Structural Design of a Software System using Statechart and Activity-chart in Statemate Tool

Kamran Sartipi

Assistant Professor
Dept. of Computing and Software
McMaster University
Hamilton, Ontario
Canada L8S 4K1

Abstract

This report introduces Statemate as a CASE tool for developing large reactive systems. Reactive systems perform different actions in response to the input signals and usually receive a large number of inputs. This characteristic complicates the design of the behavior control parts of the system. Statemate provides statecharts as means for designing behavior parts and activity-charts for designing functional parts of a system. These tools assist the designer to separate the important design concerns with respect to the system functionality and behavior. This report starts with a brief introduction to the Statemate's language and continues with description of the design techniques used for developing a large reactive system with Statemate. The design of a fast-food restaurant system is used as an example of a large system to illustrate the various design aspects such as: different design styles, hierarchical structure, data and control flow, user-interface, managing parallel processes, and instantiation. The last part of the report evaluates the Statemate tool against a number of criteria.

1 Introduction

Statemate, a *CASE* tool for designing of large *Reactive Systems*, is a product of the i-logix incorporation. The term *Reactive Systems* was first used by Dr. Lavi at Israeli Aircraft and later used in Statemate description by Dr. David Harel. Reactive systems receive a large number of external and internal signals of the form *Event* or *Condition*, hence they are sometimes called *event driven* systems. The system reacts to the signals and performs different tasks accordingly. The examples of such systems include: avionic systems, telephone and communication controller, robots, etc.

Statemate provides facilities for *Structured Analysis and Design*, and is a collection of different tools that supports a complete design life-cycle from requirement-analysis to prototyping, designing, and code-generation. It benefits from different graphical editors for design task which aid the designer to develop the system using its functional and behavioral models. Statemate provides the following facilities in one package:

- **Graphical Editors** are design tools for specifying the system in *Activity-Charts*, *Statecharts*, and *Module-Charts*.
- **Project Management** provides an environment for a group of system designers to work in a team for developing a system. It manages the authority to access the shared data base (*Data-Bank*) among the design team-members.
- **Version control** keeps track of the changes of each module in the system and generates different versions for each.
- **Check Model** checks the data-flow and control-flow diagrams for consistency and completeness.
- **Data Dictionary** defines the whole elements of the system and searches through them.
- **Data-Base Query** offers a full spectrum of query models for querying the attributes of the elements in the system.
- **Simulation** simulates the system functionality at different levels of abstraction i.e. prototype and design.
- **User Interface Design** uses a dedicated *Panel* editor to define the required displays and buttons for input/output tasks.
- **Code Generation** produces the source code in *C* or *Ada* from the designed system.
- **Document Generation**.

In this report we are interested in modeling a reactive system using three aspects of the Statemate language: *Activity-charts*, *Statecharts*, and *Module-charts*. A multi-station restaurant system with several communicating units is used as our design example. Statemate has facilities for designing a system in different levels of abstractions which assists in designing a system in top-down or bottom-up styles. A precise system specification is required to ensure the production of a correct system. This specification is created from the initial system requirement proposed by the customer and further investigations of the designer. Provided that a detailed specification is available, the design can be started from bottom. The design of our restaurant system is started by implementing the basic functionalities of various units of the system i.e. order-taking, assembling, food-preparation, and inventory. The activity-charts are means for defining the system functionalities. Defining the data-structures and input/output characteristics of the system are the next design efforts. The behavior of the

system in response to the incoming or internal signals is demonstrated by statecharts. Statecharts react to the signals and control the functionality of the system manifested by the activity-charts. This is a typical behavior of reactive systems.

Statemate contains the required mechanisms for gathering the small pieces of a design functionality in a group to make a higher level of abstraction. The reverse design style i.e. top-down, is also feasible. The system is first designed in its highest level of abstraction. Then, using functional decomposition of the modules, lower layers of the design are developed.

In complex systems such as this restaurant system, several concurrent and communicating modules exist. Typical problems in such a systems include: inter-process communication (IPC) and process synchronization. To illustrate the IPC between different charts of a design, a simple example of producer and consumer is provided. Synchronization is another issue of the concurrent charts which must be resolved. A synchronization technique which is employed in our restaurant system is presented. Reusability of the existing modules is a common practice. The concept of generic-chart and instantiation in Statemate allows the use of multiple instances of a chart. A simple example describes this feature. Most of the units in the restaurant example have several stations which use the instantiation concept.

This report is structured as follows: The next section is allocated to a brief description of the Statemate's language which consists of statecharts and activity-charts. In section **??**, various design aspects such as: system specification, system architecture, data and control flows, data-structure, and user interface are described. Section 3 of the report describes the design peculiarities of a concurrent reactive system with emphasis on the two design styles: *Top-Down* and *Bottom-Up*. In section 3, the concurrency related issues such as synchronization and interprocess communication are discussed. In section **??** we evaluate the Statemate tool against a number of criteria which are devised for evaluation of graphical design tools. Finally section 4 provides a conclusion to the report.

2 Language of Statemate

The language of Statemate consists of three aspects:

Activity-charts: These charts manifest the functional decomposition of the system in a hierarchy of charts, each at a different level of abstraction. These charts define the functionality of the system and are considered *Data-Flow* diagrams. Their tasks include transportation and manipulation of the system variables. They are concerned with **what** tasks the system must execute.

Statecharts: Statecharts present the behavior of the system and control its functionality. Statecharts manage the behavior of the system by sequencing the order of execution of different functions of the system. These charts are considered *Control-Flow* diagrams.

