UML Statecharts

Bruce Powel Douglass
Chief Scientist
I-Logix
bpd@ilogix.com

Introduction

Objects have both structure and behavior. Object behavior may be viewed solely within the context of individual objects or behavior may be viewed in the larger context of object collaborations. Objects work together in these collaborations to implement (*realize* in UML-speak) higher-order behaviors defined by use cases. I find it useful to partition behavior into three types: simple, state, or continuous. These different types of behavior are distinguished on the basis of how they treat the object's history over time. Simple behavior does not depend on the history of the object whatsoever. State behavior divides up the behavioral space of the object into nonoverlapping chunks called *states*. Continuous behavior depends on the object's time history, but does so in ways that cannot be easily divided up into disjoint states.

Both state and continuous behavior constrain the overall behavior of their context. For example, an object might have five different states, each of which accepts a different set of events, performs different actions, and can reach a different set of subsequent states. Such an object might have 15 different operations defined, suitably constrained by the object's state machine limiting both the sequence of those operations and the conditions under which they may be executed. This can vastly simplify the overall dynamic behavior of the object by providing a set of rules which govern how and when those operations can be invoked. Similarly, a continuous object, such as might implement a feedback control system, operates by applying differential equations (in their discrete formulation as difference equations). The equations, which map somehow to operations defined on the class, execute in a particular sequence in response to incoming data and previous results. This again results in simpler behavior that equally-rich but unconstrained application of those operations.

State machines are the primary means within the Unified Modeling Language (UML) for capturing complex dynamic behavior. The UML is a third-generation state-of-the-art object modeling language that defines a comprehensive set of notations, and, more importantly, defines the semantics ("meaning") of those language elements. The UML is an inherently discrete language, meaning that it emphasizes discrete representations of dynamic behavior, such as state machines, over continuous representations. Although many object systems do, in fact, perform continuous control functions and do it well, the UML provides special support in the area of finite state machines. In this article, I would like to explore the underlying semantics and notation of state behavior as defined by the UML.

I'd like to start off with two interesting questions: (1) what exactly is a finite state machine, and (2) to what kinds of structural pieces of the object model do they apply? It turns out that there are many answers to the first question, using different vocabularies and addressing different concerns. For example, a relatively formal definition is

- A finite state machine (M) is an abstract model consisting of
  - a finite state set (S),
  - a starting state (s_0)
  - an input alphabet (Σ)
  - a mapping function $\delta = \Sigma \times S \rightarrow S$
  - an input sequence acceptance function

    $\beta = S \rightarrow \{0, 1\}$

    such that

    $M = (S, \Sigma, s_0, \delta, \beta)$

A graph-theoretic definition would define a state machine in terms of nodes and edges. However, I'm more interested in a definition that emphasizes the practical applications of state machines (while keeping in mind that somewhere in the closet, there's a bunch of mathematics if and when I need it). Therefore I'll use the following definition:

> *A finite state machine is an abstract machine that defines a finite set of conditions of existence (called "states"), a set of behaviors or actions performed in each of those states, and a set of events which cause changes in states according to a finite and well-defined rule set.*

This definition makes very clear the notion of a finite state machine as something that constrains object behavior. When this "abstract machine" is in one state, it performs only a certain subset of actions, it accepts only a certain set of incoming events, and can change state directly to only a subset of all possible states. A traffic light system is a common example of a state machine. It might have states such as "Green", "Yellow", "Red", "Flashing Yellow" and "Flashing Red" (some countries even have a "Flashing Green" state to keep you on your toes). These states cannot be entered except in particular, well-defined sequences. This definition of a state machine maps well to Bertrand Meyer's idea on object contracts, which consist of preconditional invariants (predecessor states and triggering events), event signatures, and postconditional invariants (actions performed).

The second of the two questions what sort of things exhibit state behavior? In functionally-decomposed systems it is some usually vaguely defined set of operations that act on some set of shared data variables. In object-oriented systems, the boundary is rather more crisp – objects are things that exhibit state behavior. In the UML metamodel (model of the UML itself), Classifiers are metaclasses that can associated with state machines. Classifiers include such things as classes, interfaces, datatypes, use cases, components, and subsystems. Instances of these things (objects are instances of classes) then have their own individual machine executing that defined state behavior. The most common things in the UML to model with explicit state machines are classes and use cases.

A *use case* is a named piece of functionality visible in a context that returns a result visible to one or more objects in that context (called *actors*). They define a piece of magic that some aspect of the context performs without implying any specific design or implementation. This magic can be expressed with multiple scenarios of interaction between the thing in that context and the associated actors.

Note: If you're building functionally-designed systems today rather than object oriented systems, it does not imply that state machines are inappropriate or cannot be used. Rather than apply the state machines to specific objects, the state machines will apply to a set of independently-defined functions and the data that they share. State machines are a very powerful formalism that transcends the notions of functional and object-oriented design. Nevertheless, state machines map very clearly and obviously to objects, which accounts for much of their popularity in object-oriented designs.

Basic Semantics and Notation of State Machines

The UML provides two different kinds of state machine formalisms: statecharts and activity diagrams. They differ in the kinds of situations to which they are applied. Statecharts are used when the transition from state to state takes place primarily when an event of interest occurs. Activity diagrams are appropriate when the object (or operation) changes state primarily upon completion of the activities executed within the state rather than the asynchronous occurrence of events. We will limit our discussion here to the much more commonly used form – statecharts. The use of activity diagrams is covered in several of the references.

Statecharts consist of three primary things – states, transitions, and actions. *States* are distinguishable conditions of existence that persist for a significant period of time. *Transitions* are the means by which objects change states in respond to events. State machines also execute *actions* – atomic behaviors – at various points in a state machine, such as when an event triggers a transition, when a state is entered or when a state is exited. Actions may be simple statements, such as "a++" or they can invoke operations defined within the context object or other objects.

An Example: Reliable Communication Service

Let's consider a simple example: the communication between sender and receiver objects, mediated by a communications controller. The sender can specify the quality of service (QoS) on a per-message message basis. The system supports three qualities of services:

- At most once — This QoS can be thought of as "send and forget." If the message gets lost or garbled, then the receiver won't get it, but that's ok. The receiver receives the message no more than one time.
- At least once — This QoS sends a message and waits for an explicit ACK message from the receiver. If the sender side doesn't receive an ACK within a reasonable time, then the sender side resents the message. This

	can repeat until some maximum retry count is reached. This QoS is called At Least Once because if the returning ACK is the message lost or garbled, rather than the informational message, then the receiver may get that message more than once.
• Exactly once	At Least Once QoS works fine for informational messages that send absolute values (like “Speed = 60”), but it fails for incremental messages (such as “Increment Count”). For this reason, the communications systems supports a QoS that ensures that if multiple copies of a message are received, the duplicates will be quietly discarded.

