

OARS: An Object-Oriented Architecture for Reactive Systems¹

Bernard T. Barcio, S. Ramaswamy, K. S. Barber

The Laboratory for Intelligent Processes and Systems, Department of Electrical and Computer Engineering
The University of Texas at Austin, Austin, Texas 78712-1084 USA

Abstract¹ - This paper discusses an architecture designed to provide support for the development of state transition models for an object-oriented distributed environment. The state transition models can be constructed and specified hierarchically as well as derived into new classes of objects through the use of inheritance. A new concept called "exit-safe states" is introduced to assist in the specification of hierarchical state transition models. A graphical monitor to analyze the system at run time has been developed.

I. INTRODUCTION

The object-oriented approach to analysis, specification, and design of large software systems provides an intuitive mechanism for problem-domain specification. Object-oriented methodologies promote² software reuse and maintenance. There have been many object-oriented models and methodologies to describe systems [1], [6] and [13]. When describing systems, formal methods are also used to provide mathematically concise descriptions. Moreover, it is understood that visual specification techniques should be used to ease understanding of the system specification and still provide formal conciseness in real-world applications. More on these techniques is discussed in [3] and references therein.

Object-oriented analysis can be broken down into two parts: developing the information and the state transition models. The information model is fairly straight forward and represents the information contained within an object. The behavioral model specifies the time-dependent behavior of the object. An event-driven approach is followed in this paper. Events are assumed to cause a change in object behavior as they trigger the transitions between states. An object with a state life-cycle definition is often called an Active Object [12] (as opposed to a passive object which has no state life-cycle definition). Specifying high-level lifecycles and allowing different implementations of substates will promote the reuse of high-level descriptions of systems operations. The state life-cycle model of [12] and the stimulus-response diagrams of [4] do not provide a mechanism for specifying the state lifecycles in a hierarchy. Rather, they specify state-lifecycle specification as a single level of states. This complicates top-down specification and lifecycle reuse across objects. Harel's state charts [8] extend the state-transition model to include hierarchies and other methodologies have been developed based on Harel's model [9] [11]. When specifying state transition dia-

grams for active objects it is convenient to think in terms of the states themselves, the events which are legal in those states, and which states those events lead to when processed.

Language support for the information model is prevalent in object-oriented languages like C++ and Smalltalk, however, state transition model support is underdeveloped. The objective of this paper is to propose and demonstrate an architecture that provides a convenient mechanism for specifying and implementing the information and the state transition models of object-oriented distributed systems. The following are some of the important characteristics of the architecture: (i) Supports hierarchical transition models for objects in a distributed environment, (ii) Provides object-oriented features like inheritance, encapsulation, and polymorphism within the state transition model, (iii) Defines an extension to hierarchical state specification called exit-safe states, (iv) Provides a visual technique for system specification and analysis.

The paper is organized as follows: Section II presents an overview of the architecture being developed. Section III describes some useful features of the architecture. Section IV discusses the current implementation. Section V presents an example. Section VI concludes the paper.

II. OVERVIEW

In a top-down specification, the developer need not be attentive to lower level system behavior. Consequently, when specifying state transitions based on events, an event is allowable in a state if it can cause a transition from the current state to another at the same level in the hierarchy. However, in order to promote reuse, a mechanism for encapsulating substate information has to be defined. The substates underlying a state are concealed from the external view. Therefore, the implementation of the substate configuration can change without affecting the parent state. Since only the transitions into and out of the state need to be known, the implementation is made private.

Another important requirement of the architecture is that it must be able to run on heterogeneous platforms within a distributed environment such that it allows locally optimal agent construction, modularity, and parallelization of concurrent processes. Therefore, it supports both point-to-point and multi-cast message passing. However, the traditional object-oriented message passing mechanism is point-to-point and is useful when requesting(providing) a specific service from(to) an object. Since the ordering of events becomes crucial in certain situations, the system ensures some basic characteristics of message communication.

In order to ease the transition from analysis and design to implementation, the application programming interface (API) is simple and logically extended from the analysis and design methodology. That is, a one-to-one correspondence exists

1. This research was supported in part by the National Science Foundation under grant IRI-9409192, and in part by the Texas Higher Education Coordinating Board under Grant ATP-115.

2. Object-oriented techniques are often attributed to increased software reuse because of their attributes, namely, inheritance, encapsulation, and polymorphism. Object-oriented techniques have to be considered as a convenient platform to generate reusable software and not a platform that guarantees wide software reuse.

whenever possible, between what is established when designing the system and its corresponding implementation. In the case of establishing state hierarchies, it is easy to add a state to an active object; to attach an action with the state; and, to establish substates to those states. Moreover, in order to ease system development, especially when dealing with distributed systems, a graphical monitor is used in viewing the state transitions.

