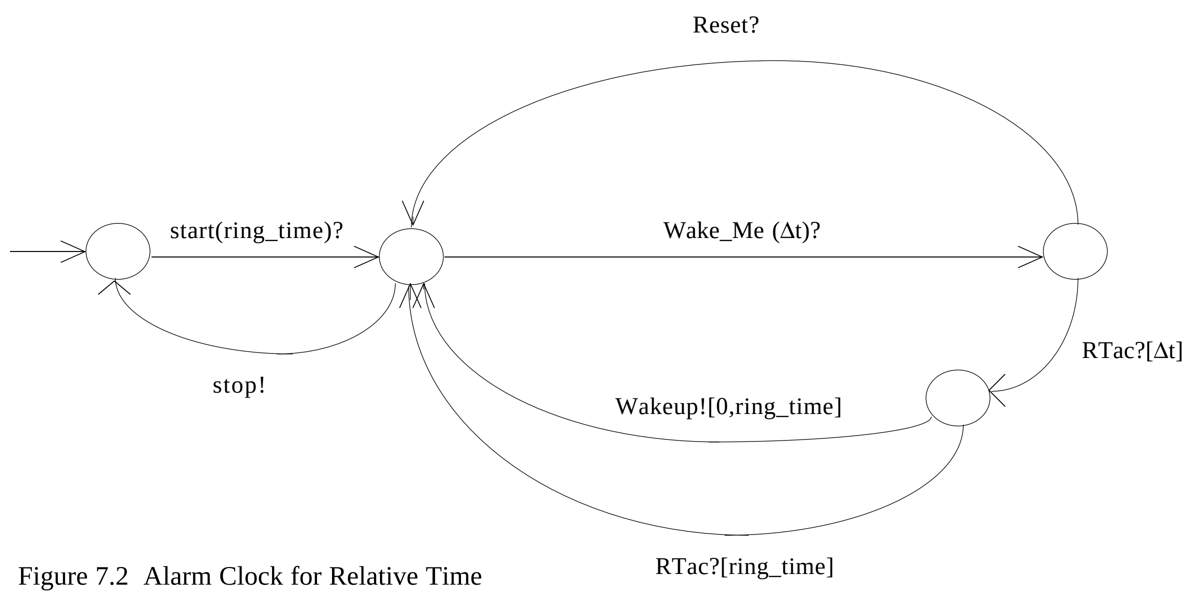


Figure 7.2 Alarm Clock for Relative Time

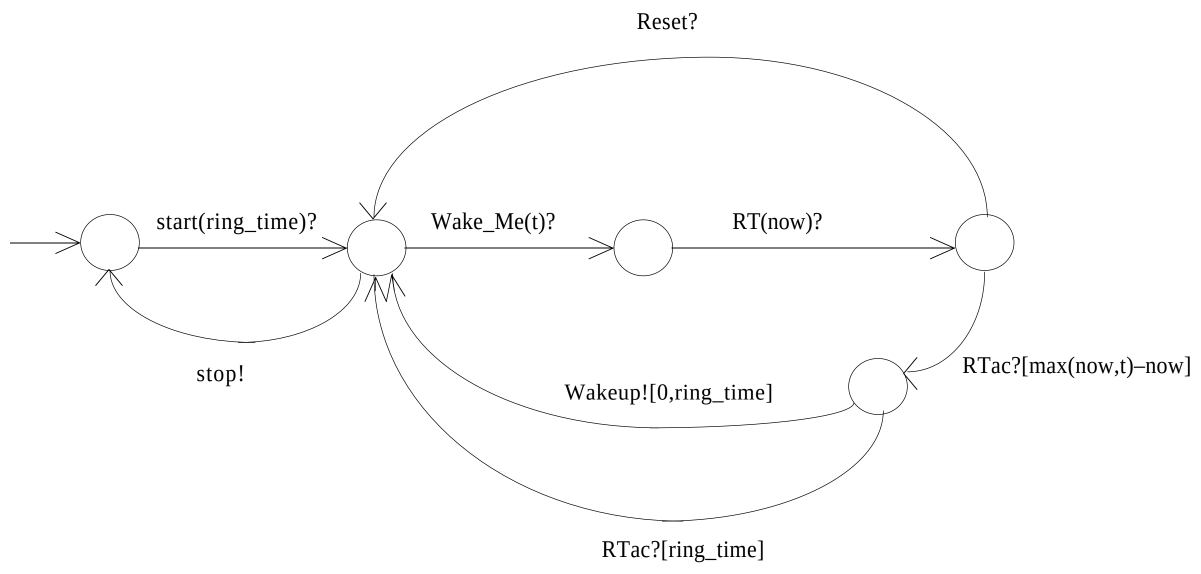


Figure 7.3 Alarm Clock for Absolute Time

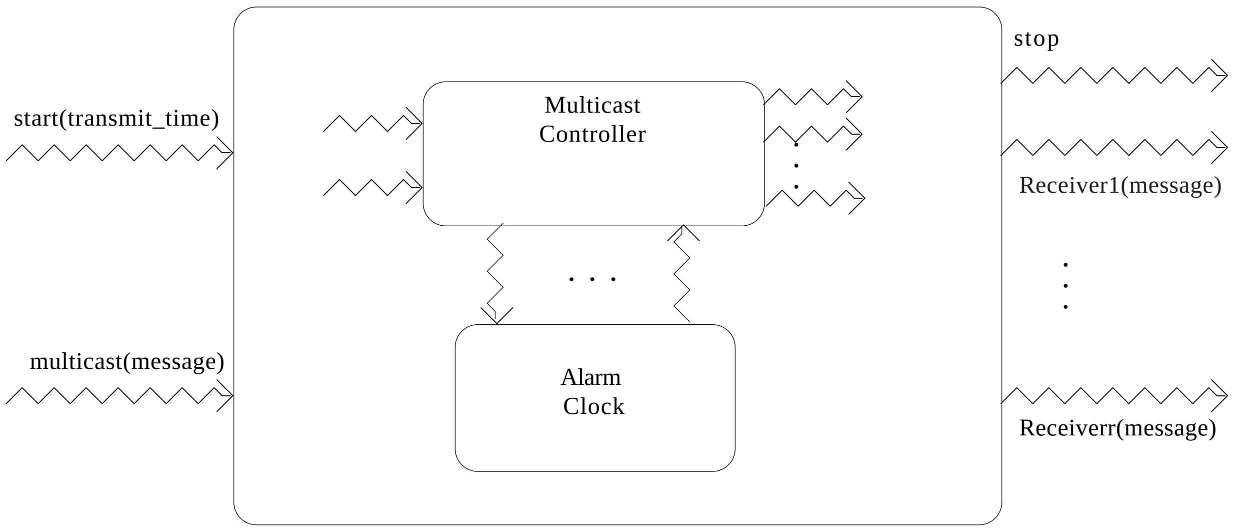


Figure 7.4 Multicast Communications

The system can be implemented with a basic multicast controller and an alarm clock for relative time. The details of the multicast controller CRSM are given in Figure 7.5. [6]

Example:

For an obvious example in real-time systems, consider a power system that broadcasts power_on/power_off signals to its *r* components (power consumers). The multicast message is either "power_on" or "power_off", and components $c_1,...,c_r$ receive the same message on their Receiver$_i$ channels. The *transmit_time* parameter might reflect the physics of turning power on and off, the complexity and nature of the components, and the system requirements.

Our original paper on CRSMs [Shaw 92] indicates how some other common forms of communications may be implemented. This includes asynchronous communications with a non-blocking send and a blocking receive, and synchronous but undirected one-to-one message passing. Utilities for these and others can be constructed when the need arises.

7.2 Interrupts and Faults

In order to specify a broad variety of exceptions and faults, as well as "handlers" or procedures that monitor and recover from them, it is convenient to have a uniform mechanism for interrupting and reentering system components. Our proposed scheme is similar to and inspired by the statechart method for exiting and entering superstates.