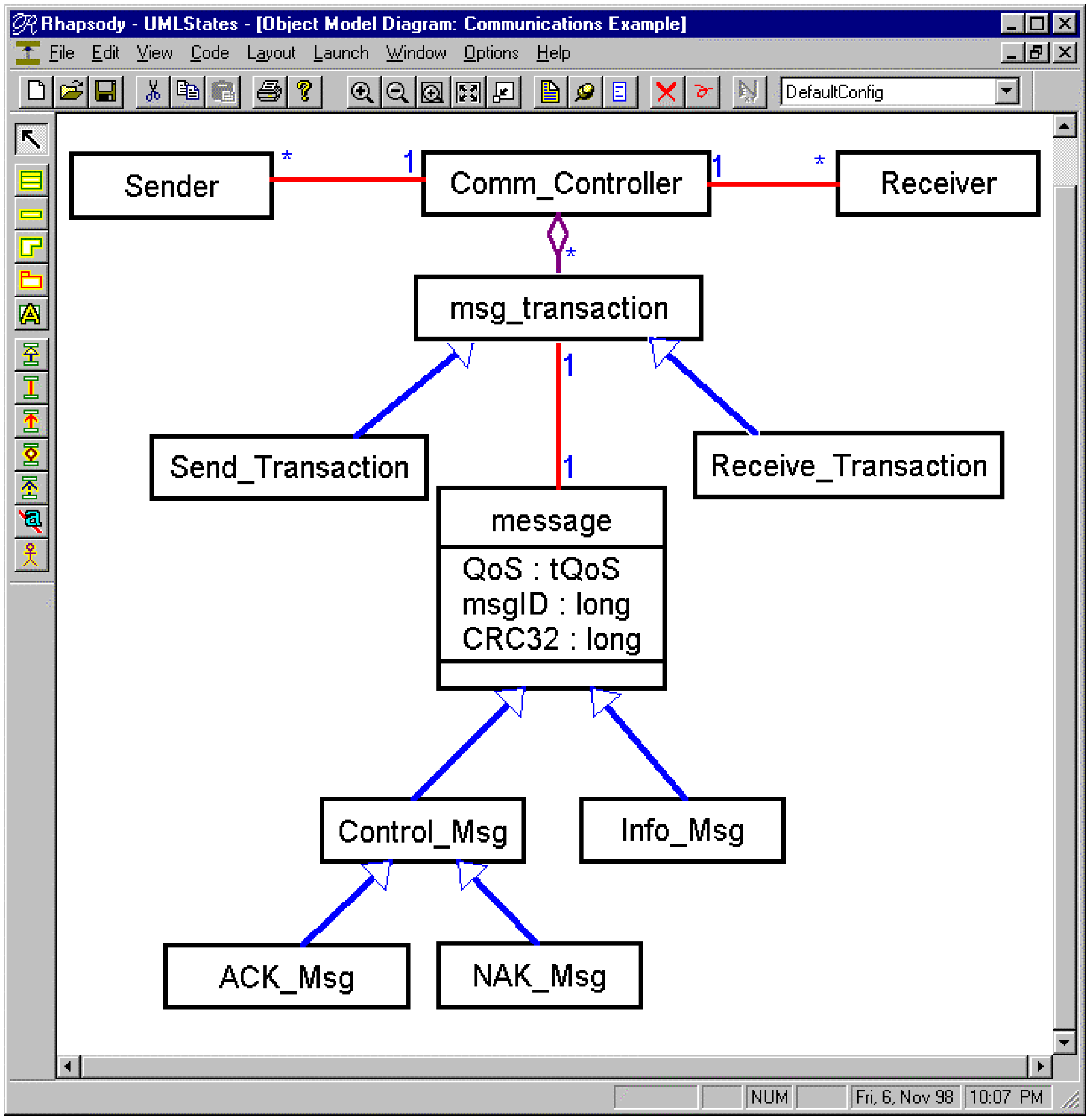


Figure 1: Communication Systems Object Model

An object model that addresses this problem is shown in Figure 1. The Comm_Controller object gets a Send(msg) message from the sender and, if appropriate, creates a message Send_Transaction object that manages sending the message, waiting for the returning ACK and resending the message if necessary. This alone handles the At Least Once QoS requirement. The state machine for this is shown in Figure 2.

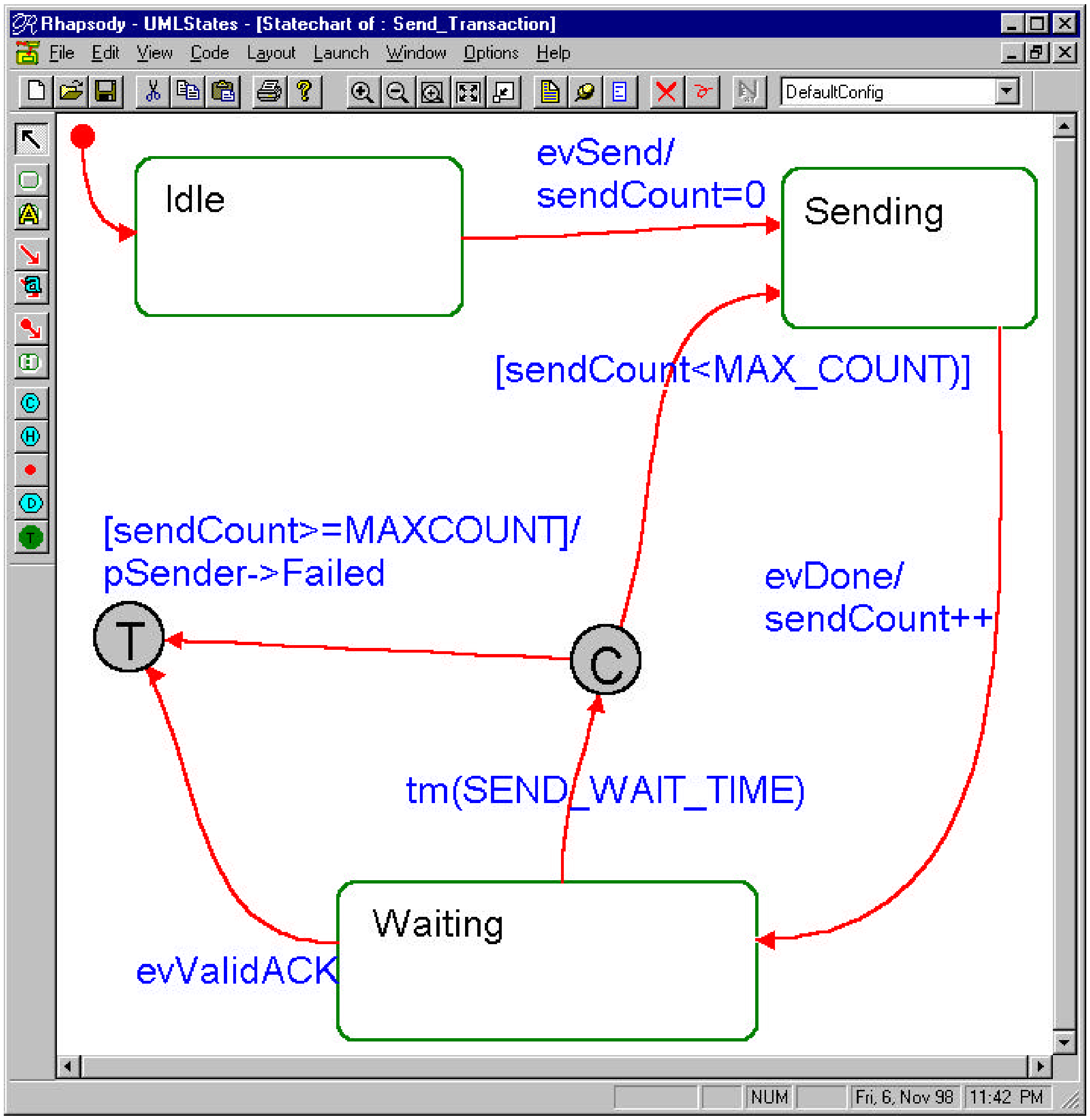


Figure 2: Statechart for Send_Transaction

The statechart for the Send_Transaction is fairly straightforward. States are represented by the rounded rectangles while transitions are depicted as directed lines. The object starts life in the *Idle* state, as indicated with the initial pseudostate (the transition with a ball at one end). Then the object receives the *evSend* event, it performs an action to set one of its attributes called sendCount to zero. The object then resides in the *Sending* state until the message is done being sent. Then the object transitions to the *Waiting* state.

In the Waiting state, two different things of interest can happen. First, the Send_Transaction could receive an *evValidACK* in response to receiving a valid ACK from the sender. In that case, the object transitions to the *terminal pseudostate* (shown with the circumscribed T). The terminal pseudostate indicates that the object stops receiving events and is destroyed.

The other event of interest in the *Waiting* state the elapse of a timeout. This indicates that the ACK has not been received within the appropriate time window. But there are two possibilities in this case. If we have exceeded the number of retries we're willing to make, then we should notify the sender and proceed to the terminal pseudostate. On the other hand, if we have not exceeded the maximum retry count, then we ought to go to the *Sending* state and try again. This selection is done by transitioning on the timeout event (shown by the special tm( ) event) to a *conditional pseuodostate*. The two different possibilities exiting the pseudostate are distinguished by having different guards, or selection criteria.

The Receive_Transaction works somewhat differently. It is created when the receiving Comm_Controller receives a message that has a QoS of Exactly Once. Each new message received with this QoS results in the creation of a Receive_Transaction object. The Receive_Transaction stores the message ID of that message and waits. When another message comes, the Comm_Controller asks each Receive_Transaction object if it is waiting for a message with that particular ID. If yes, the object responds to the message with a transition-to-self (i.e. transition in which the source and target states are the same). This has the semantic effect of resetting any timeout transitions exiting that state. The other event of interest is the elapsing of a period of time since the last time that message was received. When this latter event occurs, the Receiver_Transaction object is destroyed and the receiver side forgets that it ever received a message with that particular message ID.