A. Exit-safe States

Exit-safe states represent states wherein the substate lifecycle is in a stable state to process the transition out of the parent state. The following example illustrates the concept of exit-safe states. Let s be a state with an outgoing transition t occurring on an event e . Let $\{a, b, c\}$ be the set of substates under s . When specifying the substate lifecycle, s , the developer does not want an outgoing transition on s to occur until substate c is reached. Therefore, c , is designated as an exit-safe state. Another implementation might define a substate lifecycle defined by the substates in $\{a, b, c, d\}$. The designer might designate c and d as “exit-safe”, so that t can occur if the substate lifecycle is in either state c or d .



Figure 1. Icons for Exit-Safe and NonExit-Safe States

Exit-safe and non-exit-safe states are illustrated in Figure 1. Figure 2 illustrates some examples of exit-safe states, where the shaded states conceal their underlying substate diagrams and the non-shaded states show their substate lifecycle.

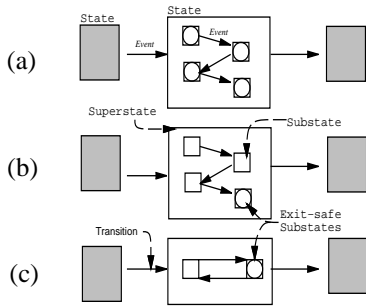


Figure 2. Examples of Exit-safe State Use

Three classes of substate configurations are distinguished: (i) *XOR class*. In this class, all the substates of a parent state are exit-safe states. This yields the same effect as Harel’s XOR-states in [8]. When a transition legal for the parent state occurs for an *XOR class*, it would immediately transition out into the next state (Figure 2a), regardless of the current substate of s , (ii) *Single-Exit class*. In this class, a stream of substates terminate in an exit-safe state, such that it guarantees that all of the substates are entered before exiting the parent state (Figure 2b), and, (iii) *Multi-Exit class*. In this class, a cycle exists among with one or more substates which comprise the parent state and designate one or more of these substates as an exit-safe state(s). Thus, substate lifecycles are established and it is ensured that the parent state is in a stable substate before transitioning between parent states.

For example, if a machine is to inspect a part at a certain point in its state lifecycle it would enter an *Inspection* state. The manipulator rotates the part while a camera inspects the part. It might take several passes to ensure that the part meets its specifications but it could exit sooner if a defect is found. In this case, the substate transitions in a cycle between two states: (i) the state in which the manipulator is rotating the part, and, (ii) state in which the part is stopped in the appropriate position to continue. The manipulator would cycle between these two states until either a defect is found or the part is determined to be defect-free (Figure 2c).

B. Formal Notation of the Exit-safe States

As stated in [7], states are defined along with a hierarchy function, ρ . A terminal type function ω is added to express the exit-safe states.

The *hierarchy function* designates the substates of a state s . Where S is the set of all states in the system, ρ is defined by the following: $\rho : S \rightarrow 2^S$
 $\rho(s)$ defines the immediate substates for a given state s . Two different states cannot have the same set of substates. Therefore, if $\rho(x) = \rho(y)$ then $x = y$. The root state of the description is the state which is not a substate of any other state. r is unique and defined by the following:

$$(r \in S) : \forall (s \in S) (r \notin \rho(s))$$

Two extensions of ρ are defined. (In the following, ρ^i denotes the substates i levels below s ($\rho^0(s)$ denotes s itself, $\rho^1(s)$ denotes the immediate substates of s , etc.)):

- $\rho^*(s)$: designates s , all the substates of s , the substates of those substates and so on recursively. Thus:

$$\rho^*(s) = \bigcup_{i \geq 0} \rho^i$$

- $\rho^+(s)$: designates all the substates of s , the substates of those substates and so on recursively. It does not include s .

$$\text{That is: } \rho^+(s) = \bigcup_{i \geq 1} \rho^i$$

The *exit-safe* function determines exit-safe or nonexit-safe state. That is: $\omega : S \rightarrow \{EXIT-SAFE, NONEXIT-SAFE\}$
This function is used when defining Ω which determines whether a transition leading out of a state can occur:

$$\Omega : S \rightarrow \{True, False\}$$

$$\Omega(x) : in(x) \wedge \forall s \in \rho^+(x) : (in(s) \rightarrow \omega(s))$$

where $in(x)$ denotes that the system is in state x . $\Omega(x)$ is a function which returns true if the object is in a state, x , and all the substates which x is in are exit-safe. If a valid event occurs in state, x , the corresponding transition out of x can be processed if $\Omega(s)$ evaluates to *True*.

The *pending events function* returns the events pending to a state. That is: $\pi : S \rightarrow 2^{E_{legal}}$, where E_{legal} is the set of all legal events in a given state, s . $\pi(s)$ returns a set of events which are to be processed when it is possible to exit a state s .