We first define interrupts for a CRSM. A CRSM can designate an input channel to be an interrupt channel, using the graphical notation illustrated in Figure 7.6. An interrupt channel, say *int*, is identified as such by adding a dashed arrow; the meaning is that a transition with input command *int*? is connected from *every* state in the CRSM to the *start* state of the machine. Thus, to interrupt a CRSM, one would issue the command *int*!. An example is given in Figure 7.7, where our real-time bounded buffer machine (Section 3) has been augmented with an interrupt.

[6] The message is available between *now* + δ and *now* + *transmit_time* + $k\delta$, where *k* could range from 2 to 2 + 2*r*.

The extra transitions are shown as boldface arrows; normally, these would not be explicitly drawn. Note that the interrupt signal has very much the same effect as the *stop* command.

The same notation is used to denote interrupts of higher-level components (Figure 7.8). To gracefully propagate the interrupt signal to all machines, we employ the multicast facility defined in the last section. Each level of component will broadcast the message to the next level, until basic CRSMs are reached[7] ; in the figure, for example, each R_i is interpreted as an interrupt signal to its component C_i. As an application of higher-level interrupts, consider a power supply subsystem that provides electrical power to the rest of the system; if a power failure occurs, a "power-off" interrupt message can be sent causing all components to revert to their start states.