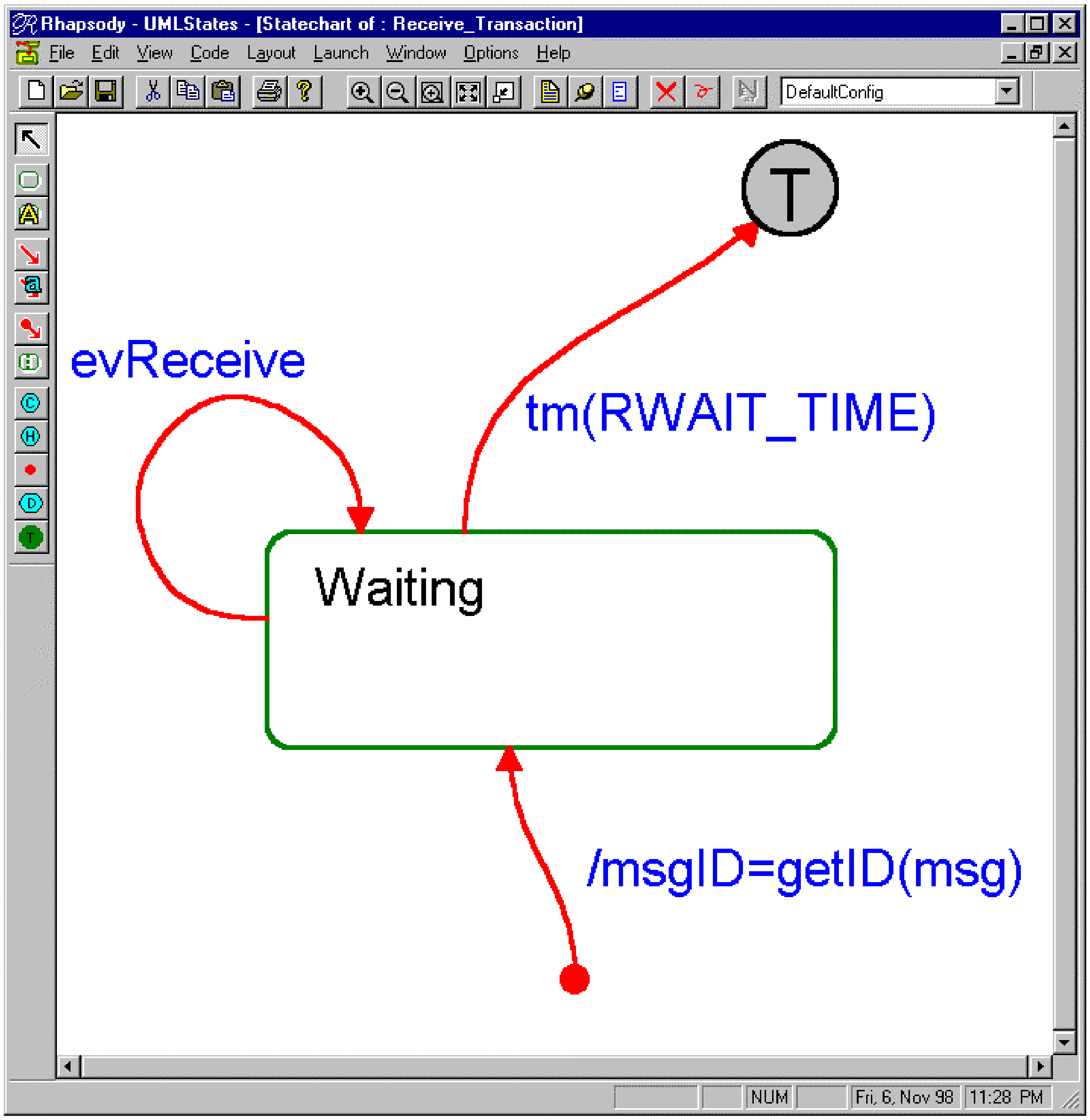

Rhapsody - UMLStates - [Statechart of : Receive_Transaction]
File Edit View Code Layout Launch Window Options Help
DefaultConfig
evReceive
tm(RWAIT_TIME)
T
Waiting
/msgID=getID(msg)
NUM
Fri, 6, Nov 98
11:28 PM

Figure 3: Statechart for Receive_Transaction

The Basic Notation

States

Figure 4 shows all the basic elements of statecharts: states, transitions, actions, and a variety of different pseudostates. In the figure, the high-level states are S0, S1, and S2. These states are called *or-states* because within their context, the object must be in one and only one of these states at any given time. The initial state is indicated by the *initial* *state pseudostate* on the left and its associated transition to state S0. This indicates that when the object is created, it enters state S0 initially before processing any events.

Some, but not all of these states have substates nested within them. S0, for example, has nested states S0_1 and S0_2. This is another way of saying "when the object is in state S0, it must be in either of two more specific conditions." For example, when a device might have high-level states of {*off, on*} and when it is in the on state it can be in any other following substates {*booting, operating, shutting down*}. Substates can be nested arbitrarily deep. For example the booting state might be decomposed into sub-substates of {*BIOS test, RAM test, ROM test, identifying peripherals, configuring system*}. In Figure 4, state S2 has three substates, one of which is broken down into substates of its own.

If an object's behavior is defined by a state machine (such an object is said to be *reactive*), then it spends all of its time in one state or another. It must always be within exactly one or-state within the level of context. Thus, the machine shown in Figure 4 must be in either state S0, S1 or S2. If it is in state S2, then it must be in one of its substates S2_1, S2_2, or S2_3. If it is in state S2_3, then it must also be in either state S23_1 or S23_2. States can be referenced by a unique name, such as S23_1 or by referencing its owner state, as in S2.S2_3.S23_1.

Transitions

As previously mentioned, transitions indicate that the state machine responds to an event while in certain states. For example, if you look in state S0, you'll see that while the state machine is in state S0_1 it accepts an event e0 and when that occurs, the object transitions to state S0_2. Similarly, while in state S0_2, if an e2 event occurs, the state machine transitions to state S0_1.

Transitions affecting a superstate apply at all levels of nesting within that superstate. While in state S0, if an e1 event occurs, the system transitions to state S1. Since the object must be in either state S0_1 or S0_2 while in state S0, this means that this event transition applies to both substates of S0. This is one of the benefits of nesting states – you can show multiple state transitions to a common target state with a single transition by simply nesting those states to which the transition applies. In general, a transition exiting a superstate applies to all substates within it, regardless of how deeply they are nested. The transitions triggered by event e3, for example, applies to all substates of state S2, including the deeply nested substates S23_1 and S23_2.

Transitions are modeled as taking approximately zero time to execute[1], as implied by the statement that an object spends all of its time in states. If a transition can take a significant amount of time, then the object should be decomposed into more states so that

[1] Classical state machines assume zero-time transitions, but this constraint is relaxed in the UML statechart semantics definition. Nevertheless, transitions need to be "very short" and the object dwells in states virtually all of its life.

eventually, the time taken to get from a predecessor state to a subsequent state is insignificant. State machine execution itself is said to proceed in discrete time units called *run-to-completion (RTC) model steps*. An RTC step is the period of time in which events are accepted and acted upon. Processing an event always completes within a single model step, including exiting the source state, executing any associated actions, and entering the target state.

Transitions represent the response of a state machine to events. Any event that is not explicitly listed as causing an event to occur in a given state is quietly discarded should it occur. For example, in state S0_1, only three events result in a state transition: e0, e1, and e5. Event e0 causes a transition to state S0_2. Event e1 causes the object to leave S0_1 and its enclosing superstate S0 and transition to state S1. Event e5 works the same way, but it ends on a conditional pseudostate which selects the target state on the basis of guards.

A guard is a Boolean condition that returns a TRUE or FALSE value that controls whether or not a transition is taken following the receipt of a triggering event. A transition with a guard is only take if the triggering event occurs *and* the guard evaluates to TRUE. For example, look at the transition from the state S2 to the terminal pseudostate in Figure 4. It is called an anonymous or null transition because it has no triggering event. It becomes enabled as soon as the object enters state S2. However, it has a guard – the action isDone(). If this function returns TRUE, then the object transitions to the terminal pseudostate (and ceases to respond to any further events). Because there is no triggering event in this case, if the guard evaluates to FALSE, then the transition is not re-evaluated until and unless the source state is exited and reentered. If there were a triggering event, then whenever that event reoccurred while the source state was active, the guard would be reevaluated. As long as the guard evaluated to FALSE, the triggering event would be discarded and the transition would not be taken.

Another place in the figure that you see guards in on transition segments exiting the conditional pseudostate (shown as a circumscribed “C” in the middle of the figure). Conditional pseudostates are a notational shorthand for multiple exiting transitions all triggered by the same event but each having different guards. In this case, the triggering event is e5. The resulting state differs depending on the evaluation of the guards. In this case if a<0, then the resulting state is S1. If a>0, the object transitions to the terminal pseudostate. The “else” guard handles all other conditions and transitions to the S2 state. In this case the inclusion of an else guard means that whenever event e5 occurs, the transition will be taken. If you removed the else clause and neither remaining guard returned TRUE, the triggering event would be discarded.

So far, we’ve discussed two kinds of triggers. The first is a named trigger – that means that some named event results in a transition being taken. The other is the null transition, meaning that the transition is evaluated only once upon entrance to the source state. If it has no guard, or if the guard evaluates to TRUE, then the transition is taken immediately. Another kind of trigger is shown in state S2, the tm(x) event. This indicates a timeout event which fires some specified period of time after the state is entered. The UML does not define the units, but commonly they are milliseconds or microseconds. The timeout event is cancelled if the source state is exited prior to the timeout. Also, if another event triggers a transition-to-self (a transition in which the source and target states are the same), any timeout transitions leaving that state are reinitialized. This is the case for event e7 applied to state S23_3. The timeout interval represented by the tm(660) event transition (shown at the bottom of the state) is restarted from scratch whenever the object is in state S2_3 and an e7 event occurs.