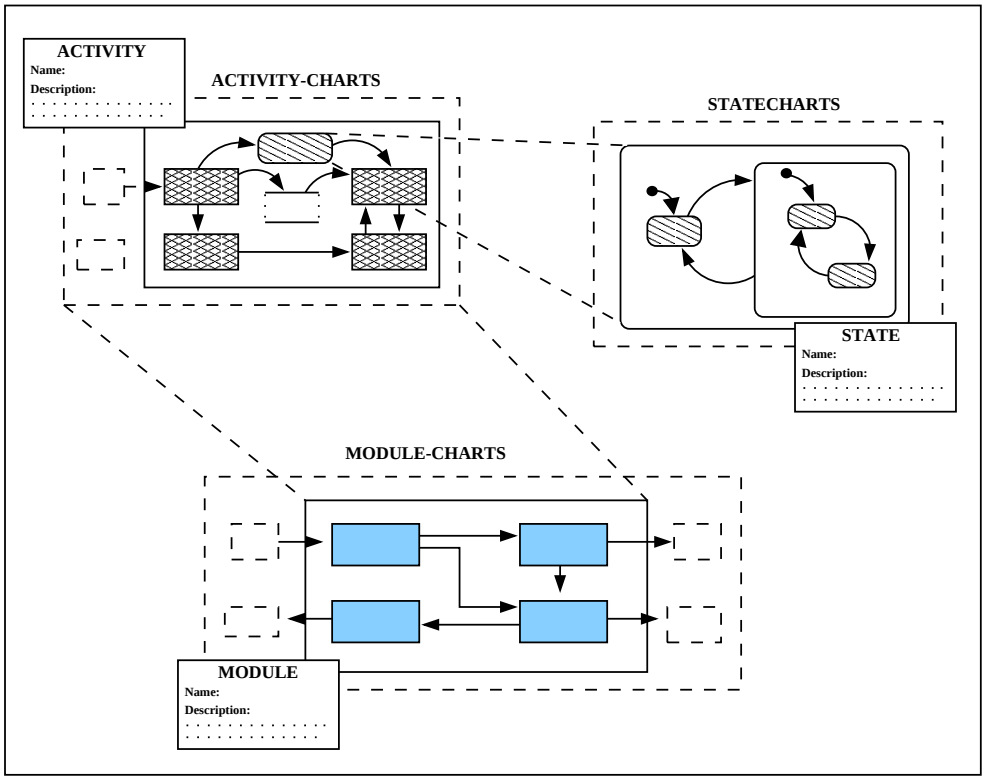

Figure 1: The *Language* of the Statemate consists of three graphical concepts: Statecharts, Activity-charts, and Module-charts.

They illustrate the concepts of causality, concurrency and synchronization in the system. In general they are concerned with **when** a particular task should be executed. Statecharts and activity-charts demonstrate the *Conceptual Model* of the system.

Module-charts: These charts define the *physical* connections among the various modules in the system and provides the *structural view* of the system. It answers the question of **how** different modules of the system are interconnected. The three views of the system-model and their relationships are illustrated in Figure 1. Different attributes of each chart in the system are kept in data-dictionary forms which are shown in the Figure.

In the following sections the important features of the Statemate semantics are presented. The semantics of the Statemate language is described in its conceptual model: statecharts and activity-charts.

2.1 Language of the Statecharts

Statecharts to some extent are *state-transition-diagrams* with increased capabilities, such as, multi-level states, decomposition of states, concurrent states, and synchronization. Below, some important concepts are described.

2.1.1 Elements of a statechart

The elements of a statechart consist of:

State: A statechart consists of several states which are represented by rounded boxes. States show the distinct configurations of the system. Three kinds of states exist: or-states have sub-states (hierarchical), and-states have concurrent components (parallel execution), and basic-states have no sub-states. States are used for several purposes:

- A state can produce time synchronization by creating unit-delays in the system. In this form we add a state between two states.
- A state can be used to start one or more activities in an activity-chart. After assigning this task to a state (using data dictionary), its name is suffixed with the sign ">".
- A state can be a *reactive-state*, which has some *static-reactions* associated to it. Static reactions are reactions of the system to an event within a particular state, The syntax of the reaction is:

  trigger/action;;
  trigger/action;;

  It is a list whose components are separated by two semi-colons. Each one has a syntax of a transition label. Static-reactions are defined using a data dictionary. The name of the reactive-state is then followed by a ">" sign. This feature is demonstrated in section 3.2.1
- A state can represent an off-page chart. In this case, its name is prefixed by the sign "@" and is called *box-is-chart*. (see section 2.1.6)
- A state can represent an instance of a generic-chart, in which case its name has special format (see section 3.3).

Transition: Transition is the movement from one state to another, and is represented by an arrow. The label of the transition has the following format:

Event[Condition]/Action

Events are edge sensitive elements of the system. They are effective only at the time-unit when they are triggered, and will disappear at the next time-unit. Various events are generated as a result of the following actions:

- Pressing of a *button* in the panel, which is bound to an event.
- Entering or Exiting of a state.
- Activating or Stopping of an activity.
- Changing of data or contents of an array.
- Time-out after that an event is occurred.

Conditions are level sensitive elements of the system. As long as the condition is true the associated transition is allowed. If the condition and event are both used in the label of an arrow, the transition is allowed only when the condition is true and the event is triggered. The value of a condition becomes true by one of the following actions:

- Pressing of a button in a panel, which is bound to a condition.
- Being in a state.
- When an activity is active.

Actions manipulate some elements in the system after which a transition is performed. They are summarized below:

- Trigger an Event or change a Condition.
- Assignment of an expression to a variable.
- Starting or Stopping of an Activity.
- If, then, else construct.
- While and For loops.

Step: Step is a transition of the system from one configuration to another and involves the execution of one or more transitions. If no transitions can be executed, the system is said to be in a stable configuration. Step is executed in zero time. The values associated with transition actions are changed only at the end of the step and the generated events may be sensed only at the step that immediately follows the current one.

In the next part, some of the most important concepts of the statecharts are described. The Figures are the snapshots of the charts when their operations are simulated with the *simulator* tool. The thicker arrows represent the currently active transitions when simulating the chart and help us to pinpoint the important features of the examples.

2.1.2 Statechart Timing

A transition may occur on a time-out basis. This type of transition has the label of the form *tm(e,n)* in which "e" is any event of the above described form, and n is the number of system time-units. This transition will occur n time-units after the occurrence of event "e". Figure 2 illustrates the use of delayed transition. When the state *idle* becomes active the event

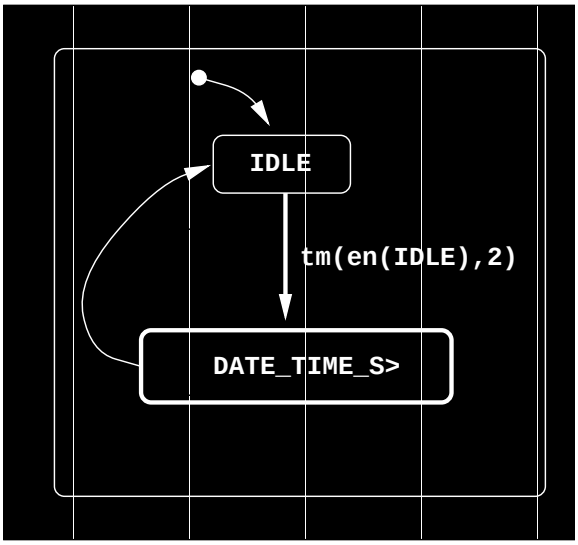

Figure 2: Timing in the *Statechart*. Delayed transition occurs after n time-units.

en(idle) is triggered (state *idle* is entered). After 2 time-units the transition will be triggered and the system moves to state *date-time-s*. At the next step the *idle* state is active again. In simulation of a chart, steps are performed at zero-time, but the time-out transitions are dependent on time not on steps. Therefore, to see the effects of time-based transitions, the simulation should be performed in auto-run mode, not single-step, to let time proceed.