When a timing or other fault occurs in some component, perhaps asynchronously, it can be propagated through the system by means of this interrupt scheme. The remaining part that needs to be described is the actual fault handler - that component that takes whatever corrective, monitoring, or restorative action is necessary to record and recover from the fault. The detector of the fault which generated the interrupt signal could handle it directly or it could send an appropriate message to some other exception-handling component.

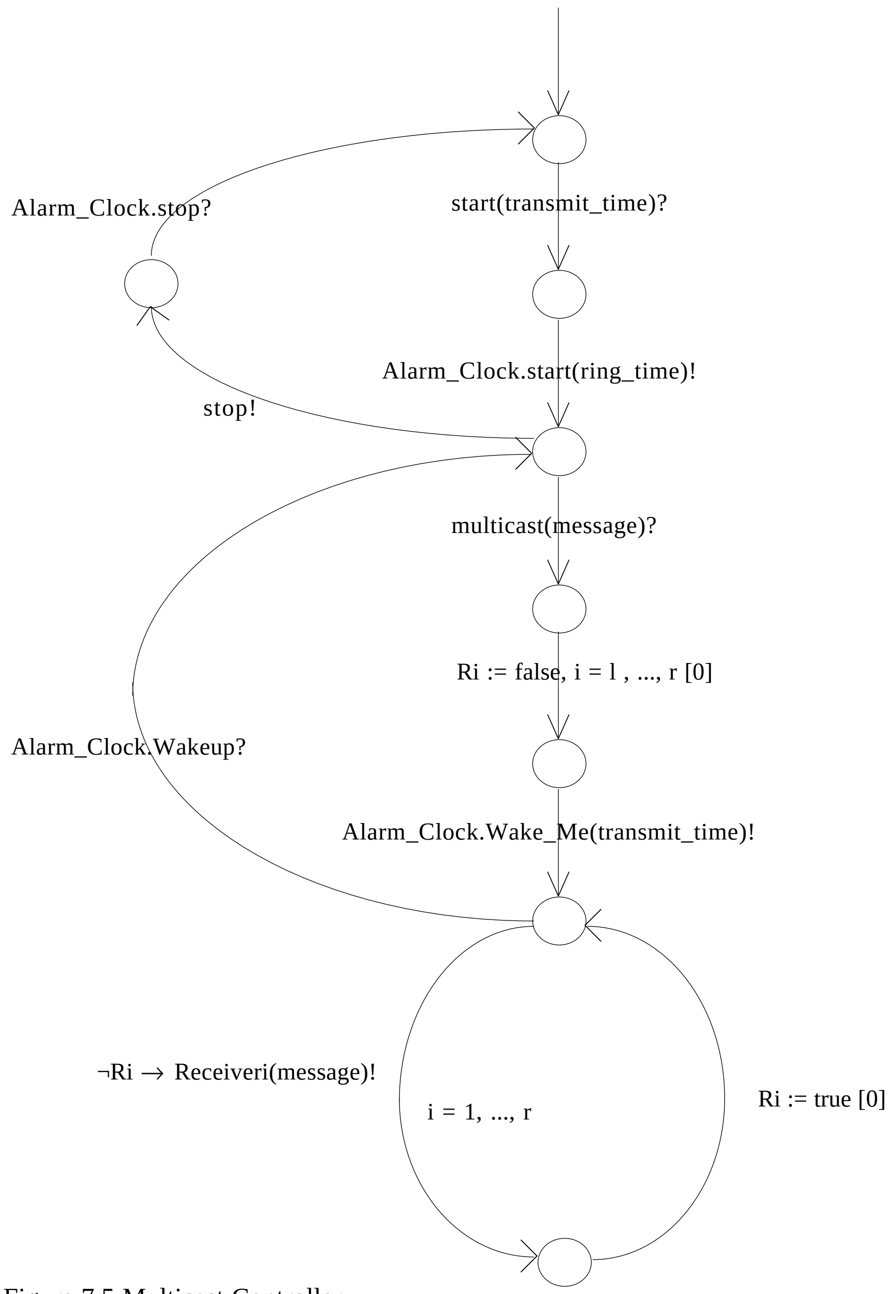
7.3 Control of Periodic and Sporadic Activities

From requirements through implementations, and from theory through practice, the principal means for organizing real-time functions and tasks are as periodic and sporadic activities. We propose a *CC* structure for families of both kinds.