The UML defines four different kinds of events:

• Signal Event	An event due to some external asynchronous process
• Call Event	An event due to the execution of an operation within the object
• Change Event	An event due to the change in value of an attribute
• Time Event	An event due to either the lapse of an interval of time

By far, Signal Events are the most common in practice, but time events are also frequently used. A Signal Event is associated with a Signal, which is a type of Classifier in the UML metamodel, similar to a Class or a Use Case. Events can have zero or more parameters. This allows the event to convey not only the occurrence of some interesting incident in space and time, but also qualitative and quantitative information regarding that occurrence, such as the number of knob clicks, the airspeed determined by polling the sensor, or the level of hazardous radiation due to the occurrence of a leak in the reactor core.

In the UML, events are generalizable. This means that they can be members of a generalization taxonomy. The practical implication of this is that a transition triggered by any event e will also be triggered by any subclass of that event. This is sometimes called *polymorphic event triggering*.

The general transition for a transition is

event-name '(' parameter-list')' '[' guard ']' '/' action-list '^' event-list

All of these fields are optional, as mentioned previously. If a transition is totally unnamed, then it activates as soon as its source state becomes active and completes its activities (if any). This is called a null trigger or anonymous event, as mentioned previously.

The event-name is the name of the event triggering the transition. The same event may trigger multiple transitions in a state machine. They can be unguarded if they apply to different source states, or they may be guarded (in which case they may apply to the same source state). For example, in Figure 4, the e1 event triggers a transition between states S0 →S1, but the same event also triggers a transition between states S23_1 → S23_2. Should this event occur while the object is in any other state, it would be discarded.

Similarly, the event e5 triggers three transitions from the S0 source state. The target state is selected by which guard evaluates to TRUE. If no guard evaluates to TRUE, then the event is discarded. However, if more than one guard evaluates to true, then the model is ill-formed. In such a case, the statechart semantics stipulate that only one of the transitions will be taken, but you can't predict which one it will be.

The parameter list, when present, is enclosed within parentheses, just as you would function parameters. They may include types, if desired. Guards are enclosed within square brackets. A guard must return a Boolean, but it can be an arbitrarily complex expression, including logical operators such as AND and OR. Guards should not have side effects because a guard may be evaluated many times prior without actually taking the transition. So using a guard such as

```
        [++a == TIME_TO_MOVE]
```

is a bad idea.

Action lists consist of zero or more actions, which can be simple program statements, such as "a++" or calls to operation defined within the object or messages sent to other objects. Action-lists are also preceded by a slash ('/'). The transition label can also include an event-list. Event-lists are always preceded with a carat ('^'), and have a list of events that are generated as a result of taking the transition. Events generated in this way are called *propagated events*. Both the execution of a transition's action-list and the generation of the propagated events in the event-list are only done if and when the transition is actually taken. If the event occurs but the transition is not taken (due, for example, to a guard evaluating to FALSE), then the actions are not executed and the propagated events are not generated. A propagated event is shown in the figure attached to the timeout transition-to-self for state S2_3.

It should be noted that there are those within the OMG UML RTF and RTAD working groups that think that propagation of events is really just a special form of an action, and it would be better to use an action within the action-list, such as GEN(FlameOn) rather than have a separate event-list clause. Time will tell who wins out, but in the meantime, a carat is used to indicate an event-list in the UML standard (individual tools vary in the compliance with this notation).

Actions and Activities

Actions are small atomic behaviors executed at specified points in a state machine. Actions are assumed to take an insignificant amount of time to execute and are non-interruptible. Actions are separated from the event-name and guard with a slash ('/'). The action-list clause may contain zero or more actions. The exact syntax of the action expressions isn't defined by the UML, so many people use either a structured English or expressions in an implementation language such as C++ or Java. In statecharts, actions may be placed in any of three places: on a transition, on a state entry, or on a state exit.

Figure 4 shows many examples of actions in all three positions. Notice, for example, the text "/setupAction()" next to the outermost initial pseudostate →S0. This in a null transition (you should never specify a triggering event for an initial transition) but it does specify an action. This means that prior to entering the state S0 when the object is first created, execute the setupAction() operation. The transition triggered by e1 between S0 →S1 has two actions named – action1() and action2(). The actions are only executed if the transition is taken (e5 occurs and isOk() evaluates to TRUE). Note that the actions can be associated with different transition segments, as with the transition triggered by event e5. A common action (action1()) is executed followed by the action executed by the conditional segment branch taken.

Actions associated with states may be shown on the statechart, or may be hidden inside a pop-up "features" dialog. When shown on the state, the keyword "entry" indicates the entry action list and "exit" indicates the exit action list. These are the actions taken when the state is entered and exited respectively. As a rule of thumb, actions that are always executed when a state is entered should be specified as entry actions and actions that are always execute when a state is exited should be exit actions. Actions can be put on transitions when they are executed only when entering or exiting a state via specific transition paths.

Actions are always executed in a predefined order:

1. exit actions of the source state(s)
2. transition actions
3. entry actions of the target state(s)

For example, if event e0 occurs while in the source state S0_1, then the order of execution of actions is

1. S0_1 exit actions: ex2()
2. transition actions: (none)
3. S0_2 entry actions: ent1()

If then an e2 event occurs, the following actions are executed:

1. S0_2 exit actions: (none)
2. transition actions: ++a; print(a);
3. S0_1 entry actions: ent(2)

The situation is complicated somewhat when a transition crosses state nesting boundaries. The basic rules are:

- When exiting a superstate, execute the exit actions in the order from the most deeply nested state first to the least-deeply nested state last
- When entering a superstate, execute the entry actions in the order from the least-deeply nested first to the most deeply nested state last

Consider event e4 from state S2_1 to state S0_2. The order of actions executed is

1. exit actions
    - S2_1 exit actions: ex1()
    - S2 exit actions: closeShutters()
2. transition actions: t1(), t2()

3. entry actions
    - S0 entry actions: openValve()
    - S0_2 entry actions: ent1()

As mentioned above, an action is a small, non-interruptible behavior. Activities are different[2]. Activities are behaviors performed as long as an object resides in some state. Activities represent behaviors that can take a significant time to execute and are interruptible. Activities might be things like moving a control rod into the reactor core, calculating a trajectory, or executing a continuous closed-loop control process. Activities are shown within the states using the "do" keyword, such as in state S2. Activities begin once a state becomes active (all entry actions have completed) and terminates once a state becomes inactive (prior to the execution of the exit actions).

Pseuodostates

Pseudostate are a kind of state vertex in the UML metamodel that represent transient points in transition paths within a state machine. Three different pseudostates are shown in Figure 4. The initial pseudostate shows the default state assumed upon entering a state context. The outermost one in the figure shows that when the object is created, the setupAction() event is executed and state S0 is entered. The initial pseudostate looks like a transition but has a ball at the origination end.

The conditional pseudostate is represented by a circumscribed "C" (alternatively, it can be represented by a small diamond). It is a notational shorthand for multiple transitions which share a common source state and triggering event. Different exiting transition segments are selected on the basis of guarding conditions. The conditional pseudostate is not a real state – if the transition is not taken, the triggering event is discarded and the original source state continues to be active.

The third pseudostate in Figure 4 is the terminal pseudostate. This is shown by a circumscribed "T" (alternative notation is a dot within a circle). This means that the object no longer accepts events and will be destroyed.

[2] Activities were not included in the original OMG UML standard, but are making a comeback in the latest revision.