2.1.3 Statechart Hierarchy

Hierarchy is the decomposition of a state into more basic states. It reduces the number of transitions and clarifies the system behavior. The statechart in Figure 3 has the following interpretation: When a transition is made to the outer edge of a state which is further decomposed, the sub-state containing the default entrance is entered (thick arrows in Figure). When a transitions from the outer edge of a state, which is further decomposed, is triggered, all states inside that state becomes inactive. e.g. if the state B is active and condition C1 is false, transition with label [not C1] will occur and B becomes inactive. When a transition leaves the edge of a sub-state, the system must be in that sub-state for the transition to occur. e.g. in order to trigger transition "E2", state A must be active. When a transition is made to the edge of a sub-state, the state is entered whether or not that state is the default entry state (not shown in Figure).

2.1.4 Statechart Concurrency

The notion of being in more than one state at any given time is handled through the principle of concurrency. The statechart in Figure 4 illustrates the and-line construct (dotted

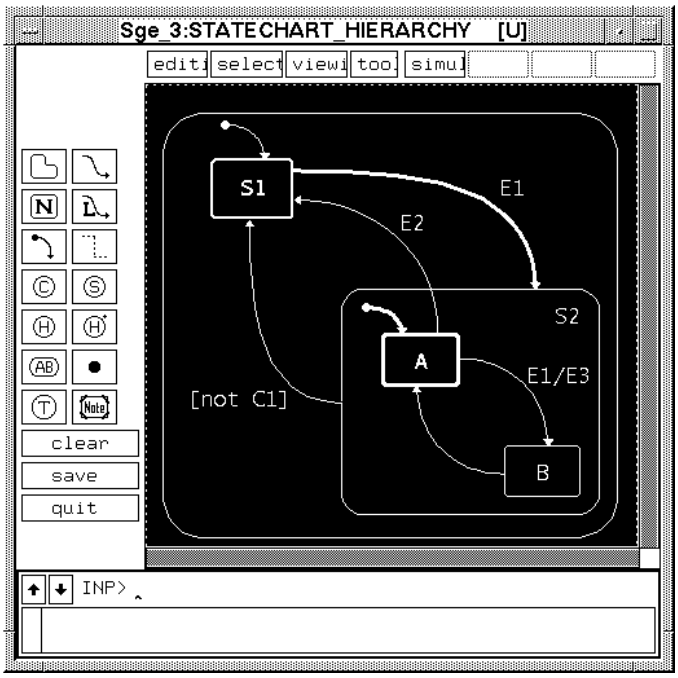

Figure 3: Hierarchy in the *Statechart*. State S2 is decomposed into states A and B.

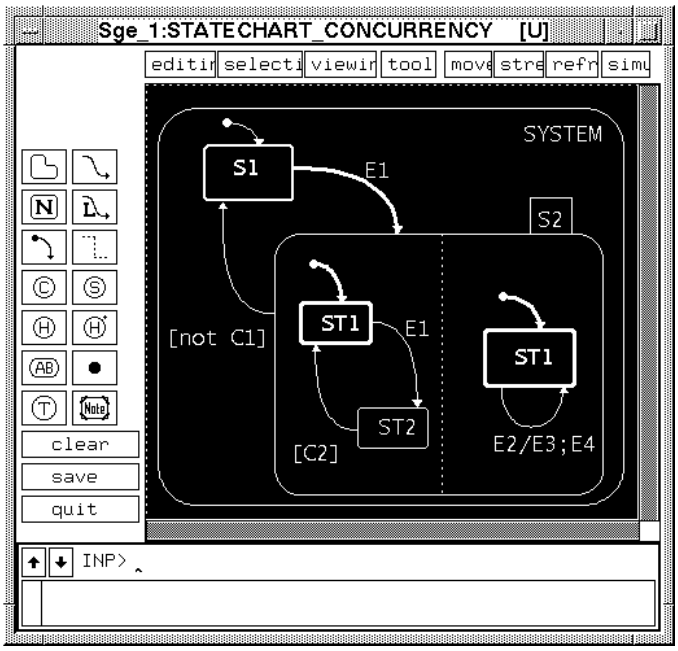

Figure 4: Concurrency in the *Statechart*. Each state can be converted to an *and state* with many concurrent elements.

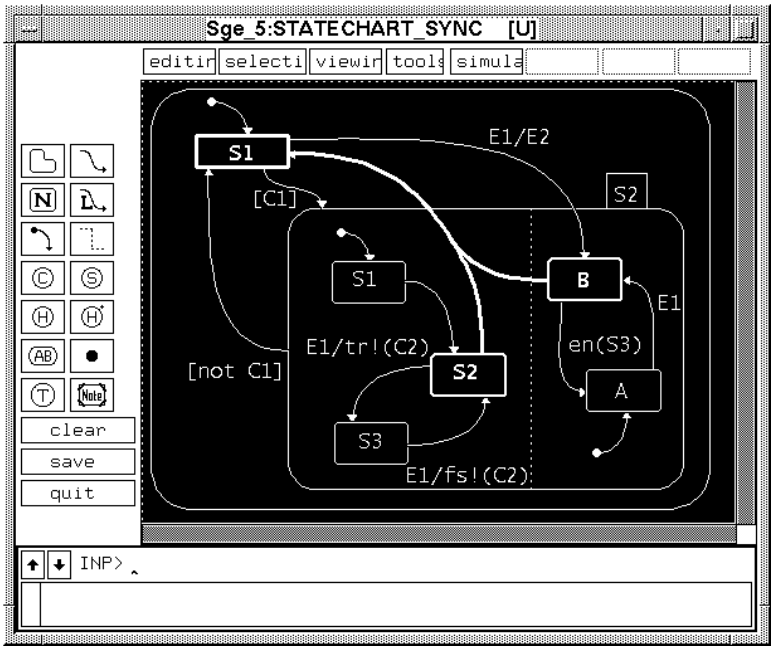

Figure 5: Synchronization in the *Statechart*. Several states can be synchronized using the *joint transition* (thick arrow).

line), which denotes concurrency. State S2 is an and-state. The and-state may have many concurrent states. When an and-state is entered, all of its concurrent sub-states are entered (thick arrows in Figure). This is true whether the and-state is entered by a transition to its outer edge or by a transition to any of its descendants. When an and-state is exited all of its concurrent sub-states are exited. This is true whether the and-state is exited by a transition from its outer edge or by a transition from any of its descendants.

2.1.5 Statechart Synchronization

The combination of hierarchy and concurrency simplifies the modeling of large, complex system. Concurrent processes (or individual statecharts) rarely exist independently. It is therefore, necessary to synchronize their interaction. A *joint transition* is one which combines two or more transitions with the same label (Figure 5). This transition will only occur when all states that share the joint transition are entered and the conditions for transition are met. This is shown by the thick arrow. The resultant transition can generate an event which is used for synchronizing of concurrent processes in the scope of the statechart or can be used to synchronize different statecharts.