Periodic objects are activated every p units of time and must complete their processing within a deadline $d < p$. The *CC* organization of such an activity is given in Figure 7.9(a). The controller (Figure 7.9(b)) sends an interrupt message *int* to its client if the deadline is exceeded and then generates a timing fault output *fault_stop*. The transition $RT_K?[d]$ provides the deadline time-out. The pair (p, d) are initial input to the controller. Variations and extensions of this basiccontroller and structure might include a fault handler component, and other parameters such as start and stop times for the subsystem and an activation time a for each cycle ($0 < a < p–d$).

Sporadic activities are triggered by an event, say e, and must complete their processing within same deadline d after the event occurrence time. Figure 7.10 contains a *CC* architecture and controller for a standard sporadic object.

[7] It may be convenient to give priority to interrupts and other IO over internal activities, to assure that IO is serviced. Also note that, if necessary, one could define several different kinds of interrupts (different interrupt channels) and interrupts that save state.



start(transmit_time)?
Alarm_Clock.stop?
Alarm_Clock.start(ring_time)!
stop!
multicast(message)?
Ri := false, i = l , ..., r [0]
Alarm_Clock.Wakeup?
Alarm_Clock.Wake_Me(transmit_time)!
¬Ri → Receiveri(message)!
i = 1, ..., r
Ri := true [0]