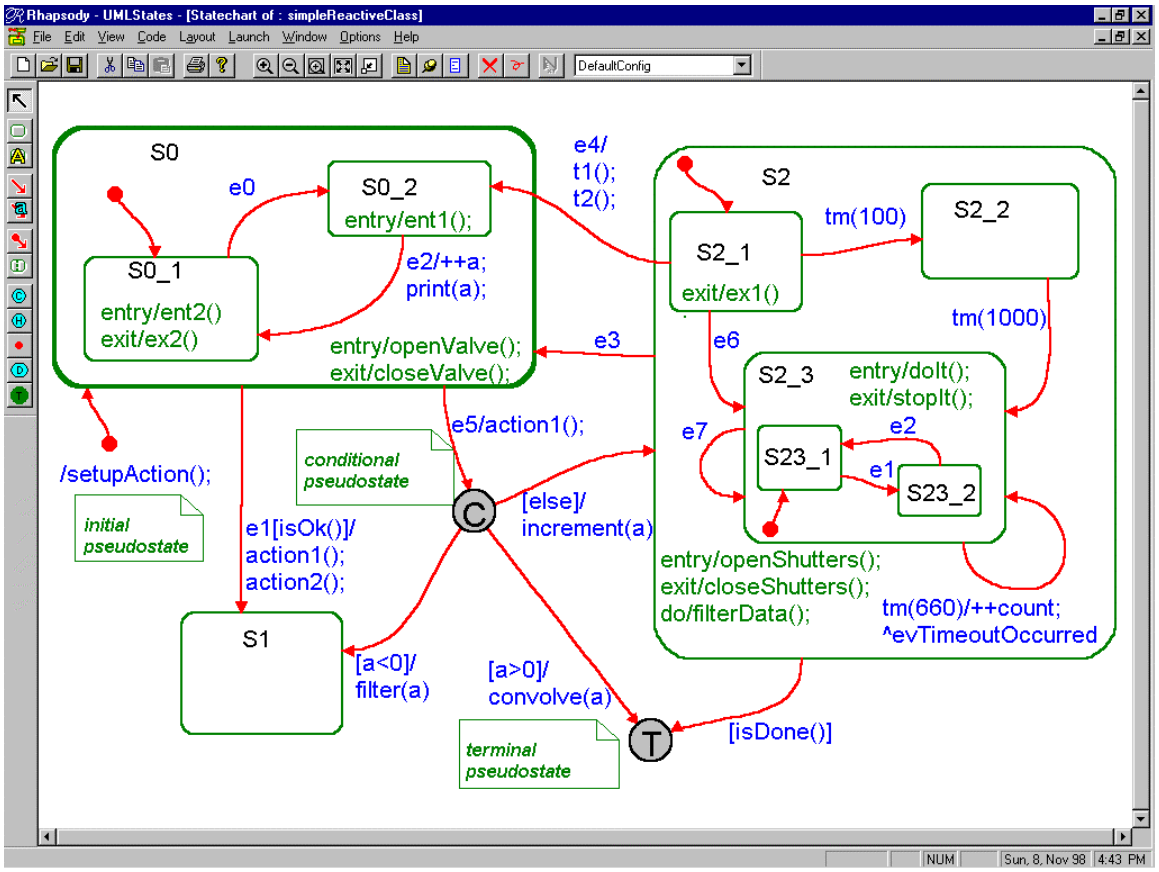


Figure 4: Basic Statechart Notation

Advanced Semantics of Statecharts

So far, we’ve seen the basic elements of statecharts. There are a number of more advanced features available to assist in modeling large or complex dynamic behaviors. The most significant of these is the *and-state*, but there are several more such as history pseudostate, state reactions, deferred events, and synchronization states. Most of these features are shown in Figure 5.

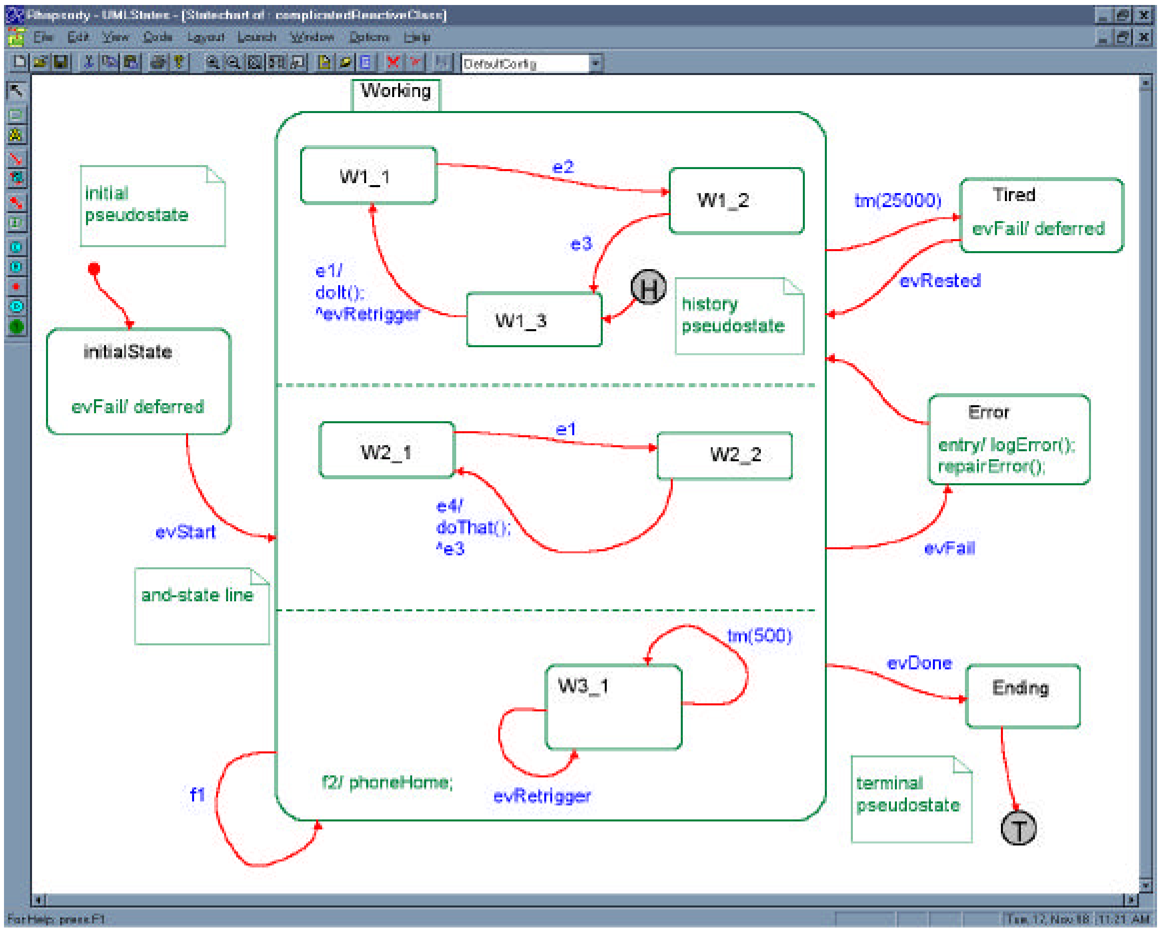


Figure 5: Advanced Statechart Features

The most noticeable about Figure 5 is the large state labeled *Working* that has three regions separated by dashed lines. These regions are called *and-states* (a.k.a *orthogonal regions*). Unlike the or-states discussed previously, when the object is in the Working state it must be in exactly one substate of each of the active and-states. Thus, the object might be in states W1_1, W2_2, and W3 *all at the same time*. This does not necessarily imply thread concurrency, but that is certainly one means of implementing and-states. Semantically, it means is that there are three independent aspects of the object, each modeled by a (sub) state machine while the object is in the Working state. These independent aspects are allowed to communicate and synchronize, but they do so in only a few, well-defined ways. All orthogonal regions accept events sent to the object, and may (or may not) respond to them, for example. In Figure 5, the first two orthogonal regions both accept and respond to the event e1. This is called broadcasting or (more properly) multicasting of events[3]. Another way that and-states communicate is that one orthogonal region may create an event as result of taking a transition that is consumed by

[3] In the UML, events are not broadcast. They apply to all currently active or-states within an object. They may also be sent (multicast) to an identified set of objects.

another orthogonal region. In the figure, when the top orthogonal region transitions from state W1_3 to W1_1 by accepting event e1, it creates the event evRetrigger which is consumed by the third orthogonal region. Similarly, when the second orthogonal region transitions from W2_2 to W2_1, it generates event e3, which may affect the top and-state (provided that it is in substate (W1_2). The third way that and-states communicate is through the use of guards. A common technique to synchronize and-states uses the IS_IN( ) operator (some people call it IN( )). This operation returns TRUE if another region is currently residing in the specified state. For example, the transition triggered by event evRetrigger could be constrained to only be taken if the second orthogonal region is in state W2_2 by adding the following guard to the transition:

evRetrigger[IS_IN(W2_2)]

Guards can be used in other ways to communicate among orthogonal regions as well. It is relatively common, for example, to use attribute ranges within guards (e.g. [x<10]). Since all orthogonal regions apply to the same object, they all have access to its attributes. Transitions in different regions may, for example, only take a set of transitions if x is less than 10.

Notice also the history pseudostate shown as the circumscribed H in Figure 5. This means that the superstate "remembers" its last active substate. Once visited, subsequent reentry of that superstate uses the last active substate as the initial substate. The reason that the history pseudostate points to a substate is to indicate the default initial substate, i.e. the substate to enter before that superstate has been visited the first time. The UML supports both shallow and deep history. Shallow history (the default) means that history applies to the current nesting context only – states nested more deeply are not affected by the presence of a history pseudostates in a higher context. Deep history applies downwards to all levels of nesting. Deep history is indicated with a circumscribed H* rather than a plain H.

There are a number of additional features of statecharts that we don't have room to discuss in any detail. States can have *reactions*, which are event responses that don't cause state changes. Reactions are distinct from transitions-to-self because the latter cause the execution of the state exit and entry actions. States may defer events using the *deferred* keyword. The list of events in the "event-list / deferred" clause will be held until the object transitions to a state in which they are no longer deferred. There are also other additions being made to the UML statechart metamodel within the Object Management Group, owner of the UML standard. One of these is the addition of "synch pseudostates" (similar to Petri net *places*) to allow synchronization of orthogonal regions.