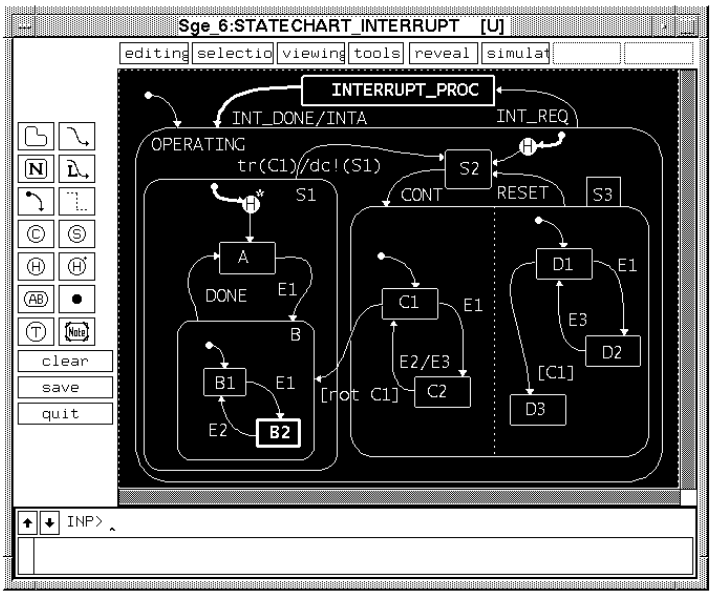

Figure 6: Putting all details of the hierarchical states in a statechart, makes the chart messy and difficult to understand.

2.1.6 Statechart Declutter & Merge

As the complexity of the design increases, the charts become messy and difficult to understand. The decluttering mechanism permits the detail of any box to be defined in a separate off-page chart. Its reference box is named with the prefix "@". *Diagram connectors* are used to mark the entry and exit points of the transitions in the defined chart. In order to illustrate the significance of decluttering and its effect in separating the different levels of hierarchies, we declutter the chart in Figure 6. Three states S1, C, and D are decluttered, and the result is shown in Figure 7. As is seen, the name of each decluttered state is prefixed by the sign (@) which means the state is the representation of a chart and is called *box-is-chart*. For each decluttered state a separate chart will be generated and the link between the new generated chart and the decluttered state is established. *Logical Diving* to a decluttered state is a concept that causes the system to display the associated chart so that you can see the details of the state.

Merge, the opposite concept of *Declutter*, causes the separate charts to be combined into one chart.

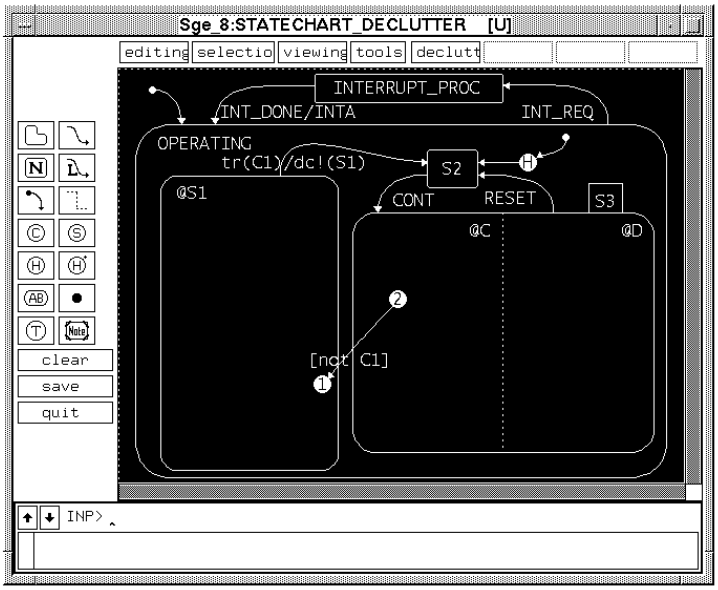

Figure 7: Decluttering in the *Statechart*, allows the details of the boxes to be defined in the off-page charts.

2.2 Language of the Activity-charts

An Activity-chart is a data-flow diagram with elements (syntaxes) such as: activities (edged boxes), information flow (solid arrow), control flow (dashed arrow), data repository (box with dashed sides). A typical activity-chart is shown in Figure 8. The relation of the state-charts and activity-charts are through a special activity called *control-activity* (box @system in Figure). This activity is in fact another statechart which controls the activation of its sibling activities.

Statemate has three kinds of activities which differ in the way that they terminate. The activation of all activities are controlled by activation of its parent activity or its sibling control activity.

Procedure-like activity: This activity performs a set of actions upon its start and then stops. It is active for one simulation step.

Reactive-controlled activity: This activity has a control-activity. It can be active for several steps, but only stops when its control-activity terminates.

Reactive-self activity: This activity is similar to the above activity, but it can also terminate by itself.

Other important concepts in activity-charts are as follows:

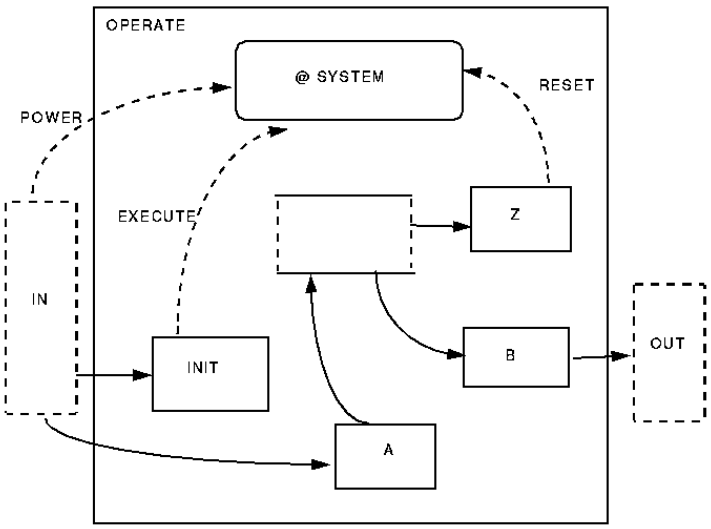

Figure 8: Activity-chart in Statemate. Edged-boxes are activities and rounded-box is the control-activity.

Mini-Specification "Mini-Spec": The activity mini-spec is a textual behavioral description which consists of a list of *reaction/actions* or *actions*. The mini-specs are defined through a data-dictionary form and become active when the associated activity is made active, and stopped when the associated activity is stopped. When a mini-spec is defined for an activity, the sign ">" will be appeared after the name of the activity. This concept is used in section **??**.

Context-Variables: Actions in transitions or mini-specs are performed simultaneously, i.e in transition label /A;B;C, actions A, B, and C occur at the same time. *Context variables* are variables whose value can change within a step. These variables have a dollar sign "$" before their names. Context variables permit us to have *For* or *While* loops within a single simulation step. For example the only way to perform a loop in a mini-spec is as follow:

```
For $x in 1 to 5
loop
  xarray($x):=$x;
end loop;
```

This loop sets the 5 elements of the xarray() to the integers 1 to 5.