Figure 7.5 Multicast Controller

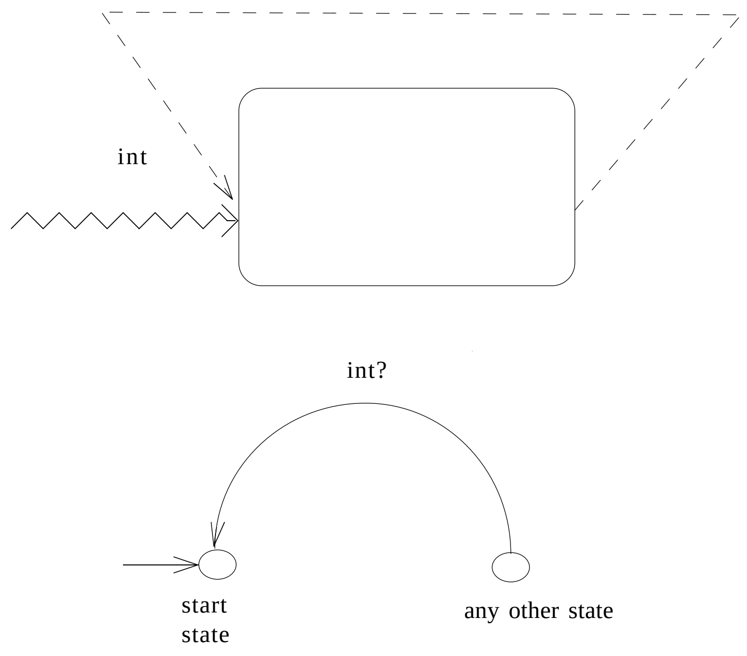


Figure 7.6 Interrupts at the CRSM Level

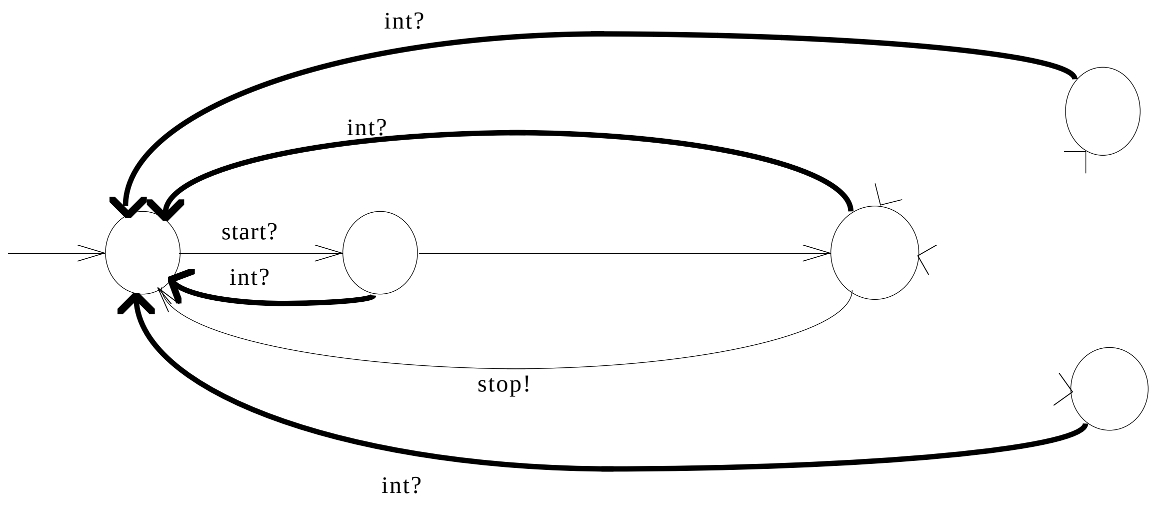