Sample application

To illustrate how objects relate to state machines in the context of an embedded system, consider a Microwave oven application, the *Nuke-o-matic*. This device does several related things:

- In cooking mode, the user may (optionally) set the power level, and (mandatory) set the cooking time. The oven turns on the light, fan, and rotisserie motor and cooks the food for the specified period of time.
- Power level is implemented by cycling the microwave emitter on and off. Power level 10 means that the emitter is on all the time, while power level 1 means that it is on 1/10 of the time. If not set, in cooking mode the default power level is 10.
- The oven only turns on the emitter when the door is closed (important for repeat business) and stops if the door is opened.
- Light goes on either when the door is opened or when the oven starts to cook or thaw food.
- The oven has a thaw mode in which the default power level is 3.
- The oven has a timing-only mode in which it acts like a count-down timer. In this mode, it does not turn on the light, fan, or rotisserie motor.

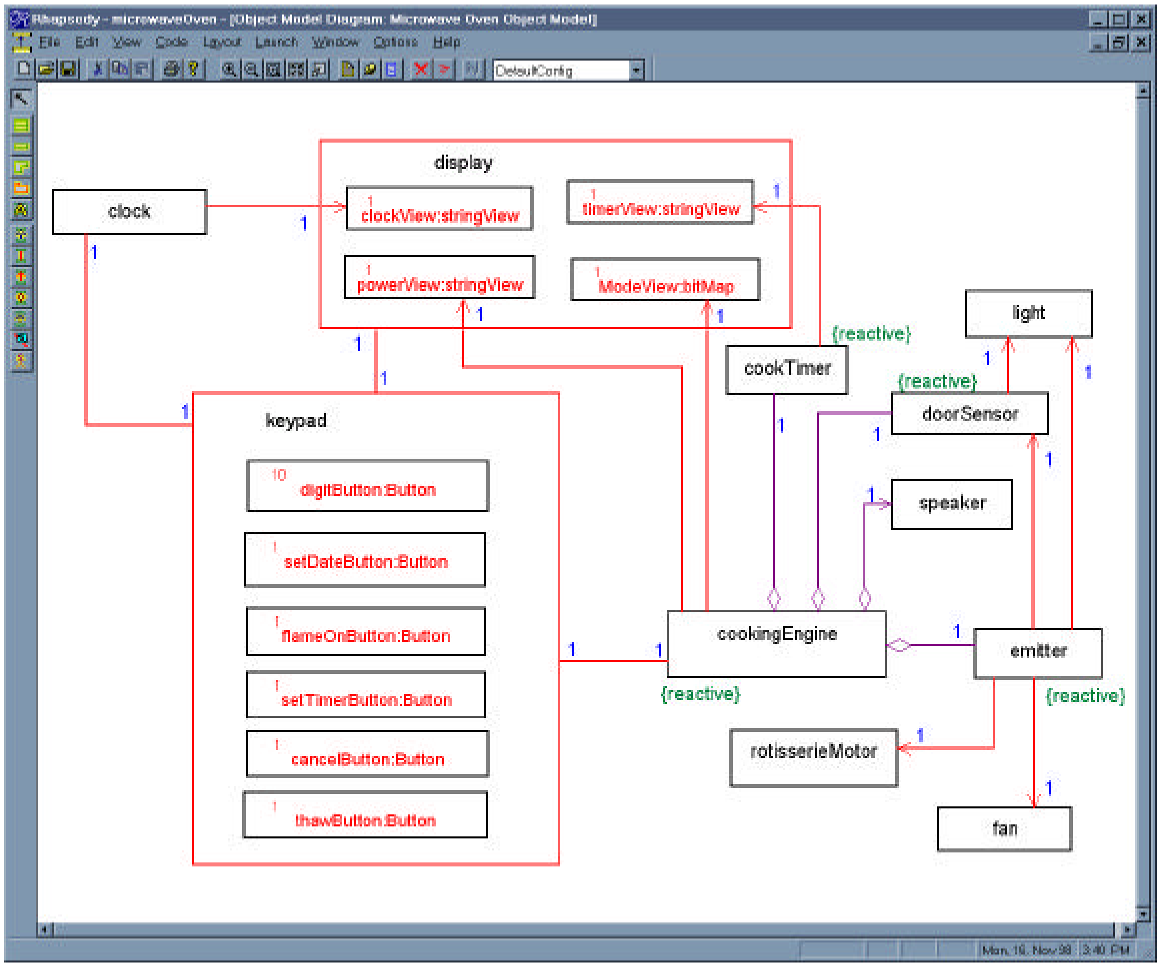


Figure 6: Nuke-o-matic Microwave Oven Class Diagram

The class model for the Nuke-o-matic is shown in Figure 6. In this example, the primary strategy use to identify object classes was to identify physical devices, but other strategies are also possible. The classes drawn within other classes have a composition relation with their enclosing class (a strong form of aggregation). The composite classes in Figure 6 are the *display* and *keypad* classes. The composition indicates that the composite is explicitly responsible for the creation and destruction of the contained objects. The *keypad* includes a number of different keys; individual mode-setting keys to select the operational mode (setting the date and time, selecting time-only mode, select thawing mode, and so on (cooking mode is the default), to start the selected operation (the *flameOnButton*) and to cancel the current or pending operation. Other classes have an aggregation relation, indicated by a diamond on the "whole" end of the relation. This is also an ownership relation, but somewhat weaker than composition. The other relations in the diagram are associations, indicating loosely coupled classes that send messages to each other in order to collaborate.

Several of these classes are reactive, indicated on the diagram with a simple note. The reactive classes in this example are the *cookingEngine*, *cookTimer*, *emitter*, and *doorSensor*. The statecharts for these are shown in Figure 7 through Figure 10. Actions specified on the statechart are shown with the name of the object and the operation. For example, in Figure 8, the *Working* state has a number of entry actions. In this example, a simple convention is used to name the pointers implementing the association -- add "its" to the name of the class at the other end of the association. Thus, the statement

```
entry/ itsLight->turnOn();
itsFan->turnOn();
itsRotisserieMotor->turnOn();
```

defines the entry actions for this state. In this case, the turnOn() operation of the light, fan, and motor are all executed. The GEN(event-name) operation is used to indicate the creation of an event targeted to the specified object (rather than the carat syntax). For example, in Figure 7, we see the state machine response to the *evDoorOpened* event while the object is in the *Active* state:

```
evDoorOpened/
itsEmitter->GEN(evPaused);
itsTimer->GEN(evPaused);
```

The means that the *evPaused* event is generated for both the *emitter* and the cookTimer with which the *cookingEngine* associates.