Box-is-chart: Activity-charts like statecharts benefit from the concept *decluttering*, and hence complex activities can be decluttered to produce off-page charts. The decluttered activity is called *box-is-chart* and its name is the name of the activity which is prefixed by the sign "@". This concept is also used in section **??**. It is immediately observed that this method of hierarchical representation of the activities hides the unnecessary details of the design at a certain level of abstraction.

Scoping of elements: Statemate tool uses a versatile *data-dictionary* which acts as a data base and contains the information from all types of elements in the system. Data-items, events, and conditions should be explicitly defined in the data dictionary, where they are bound to a statechart or activity-chart. This chart is the scope of the element, and the element can be shared among all activities or states of that chart and its offspring. This scoping permits the designer to define variables to be accessed by different groups of charts, some (global) variables accessible by all charts, and some (local) variables accessible to smaller group of charts.

2.3 User Interface Design

Designing of an interface to a software system has always been one of the essential design efforts. Creating a mouse driven, menu driven, windowing system interface is desired in any system. Existence of modern *Window Systems* such as *X11* and versatile *Window Managers* such as *OSF/Motif* facilitate the development of sophisticated man-machine interfaces. These facilities provide a series of default features for the design tool which eases the design task. Still the main task of the design tool is to employ these available mechanisms and provide some higher level means for system developer to easily design a man-machine interface.

The variety of the interface design widgets define the richness of the syntax of the user-interface designer. The semantics of the interface, on the other hand, express interactions between the design widgets in the interface and design elements in the system. For example, semantics defines how a particular display in the user-interface screen can communicate with a running process in the system. Various methods of triggering an event from the keyboard of the system is another example.

The *Statemate* integrated tool, has a rather sophisticated built-in user interface design tool called the *Panel Graphic Editor*. This panel-editor uses the *X11* window system and *Motif* or *Open Window* window manager to provide flexible window manipulations such as: expanding, shrinking, and moving the window, utilizing a spectrum of colors and fonts. Using the Network features of the *X11* window system such as *Remote Display*, enables Statemate tool to set up the user-interface of the system in any host computer or *X-Terminal* across the Network, independent of the location of the running program. This allows to easily design multi-screen systems.
In Figure 9 the *Panel Graphic Editor* of *Statemate* is shown. A part of the *order taking* panel is also shown. The design of the user-interface (*Panel*) in Statemate is easily performed using

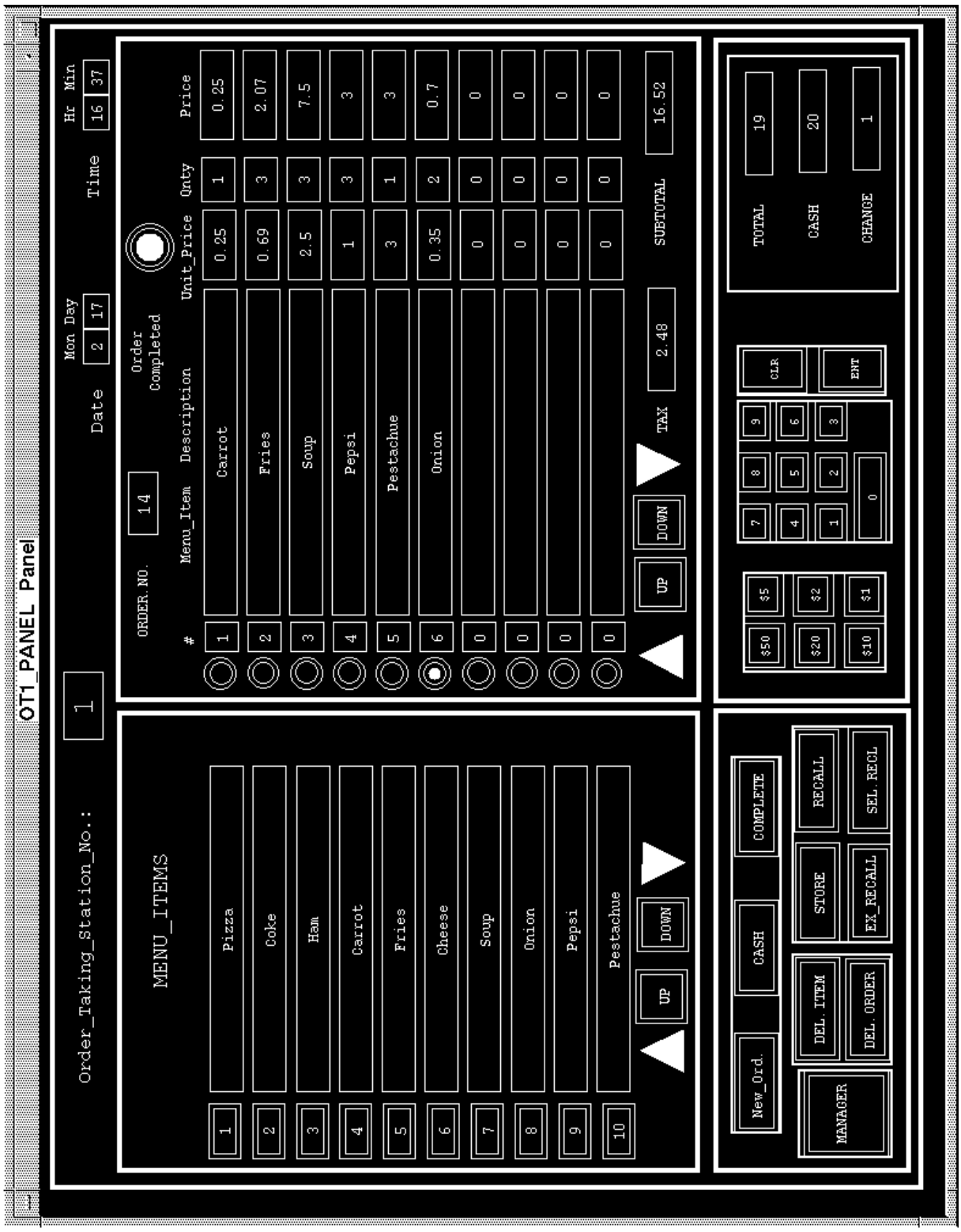

Figure 9: *Panel Graphic-Editor* of *Statemate*. Using a variety of design widgets, sophisticated user-interface panels can be easily implemented.

a variety of panel elements which are defined below:

- **Push Button**: Is mainly used for triggering an event to the system or changing the value of a condition.
- **Radio Button** (vertical and horizontal): Is mainly used for changing the values of the conditions.
- **Lamp** (Indicator with label *L*): Shows the change of a condition or occurrence of an event in the system.
- **Text Display** (square with label *DISP*): Is used for inputing (or outputting) of text to (from) the system.
- **Measuring Display** (circles or slider): Is used for indicating the values of the variables.
- **Primitive Elements**: These are primitive shapes such as circle, square, arch, line, and polynomial which can be gathered in groups to constitute composite figures. By controlling the different attributes of these groups of primitives such as: Visibility, movement, color, rotation, and scale, the resulting figure looks like an animation.