Figure 7.7 Bounded Buffer With Interrupt

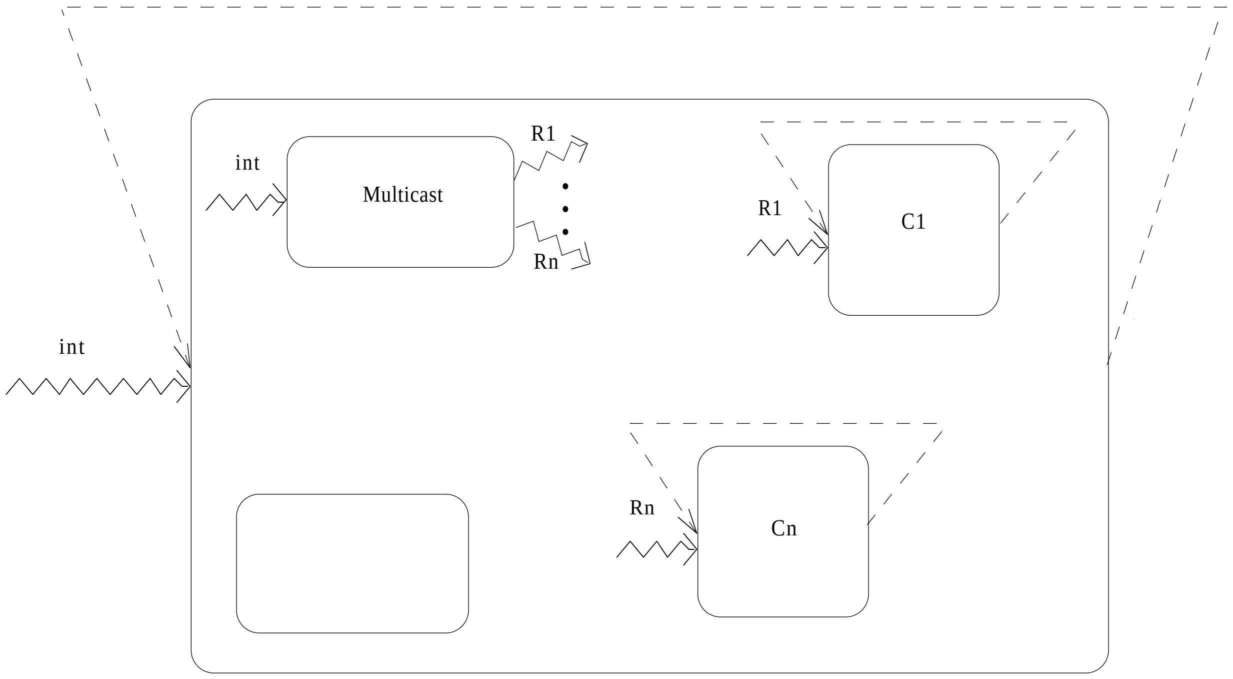
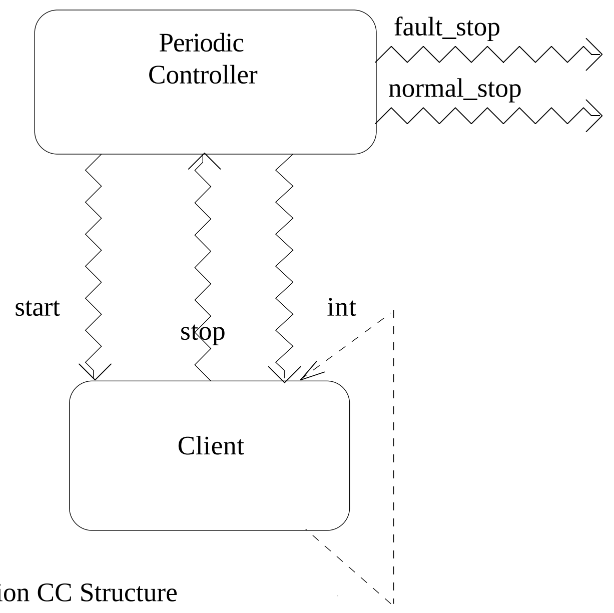
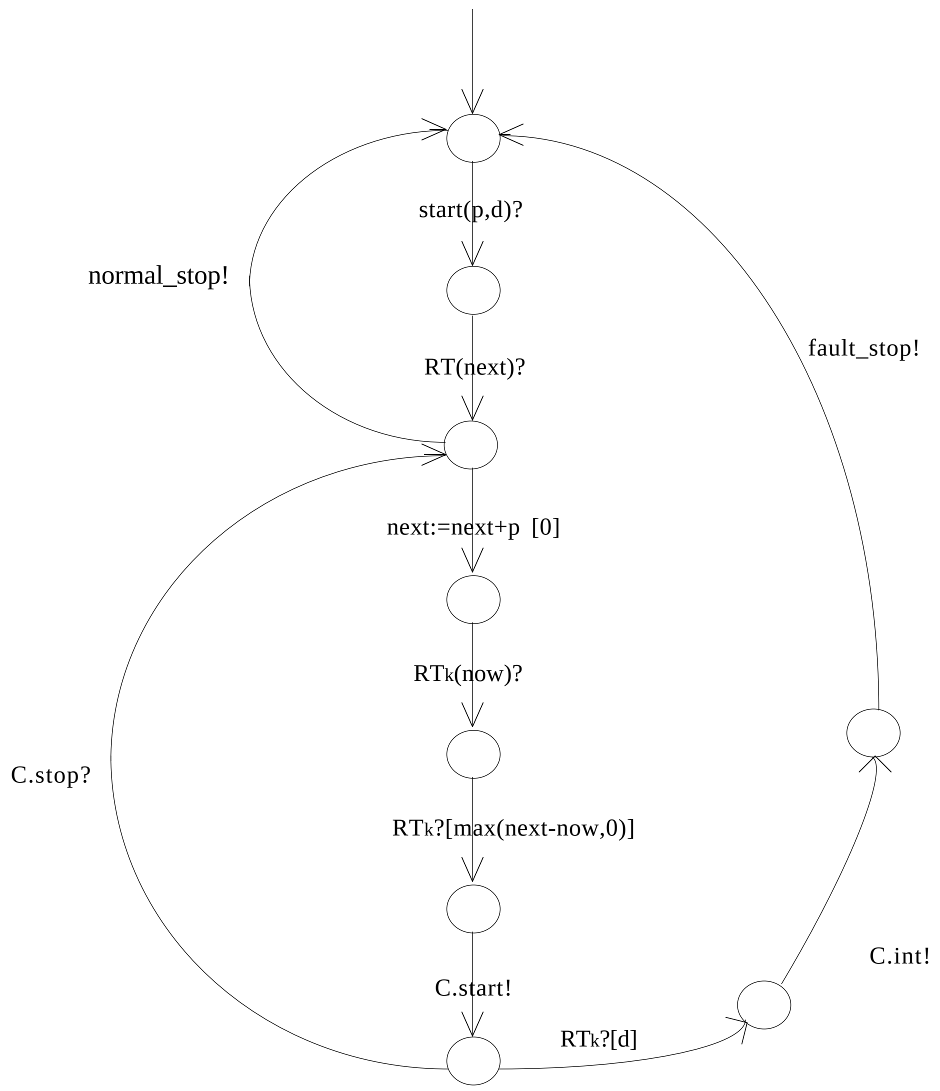