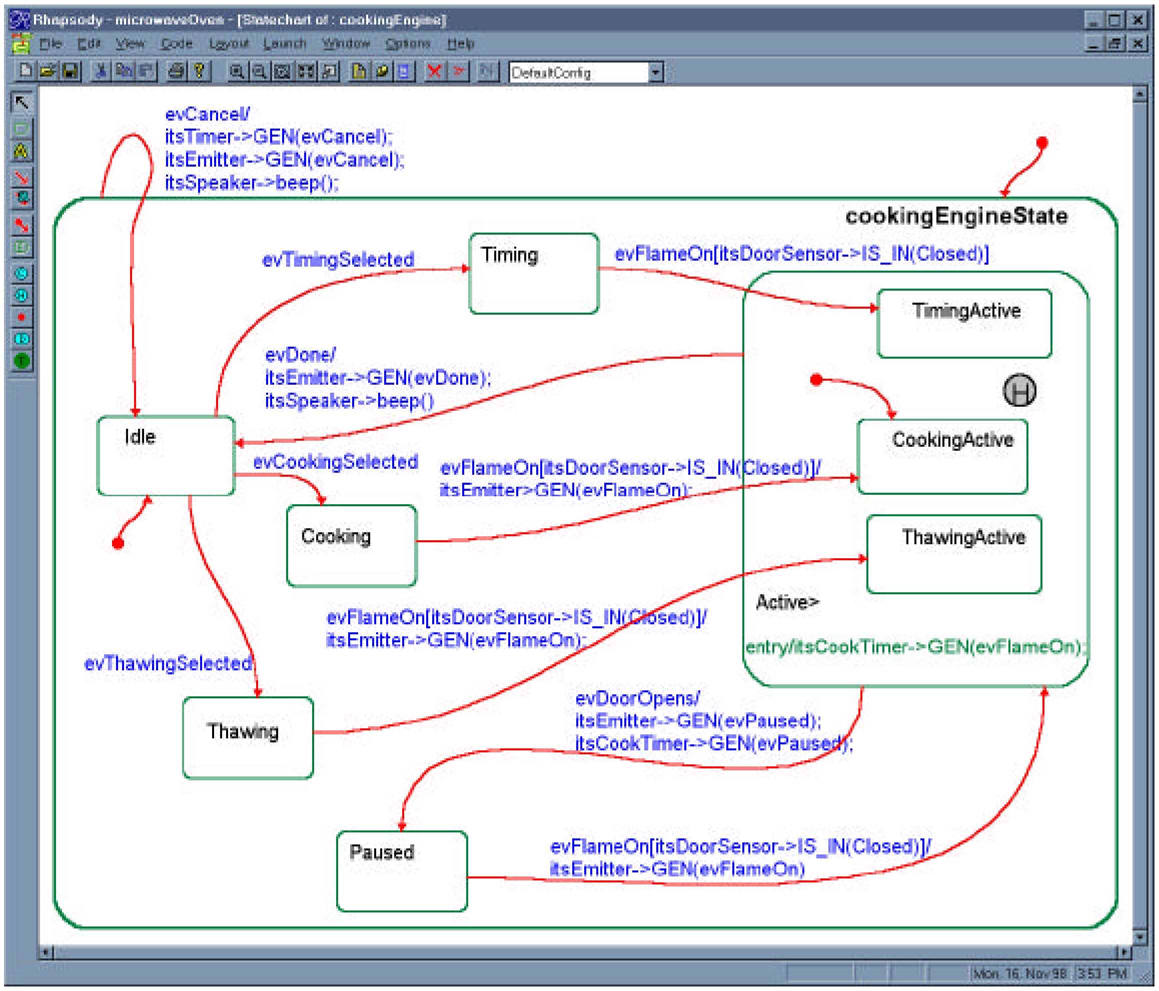
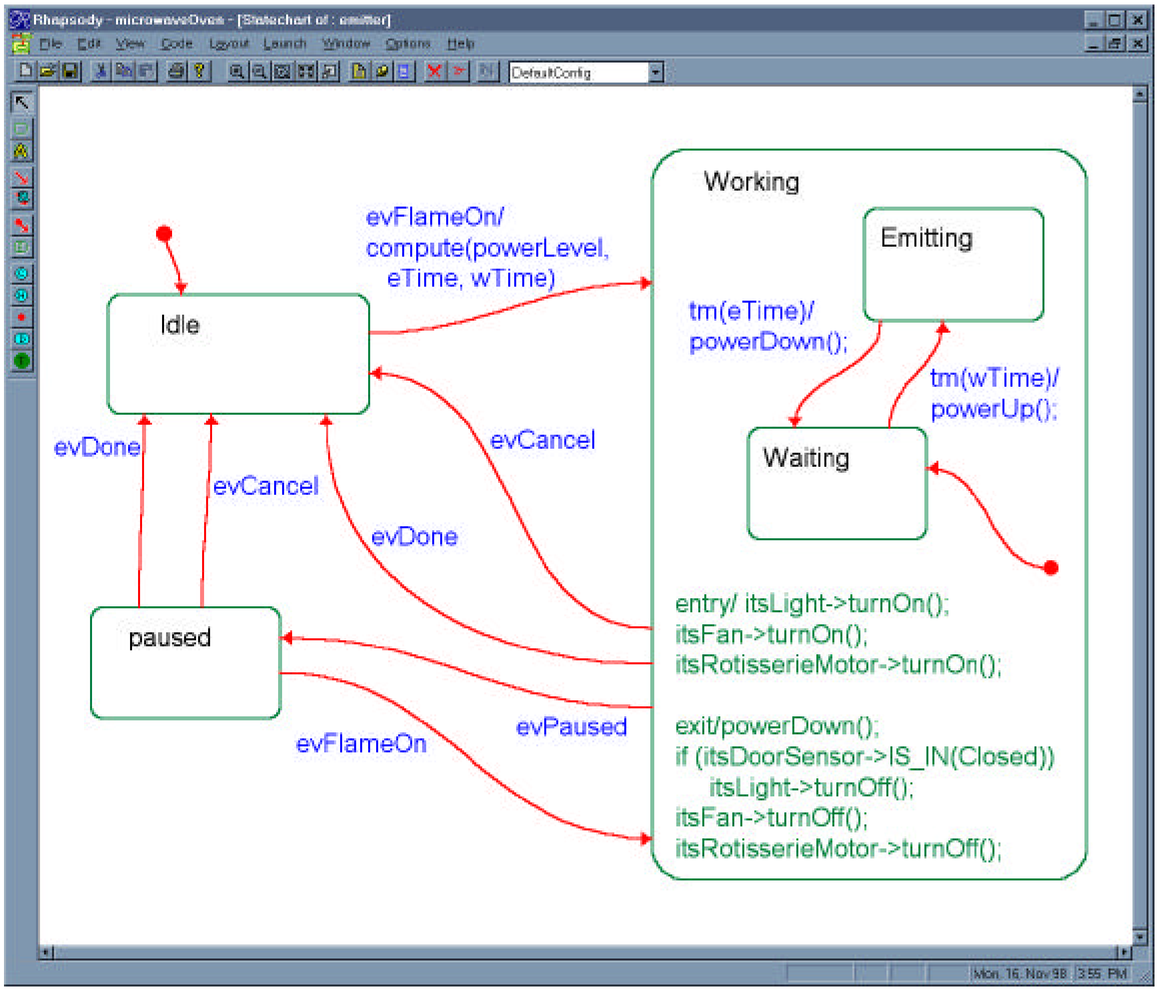

Rhapsody - microwaveOven - [Statechart of : cookingEngine]
File Edit View Code Layout Launch Window Options Help
DefaultConfig
evCancel/
itsTimer->GEN(evCancel);
itsEmitter->GEN(evCancel);
itsSpeaker->beep();
cookingEngineState
evTimingSelected
Timing
evFlameOn[itsDoorSensor->IS_IN(Closed)]
TimingActive
evDone/
itsEmitter->GEN(evDone);
itsSpeaker->beep()
H
CookingActive
Idle
evCookingSelected
evFlameOn[itsDoorSensor->IS_IN(Closed)]/
itsEmitter>GEN(evFlameOn);
Cooking
ThawingActive
Active>
evFlameOn[itsDoorSensor->IS_IN(Closed)]/
itsEmitter->GEN(evFlameOn);
entry/itsCookTimer->GEN(evFlameOn);
evThawingSelected
Thawing
evDoorOpens/
itsEmitter->GEN(evPaused);
itsCookTimer->GEN(evPaused);
Paused
evFlameOn[itsDoorSensor->IS_IN(Closed)]/
itsEmitter->GEN(evFlameOn)
Mon 16 Nov 98 3:53 PM

Figure 7: Nuke-o-matic cookingEngine Statechart



```
entry/ itsLight->turnOn();
itsFan->turnOn();
itsRotisserieMotor->turnOn();

exit/powerDown();
if (itsDoorSensor->IS_IN(Closed))
    itsLight->turnOff();
itsFan->turnOff();
itsRotisserieMotor->turnOff();
```