The combination of the panel visual-elements presents quite a wide range of design options for the user-interfaces. It should be mentioned that the panel editor has also some restrictions. For example we cannot overlap a display region over the push-button region of the screen.

Bindings: Each element of the panel should be bound to one of the variables (*Data-item*, *Event*, or *Condition*) in the system.
For example the push-buttons are usually used to trigger an event in the system. The event is bound to the label of an arrow in the state-chart so it causes the system to make a transition and change its state. Radio-buttons are usually bound to conditions, so pressing a button causes a condition to become "true". Conditions like the events are used as a label of an arrow in state-chart and this condition will also make a state transition. Text displays are always bound to the string data-items and are used for inputting or outputting of text. Writing a text string to the corresponding data-item will cause the text to be shown on the display. Typing in the region of the text-display causes it to be directly reflected in the data-item. The versatility and ease of use of the panel editor, greatly reduced the design effort for the user-interface of the system.

3 Designing of a Reactive System

A typical design using the Statemate tool can be started from any of the two conceptually different aspects of the system, its functionality or its behavior. If the system has massive data processing, the activity-chart is a good point to start the design, and if the system is mainly event-driven, the statechart is the better choice.

3.1 Top-Down / Bottom-Up Design

The *box-is-chart* feature of the Statemate permits developing a design in different styles, i.e. *top-down*, *bottom-up*, or mixture of both. This feature allows further decomposition of a high level design into its lower level details, as well as integration of individual design parts, developed by different designers, into a higher level chart. In the latter chart, each design part is represented as a chart-is-box. The project-manager feature of Statemate permits a controlled access to the charts, hence provides a safe and isolated environment for each member of the design group to develop his/her design. The version controller of Statemate generates a new copy of each chart of the system upon its modification.

Designing a large reactive system requires precise following of the design life-cycle principles of software engineering. A brief introduction to the different design life-cycles is presented in the *appendix* of this report. Statemate permits a spiral style of designing a system. Using the hierarchical structure of the charts, the designer is capable of creating a prototype model of the system from its specification. The simulator tool is used to simulate the functionality of the prototype to detect possible errors or any inconformity with the system requirements. Potential errors can be detected early in the design process, preventing the high cost of error detection when they are revealed at the later stages of the system development. The *requirement traceability* tool of the Statemate can also be used for this.

3.2 Concurrency in a Reactive-System

Major problems in a system consisting of a number of concurrent processes include the synchronization of the processes' operations and their inter-process communication through shared memory (IPC).

3.2.1 Synchronization

By synchronization we mean each process should wait until one or more processes reach a certain point in their execution. A large reactive system is exposed to a number of internal and external signals, which in most cases are random in nature and can occur simultaneously. To clear up this situation, a kind of distributed control mechanism is needed to react properly to the arriving signals. An example of this situation is random button pressing in the order-taker station. Each button triggers a signal (event). A normal sequence of button pressing for taking an order consists of, selecting menu items, followed by their quantities. This sequence ends with inputting of the amount of cash to the system. There are a great number of combinations of signals which are possible and only a small number of them are allowed.

How can the overall control-mechanism be implemented? Centralized-control mechanisms cannot be applied since managing the different threads of control make it a very complicated procedure. A distributed-control mechanism is appropriate. The overall control sequence

is divided into pieces that are separately assigned to each process to control itself. Managing this distributed-control mechanism in concurrent programming languages is done using a *binary semaphore*. The P and V operations are used to synchronize the activation of parallel-processes. The P operations in front of each process invocation control the activation of it and the V operations at the end of the process signal the activation of the next scheduled processes. By initialization of the semaphores, the first scheduled process(es) will be assigned. The activation sequence is easily made in this manner. This simple mechanism can also be used in statecharts. A condition variable can be assigned to each concurrent chart to control its operation. The activation of each chart (process) in this reactive system depends on two parameters. First, the condition-variable assigned to the chart must be true, and second, the signal which is monitored by this chart is triggered. All of the statecharts in the order-taker unit are controlled in this fashion. The chart *recall-order-cntl* is chosen to illustrate this method. This chart is shown in Figure 10.

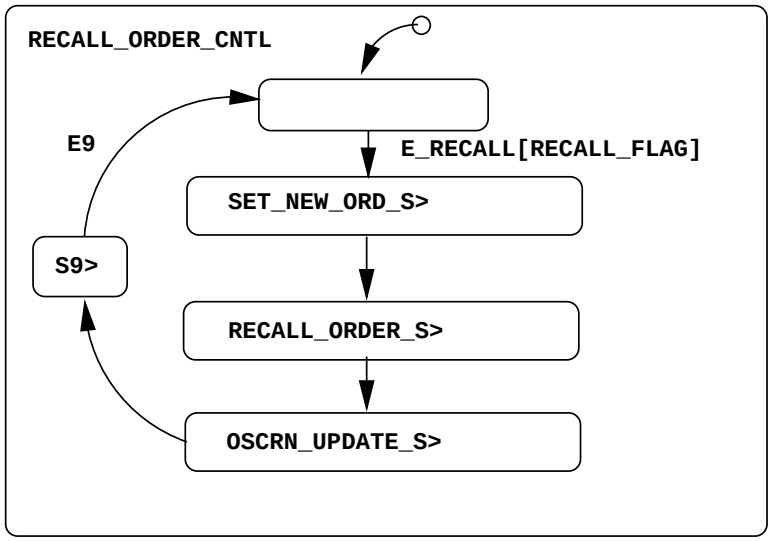

Figure 10: Statechart *recall-order-cntl*. S9 is a reactive-state and is responsible for synchronization of the chart operation with other sibling charts.

When the system is working, all of the concurrent statecharts in the order-taker unit are active and all are in their standby states (the blank state in Figure 10), waiting for external signals to be triggered. All condition variables of the charts are initialized to false other than one chart (e.g., item-selection chart). In this situation neither of the control buttons on this unit works and only item-select buttons are enabled. In fact pressing each button triggers an event, but the associated condition-variable prevents the operation of the corresponding chart. When the item-select chart finishes its operation, it is responsible for assigning the next allowed operation(s). This is performed by manipulation of the condition-variables of the other charts in a reactive-state. In the case of our example, when the condition *recall-*

flag is true, and the event *e-recall* is triggered, The chart operates and performs its duties. It then enters to state S9 and specifies which signals are allowed afterward (changes their conditions to true), and which signals are inhibited (changes their conditions to false). The contents of state S9 is shown below.

```
/RECALL-FLAG:=false;;
/SELRECALL-FLAG:=true;;
/ORDER-FLAG:=true;;
/EXITRECALL-F:=true;;
/ITMSEL-FLAG:=false;;
/E9;
```

3.2.2 Inter-Process Communication

Interprocess communication (IPC) is another important feature of concurrent systems. In concurrent textual languages there are some constructs which allow private access to shared data-structures which is essential for communication in concurrent environments. The key point in these constructs is the kernel support for *atomic* operations which manipulate some protected variables. For example, changing the value of the semaphore-variable S in P and V operations is done atomically. In Statemate, the *transitions* and *procedural-activities* are executed in one simulation step. In a typical execution of a concurrent system, many of such activities exist which are executed at the same step. As we experienced in the design of this project, Statemate atomically executes all of these activities, hence permits private access to a shared data.