Figure 7.8 Higher Level Interrupts

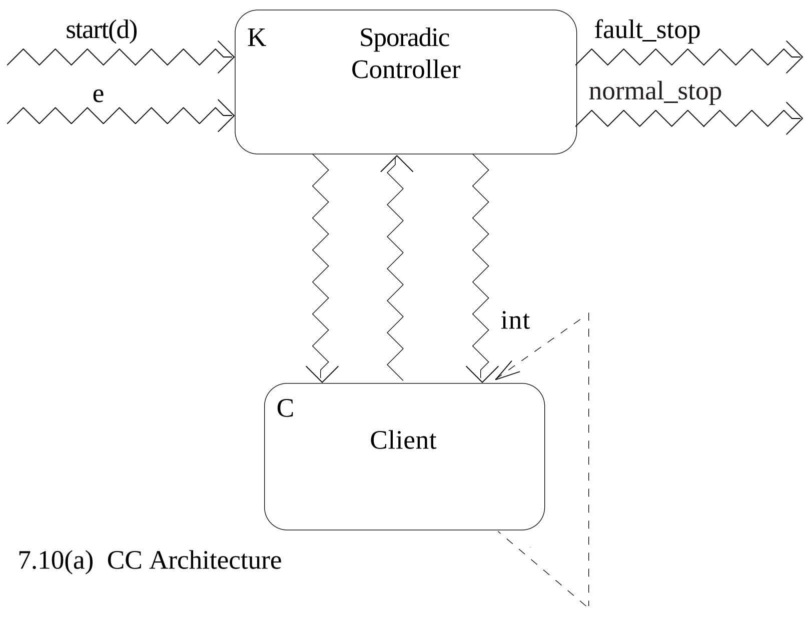


7.9(a) Periodic Function CC Structure

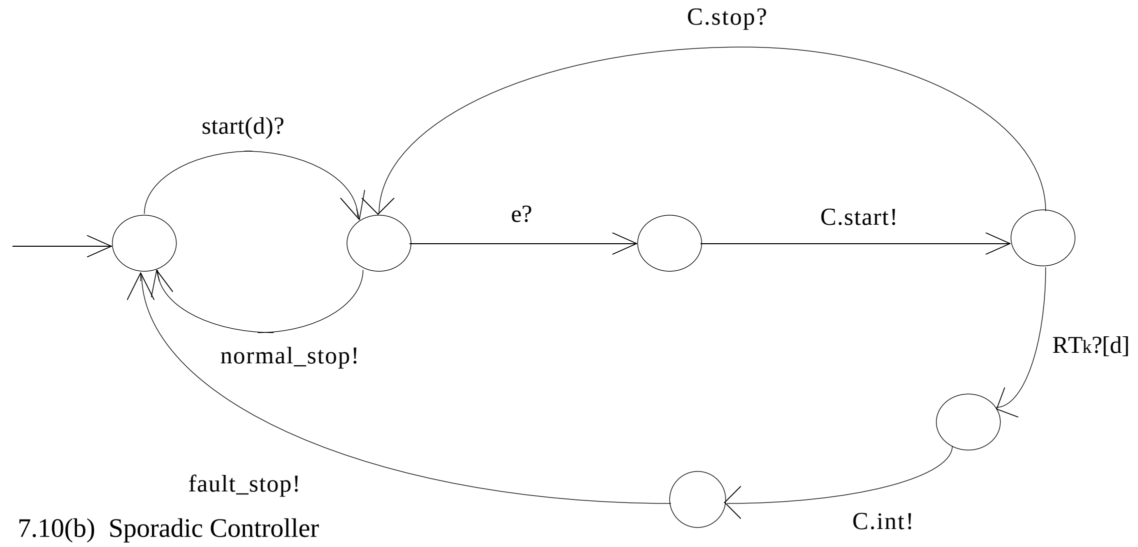


7.9(b) Periodic Controller K With Client C

Figure 7.9 Periodic Object



7.10(a) CC Architecture



7.10(b) Sporadic Controller

Figure 7.10 Sporadic Object

8. Discussion

8.1 Notation and Representations

A consistent graphical representation of components and interfaces has been employed. Component objects are denoted by rounded rectangles labeled by identifiers; geometrically-enclosed rectangles indicate hierarchical grouping of objects (the "part of " relation). Wavy lines with entering (exiting) arrows represent input (output) channels and are labeled by the channel name and the types of their parameters. Interrupts are identified by dashed arrows. At the basic CRSM level, a standard state diagram is used. It is tempting to define more specific icons for each of the mechanisms and paradigms presented, for example, so that sequential control can be obviously distinguished from parallel control, and periodic objects immediately appear differently then sporadic ones. However, the benefits are not yet evident and we will resist for a while.

Similarly, and perhaps more clearly, there are any number of reasonable possibilities for purely textual notations of in-the-large objects. (A formal (textual) notation for CRSMs is given in [Shaw 93]). For example, a subsystem M could be defined:

$M = \{ M_1, \dots , M_m \}$, $m>1$
/* M consists of m components $M_1, \dots , M_m$. */
$Ch_{\text{connected}} = \{CC_1, \dots, CC_c\}$, $c \geq 0$
/* Each channel CC_i is connected between two components of M. */
$CC_i = (name, <M_{is} , M_{ir}>, message\_type)$
/* This defines each CC_i . M_{is} and M_{ir} are in M. */
$Ch_{\text{unconnected}} = \{UC_1, \dots , UC_u \}$, $u \geq 0$
/* These are the interface channels of M. */
$UC_i = (name, M_i, in|out, message\_type)$
/* Each UC_i is defined. */

Standard control, encapsulation, and time-constrained objects may also have specific textual representations. Examples are:

sequential:	$M_1 ; M_2$
parallel :	$M_1 \parallel M_2$
guarded selection :	$(g_1(x) \rightarrow M_1)$ [] $(g_2(x) \rightarrow M_2)$
periodic activity:	*periodic* $M(in\colon p,d;\ out\colon normal \mid fault)$

As in the graphical case, it is not yet clear whether such notations are useful.