Figure 8: Nuke-o-matic emitter Statechart

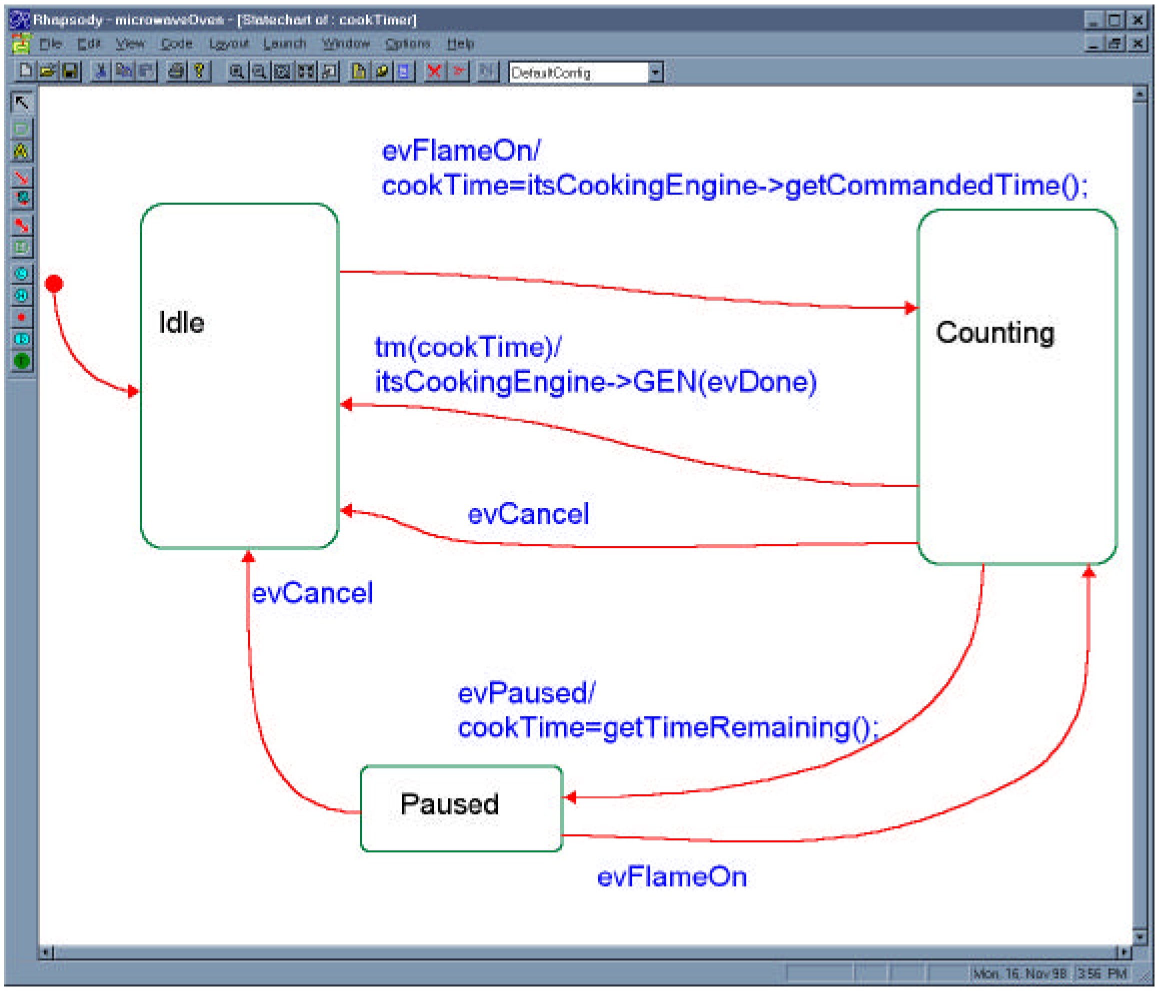

Rhapsody - microwaveOven - [Statechart of : cookTimer]
File Edit View Code Layout Launch Window Options Help
DefaultConfig
evFlameOn/
cookTime=itsCookingEngine->getCommandedTime();
Idle
Counting
tm(cookTime)/
itsCookingEngine->GEN(evDone)
evCancel
evCancel
evPaused/
cookTime=getTimeRemaining();
Paused
evFlameOn
Mon 16 Nov 98 3:56 PM

Figure 9: Nuke-o-matic cookTimer Statechart

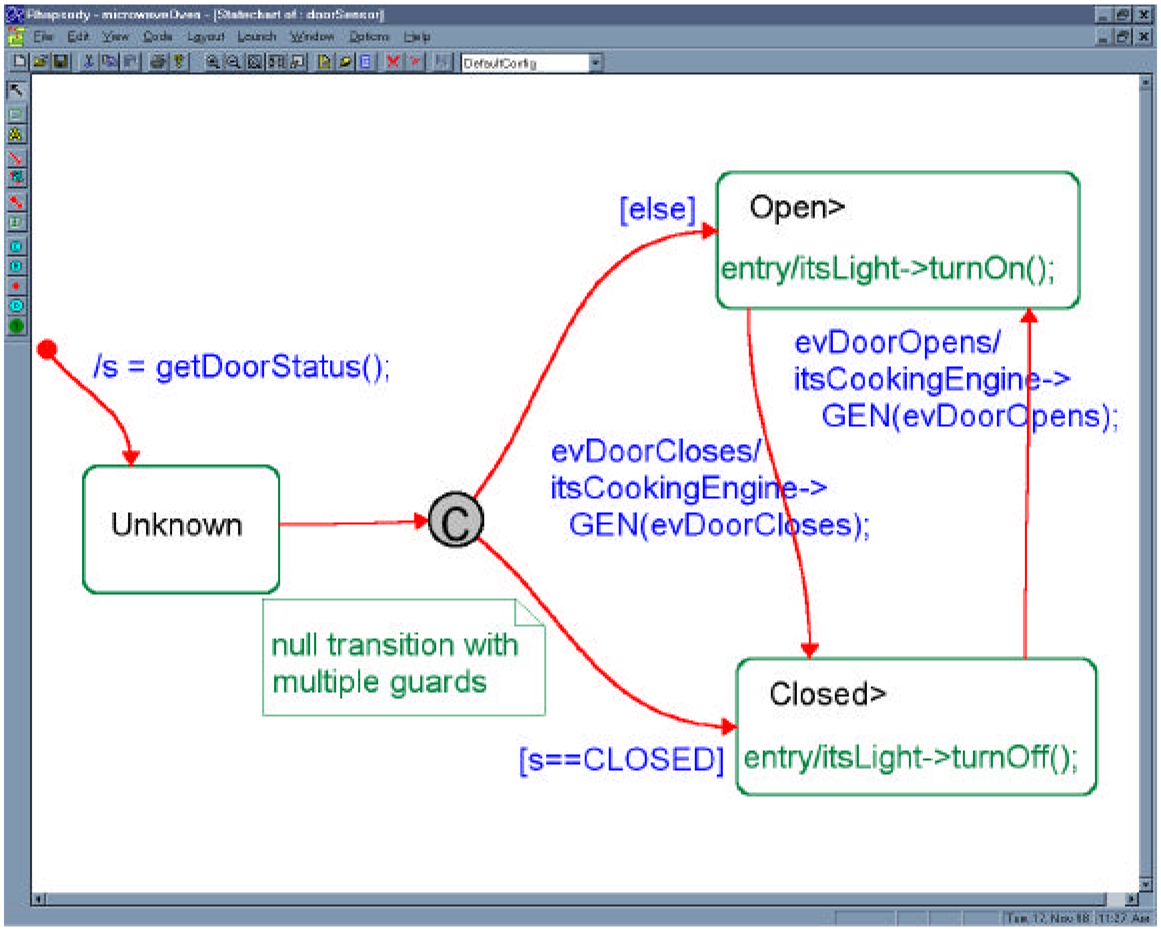


Figure 10: Nuke-o-matic doorSensor Statechart

We can see that the reactive objects in the Nuke-o-matic collaborate in three primary ways: the same event is sent to several objects, events are generated as a result of taking an event (i.e. changing state) in another object, and guard conditions.

Summary

Statecharts are a powerful visual formalism for capturing complex behavior and apply well to both functionally decomposed systems and to object-oriented ones. Objects are composite entities consisting both of information (attributes) and operations that act on that information (methods). State machines constrain the execution of those operations to better control and understand the object behavior. A state machine is defined by a set of existence conditions (states), event responses (transitions), and actions that are executed as we change state or take a transition. Statecharts add a number of useful extensions to the traditional flat Mealy-Moore state diagrams, such as nesting of states, conditional event responses via guards, orthogonal regions, and history. Although these extensions are mathematically equivalent to Mealy-Moore state machines, statecharts can be made much more parsimonious and vastly easier to understand, especially for complex state machines.

About I-Logix Inc.

I-Logix Inc., founded in 1987, through the collaboration of Dr. David Harel, the developer of Statecharts and Dr. Amir Pneuli a recipient of the Turing Award, is now the third largest worldwide provider of design automation solutions for developers of complex real-time embedded systems and software. I-Logix's award winning product line supports the entire design process through an iterative approach which enables system engineers to move rapidly to system prototypes and software engineers, to final product iterations.

Through this methodology, I-Logix is creating a paradigm shift in how complex embedded systems are developed by reallocating development resources from time spent writing and testing code to time spent designing and validating behavior. Supporting the *"I-Logix Way"* of development are such clients as JPL, NASA, Boeing, Daimler-Benz, General Motors, Roles Royce, Motorola, TRW, Otis, TRANE, and others.

References

Douglass, Bruce Powel *Real-Time UML: Developing Efficient Objects for Embedded Systems* Reading, MA: Addison-Wesley, 1998

Booch, Rumbaugh, Jacobson *The Unified Modeling Language User Guide* Reading, MA: Addison-Wesley, 1998

Douglass, Bruce Powel *Doing Hard Time: Using Object Oriented Programming and Software Patterns in Real Time Applications* Reading, MA: Addison-Wesley, March 1999

Harel, David *Statecharts: a visual formalism for complex systems* in *Science of Computer Programming* 8 (1987), 231-274.

Bertrand Meyer, *Object-Oriented Software Construction Second Edition* Upper Saddle River, NJ: Prentice Hall, 1997

I-Logix Inc.
3 Riverside Drive
Andover, MA 01810
Tel: 978 682-2100
Toll Free: 888 8ILOGIX
Fax: 978 682-5995
E-mail: info@ilogix.com
http://www.ilogix.com

European Headquarters
I-Logix UK Ltd.
1 Cornbrash Park
Chippenham, Wiltshire
England SN14 6RA
Tel: +44 1249 446-448
Fax: +44 1249 447-373
E-mail: info_euro@ilogix.com