In our design, there is an order-buffer that is shared between the chart *assembly-director*, which puts one ready-order in it, and *assembly-stations* which race to get the order to assemble it. This is a typical relation of the producer and consumers with a single buffer. In order to inspect the solution to this problem, a chart *producer-and-consumer* is used which consists of one producer and three consumers. This chart is shown in Figure 11.

Two condition-variables buffer-full (*buf-f*) and data-used (*d-used*) are employed to ensure that the data is loaded into the buffer, only when the consumer has removed the previous data, and the data will be used only when the producer has put new data in the buffer. State *put-data-in-buf* in the producer state P, and state *get-data* in consumer states C1 to C3 are critical-sections which should be privately accessed. The mechanism is very simple. Producer is allowed to start the communication by putting data in the buffer, the consumers can then get the data from the buffer. In the worst case when some or all of the consumers are racing for accessing the data, a simple mechanism provides mutual-exclusion. In transition from the state *wait* to the state *test-winner*, all the racing consumers simultaneously put their ID in a data-variable (*winner*). In the state *winner-test* they all test the *winner* to see whose ID is written in it. The consumer whose ID is in *winner*, trigger the event *WINx* (x can be 1, 2 , or 3) and proceeds. The other consumers trigger the event *LOSEx* and return to *wait* state. In Figure 11 consumer two has won and will get the data from the buffer. The main

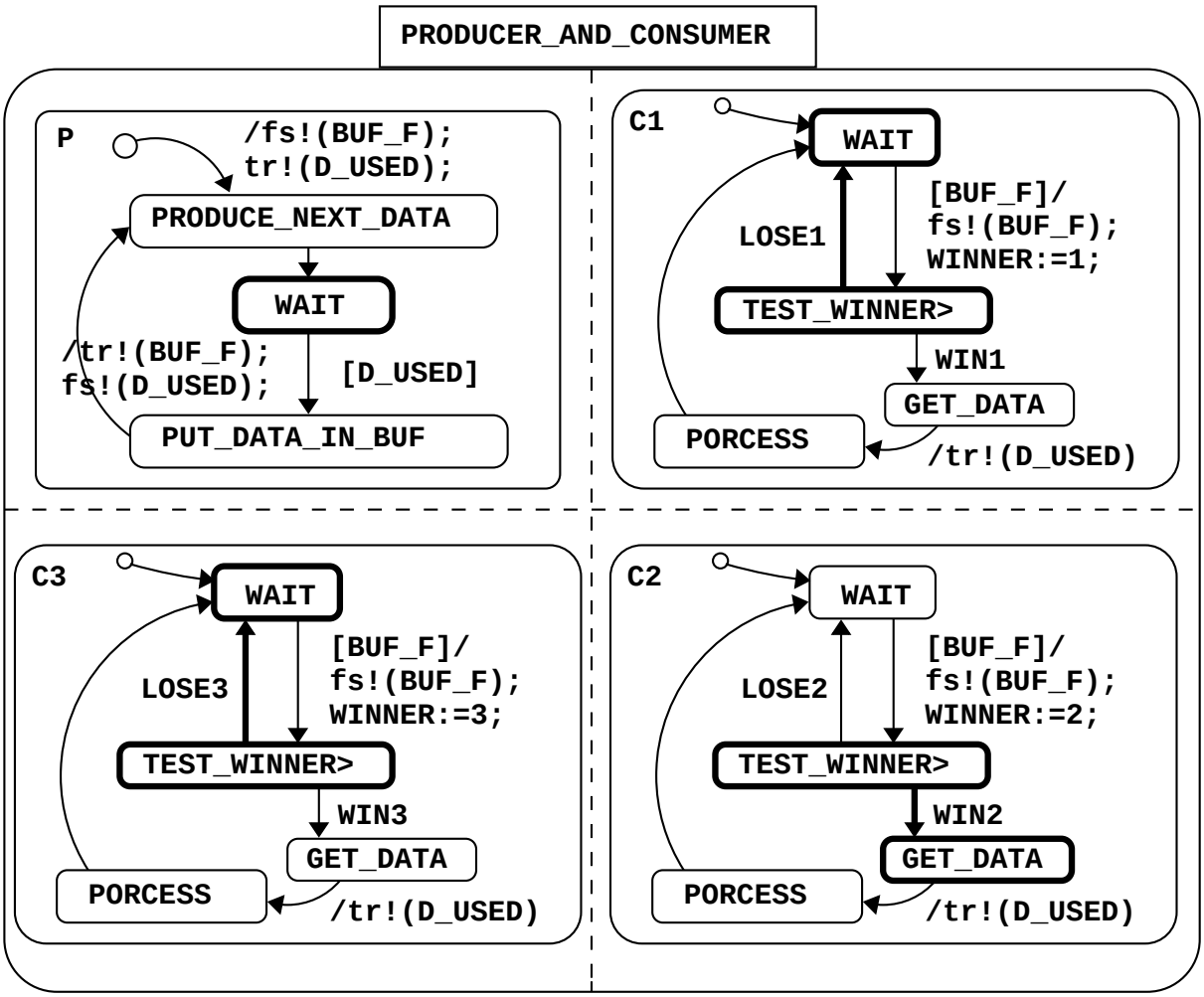

Figure 11: This chart illustrates the typical example of the producer and consumer. State *put-data-in-buf* in the producer chart, and states *get-data* in the consumer charts are critical sections.

point in this solution is the parallel and atomic execution of the actions in the transition with label:

[BUF-F]/fs!(BUF-F); WINNER:=x;

When all of the consumers are in the *wait* state and the buffer is full, in the next simulation step all can proceed and make a transition to the next state. In this transition the condition variable *buf-f* becomes false, and the data-variable *winner* is set, and all these 3 assignments are done in zero time (theoretically). Condition-variable *buf-f* controls whether the consumers enter their critical sections, and *d-used* controls whether the producer enters its critical section. These conditions are similar to the P operation in a binary semaphore. The processes, waiting in a state to enter their critical sections, are not blocked by the system hence they consume the CPU while waiting for an event. This is known as *busy waiting*.

Other forms of process communications such as bounded-buffer are also feasible with the above solution. In the restaurant system the order-taking stations provide orders and put them in a FIFO queue, and the *assembly-director* chart is responsible for removing them from the queue and assembling them. Their access to the shared buffer can be resolved using the above mechanism.

3.3 Instantiation

In the example *producer-and-consumer* presented above, there are three consumer charts C1, C2, and C3 which are the same except for the name of the events and the ID number of the charts. This method of directly copying the same chart for several times has some drawbacks. The number of the charts in the system increases. New charts and their associated variables should be defined in data-dictionary, and other time consuming redundant works. The concept of *generic-chart* and the use of its instances instead of the real charts is a solution to this problem which is offered by the Statemate tool.