Several naming issues also arise. It should be possible to connect any two type- and direction-compatible channels. This requires that the name of the channels be identical after the connection is declared, which in turn would generally require a systematic renaming of sender and/or receiver IO calls in the base CRSMs. Arrays of channels are necessary for "server" and other subsystems; examples are broadcast servers, ADTs, and n-way fork/join controllers. Class and instance declarations also need to be provided and distinguished. All of this seems no different than analogous facilities available in modern programming languages and can be directly borrowed.

8.2 Time and Other Resources : Specification and Analysis

Timing behavior can be specified using the basic CRSM facilities - through time bounds on state transitions and through communication with clock machines. Higher-level timing specifications must either use these mechanisms directly or handle each example on a one-of-a-kind basis; for example, we do this in the alarm clock machine and in the parameterization of other machines (ring_time, broadcast_time, period, deadline). A research problem that we have not addressed here is whether or not there should be some general higher-level scheme for describing timing constraints in-the-large. Ideally, it would be attractive if the CRSM techniques could be adapted, i.e. scaled-up, to the subsystem level; the *CC* architecture provides a convention for partially doing this by identifying the clock machine of the controller as the clock machine of the component. More experience will help decide whether any extensions are necessary or convenient.

One may also wish to use an in-the-large scheme to specify the time-constrained sharing of other resources that are required by components. For example, processor sharing among a set of independent components might be handled through an "executive" controller that schedules clients, say in round robin fashion. In order to do this conveniently, some means of saving state on an interrupt would appear to be necessary; this could be implemented by changing our interrupt mechanism appropriately. Sharing of communications lines and input-output devices can be described by defining components that simulate the actual sharing.

At this point, we have given little thought to techniques for analyzing and verifying the behavior of large specifications. The standard proposed solution is to prove various properties of such systems from the bottom-up, starting with individual components. Typically, the standard approach does not scale up. A top-down methodology that examines and verifies traces a level at a time, starting with of the basic systems-level requirements, would seem promising.

8.3 Tools and experiments

Paper experiments and proposals for large systems design can be deceiving and unrealistic. Proof of scalability is accomplished only by demonstrations of scalability. Software tools are needed to construct specifications, to store libraries of specifications, to test specifications for consistency, to analyze them, to simulate their execution, to monitor their behavior during execution, and to verify their properties when possible. Of these tasks, the most useful initially would be programs for building and executing specifications - an in-the-large version of the tools described in [Raju & Shaw 92; Raju 93].

With these tools, especially the latter, we will be in a position to do some convincing experiments on interesting real-time systems. This should also lead to a larger number of useful paradigms for real-time requirements and designs.

9. Summary

The research and contributions have two principle parts. First, we demonstrated how in-the-large components for specifying real-time systems may be constructed using a particular controller-client software architecture. Second, with the *CC* architecture, we defined particular types of components and control needed for real-time requirements and designs. Both parts are based on our CRSM notation. As discussed in the last section, much remains to be done.

Acknowledgments:
Thanks to H. Rebecca Callison, Travis Craig, and Sitaram Raju for critiquing the paper.

References

[Gabrielian & Franklin 91] A. Gabrielian and M. Franklin, "Multilevel Specification of Real-Time Systems," *Comm. of ACM 34*, 5 (May 1991), pp. 50-60.

[Harel 87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming 8*, (1987), pp. 231-274.

[Hoare 85] C. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[Jahanian & Mok 89] F. Jahanian and A. Mok, "Modechart: A Specification Language For Real Time Systems," IBM Tech Report RC 15140, Nov. 1989.

[Kramer et al. 93] B. Kramer, Luqi, and V. Berzins, "Compositional Semantics of a Real-Time Prototyping Language," *IEEE Trans. on Software Eng.*, Vol. 19, No. 5, May 1993, pp. 453-477.

[Leveson et al. 91] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese, "Requirements Specification For Process-Control Systems," TR 92-106, Computer Science Dept., UC Irvine, 1992.

[Raju 93] "An automatic verification technique for communicating real-time state machines," *TR #93-04-08*, Dept. of Computer Science & Engineering, University of Washington, Seattle, April, 1993.

[Raju & Shaw 92] S. Raju and A. Shaw, "A prototyping environment for specifying, executing and checking communicating real-time state machines," *TR #92-10-03*, Dept. of Computer Science & Engineering, University of Washington, Seattle, October, 1992. A revised version is in publication in the journal *Software-Practice & Experience*.

[Shaw 92] A. Shaw, "Communicating real-time system machines," *IEEE Trans. on Software Eng.*, Vol. 18, No. 9, Sept. 1992, pp. 805-816.

[Shaw 93] "A (more) formal definition of communicating real-time state machines," *TR #93-08-01*, Dept. of Computer Science & Engineering, University of Washington, Seattle, August, 1993.