Statecharts and activity-charts can be defined as a generic-chart. This concept provides the ability to instantiate different charts which encourages the designer to *reuse* the available charts. The idea is to create a *generic-chart* which performs a desired task and use it in other places without copying it. The chart is defined completely independently of the other charts of the system. When we wish to use one or more instances of that chart, we define an activity or state using a special naming pattern.

Name of Instance < Name of generic chart

As a result, the activity or state becomes an instance of the generic chart. *Formal parameters* at the generic-chart side and *Actual Parameters* at the instance side are the means of communicating the values of the variables between the generic chart and its instances. These parameters should be bound together using the data-dictionary forms of the instances. This method of instantiation available in Statemate has some disadvantages which prevents the design of a flexible software system. For example:

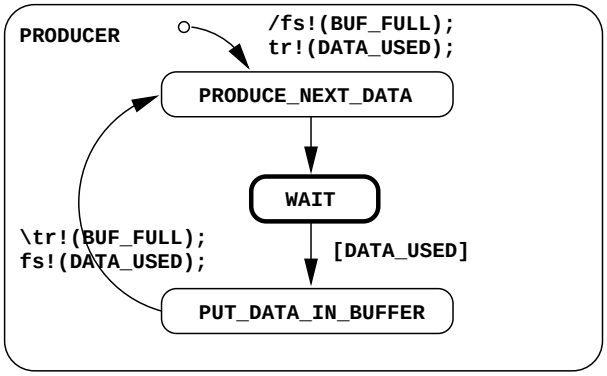

Figure 12: The producer *off-page* chart is waiting to enter its critical section.

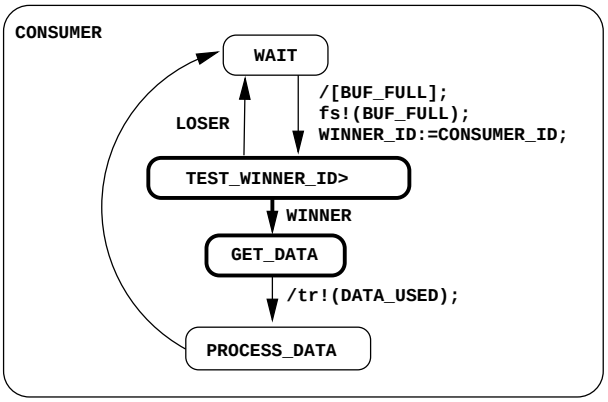

Figure 13: One of the instances of the generic-chart *consumer* which is entering its critical-section.

- The system does not support parameterized instantiation. This means each instance must be graphically defined, hence preventing a variable number of instances in the system.
- To create each instance, all of the actual-parameters should be explicitly declared in the system. There is no way of automatic creation of instances as in the fork() function in the language C.

To better illustrate the concept of generic-chart, we convert the example of *producer-and-consumer* chart into three separate charts as follow:

1. Chart *producer*, which is a regular chart (Figure 12).
2. Chart *consumer*, which is a generic chart and many instances of it can be defined in the system (Figure 13).

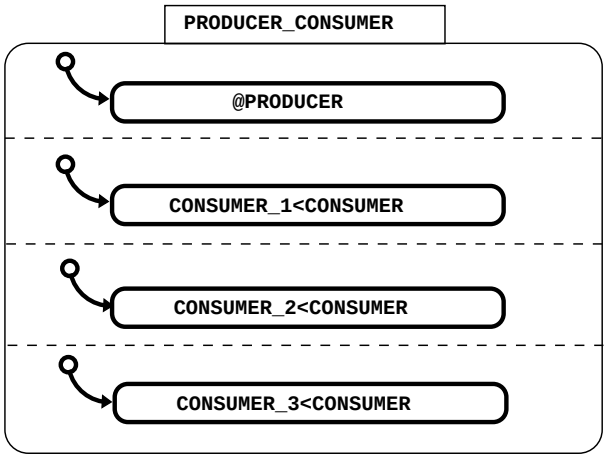

Figure 14: The parent chart *producer-consumer* provides concurrent execution of the producer chart and 3 instances of the consumer chart.

3. Chart *producer-consumer*, which is a parallel-state (AND state) and consists of the box *producer*, and three instances of the generic-chart *consumer* (Figure 14).

When the chart *producer-consumer* is activated, generates three instances of the generic-chart *consumer* which are completely similar. These instances communicate with each other and with *producer* via their actual parameters which are: *data-used, buf-full, winner-id*, and *consumerx-id*. In binding of these parameters to the formal-parameters of generic-charts, the name of the parameters are the same in both sides. For example *buf-full* (actual-parameter) in each instance-chart is bound to the *buf-full* (formal-parameter) of the generic-chart. The only difference which causes the instances to be distinguished form each other is the parameter *consumerx-id*. For example: *consumer1-id* is the actual parameter for instance one. This parameter is defined as a constant in the data-dictionary and has the value 1. It is bound to the *consumer-id* of the generic-chart consumer. The same method is used for the other two instances. Therefore when three instances are generated, the variable *consumer-id* is 1 for instance number one, and is 2 for instance number two, and so on. In this way the instances of a generic-chart can be separated and assigned to different tasks. This method of separating different instances of a generic-chart is similar to the way that the child-processes are distinguished in the *fork()* function call of the language C. Each child-process returns its process-ID to the parent-process, providing a means for the parent to assign different jobs to the child-processes. In our restaurant system we extensively used the generic-chart concept in allocating several stations to order-taking, assembling, and preparation units of the restaurant.

4 Conclusion

Statemate provides an environment for developing large reactive systems. It consists of different design tools such as: graphical editors for designing functional and behavioral characteristics of the system, simulation, code generation, a project manager, etc. This report introduced the semantics of the statecharts and activity-charts with examples for each. The step by step design procedures of a large reactive system (a *fast-food restaurant*) was described. *Top-down* and *bottom-up* design styles using the concepts of decluttering and off-page charts were explained. Functional decomposition of the system and defining the basic functions in *mini-specs* of the activities is the basis for designing activity-charts. In most designs activity-charts are the first step. Data structures and input/output design are the next steps. Designing of the data-structures is crucial since Statemate 5.0 has many shortcomings in this regard. User-interface design in Statemate is easily performed using the dedicated *panel-editor*. Various bindings of variables to the charts or to the panel elements are done in the data dictionary. Defining the external signals to the system and separate control sequences for small parts of the system function is the key to the design of the statecharts. Lower level charts of the system can be easily integrated into parent charts to provide a higher level of abstraction in system design. Techniques for synchronization and communication among concurrent charts were described and the instantiation of the generic-charts along with their advantages and disadvantages were discussed. At the end, using a number of criteria, the Statemate tool is evaluated and its merit as an ideal system development environment is indicated.