The set of transitions, T , defines a set of tuples. Let E be the set of all events in the system, G be the set of all guards in the system, and, Δ be a tuple defined by $\Delta \subset E \times G$, then the set of transitions, T , is defined by: $T \subset S \times \Delta \times S$

A transition $t = (a, l, b)$ where a is a source state, l is a label, and b is a target state. The label, l , is defined by the set $\{e, g\}$ where e is an event and g is a guard. Transition, t , will occur iff (i) the system is in a , and (ii) event e occurs, and (iii) guard g is True. Then, the system will be in state b if t occurs.

Before proceeding further, it is convenient to define the following functions:

$$X: T \rightarrow S, L: T \rightarrow \Delta, L_{G(g)}: L \rightarrow \{True, False\},$$

$$L_E: L \rightarrow E, \text{ and } Y: T \rightarrow S$$

where: $X(t)$ returns the source state t , $L(t)$ returns the label associated with t , $L_{G(g)}(t)$ returns the evaluation of the guard associated with t , $L_E(t)$ returns the event associated with t , and $Y(t)$ returns the target state t .

C. Special Events

Events also exhibit a class hierarchy. Therefore, it is possible to derive classes of events from base class events (i.e. the derived events will enable the same transitions as the base class enabled). For this discussion, it is useful to define two types of events derived from the general event class. All other types of events in the system will be derived from these classes. These two event types are denoted E_B and E_χ .

There are certain events (such as an error event) for which it is desirable to exit a state even if it is not in a terminal state. In such cases, it is necessary to define special event types that will exit a state regardless of its current substate. These types of event will allow the system to immediately handle error events or to process special interrupts which require immediate attention. This special type of event will belong to a special set of events, E_χ . All other events are called basic events, E_B .

The set of all events in the system, E , is thus defined as the union of all the special exit events, E_χ , and the basic events, E_B . That is:

$$E = E_\chi \cup E_B, E_\chi \cap E_B = \emptyset$$

For an application, it is necessary to define how pending events will affect the system. In OARS, receiving many events in a state in which its current substate is not exit-safe is handled in many ways. If the incoming event, e_i , is a system error type, then the event is processed immediately. That is:

$$e_i \in E_\chi \Rightarrow in(s) \rightarrow in(Y(t)) \text{ where } L_E(t) = e_i.$$

The handling of events that are not error events is discussed in more detail in [3].

III. ARCHITECTURE FEATURES

A. Class Inheritance

Class inheritance of the active objects developed in the system are derived from the features of C++. The user develops classes to contain the information model as well as the methods for each active object in the system. Since some of the methods within the active object represent actions for cer-

tain states, action inheritance is the same thing as method inheritance. Therefore, when deriving a new class from an active object, the states and the actions performed in the states are inherited. The benefit of this feature is that it allows the user to apply polymorphism to actions within a state. Thus, it permits a derived class to write a function with the same name and have it used instead of the base-class function wherever the base-class function would have been used. Thus, the derived class can have the same states and transitions as in the base class but different actions performed in certain states.

For example, if a derived robot class performed forward kinematics calculations in a different way than its base class, then the developer could declare the base class forward kinematic calculation action as virtual and then rewrite it within the derived class. Therefore, when the event occurred to transition a robot object into the forward kinematic calculation state, the derived robot class action is executed.

B. State Lifecycle Inheritance

The specification of the states and transitions between those states are specified in the constructors for the objects. Inheritance of a class (referred to as the “base class”) will involve calling the constructor and thus instantiating that class’ states. Polymorphism can be used to overwrite the action to be performed in the state. After the base class constructor is called and the base states instantiated, the derived class is then free to define new transitions to and from the base class’ states.

Some restrictions must be placed on inheritance of state diagrams in order to assure logical consistency. Coleman, *et al.* [5] suggest the following restrictions for subtyping in their extension of Statecharts, “Objectcharts”: (i) adding an extra state or transition corresponding to a new service, (ii) strengthening a transition specification by weakening its firing condition or strengthening its post condition, or (iii) strengthening an invariant relationship. These restrictions ensure that the newly derived state does not change how the base state reacts to stimuli. That is, the derived state will fire at least as often as the base and it will result in the original post condition. The derived class can also add substates to its own as well as the base class states. An example of this use of inheritance is provided in Section V.

C. Guards on Transitions

Guards are boolean expressions which can be added to a transition to a certain state. The transition will occur only if the guard evaluates to true. This can be useful when the developer wants to specify a transition on more than the occurrence of a single event. A guard can optionally be supplied by the developer when specifying a transition.

D. Exit-safe State Handling

As explained in Section II, the exit-safe state concept is implemented in this support architecture. With the concept of the exit-safe state comes the need to pend events. When an event occurs which is legal in the current state but the state is in a substate which is not exit-safe, the system pends the event

and will process it as soon as it enters a substate which is exit-safe. By allowing the pending of events, the system maintains consistency with the specification. The specification implies that the event is part of a legal transition for the current state and thus, should be processed for that state. However, the system is not at a point where it has completed its operations in that state (that is, since the substate the system is not exit-safe, the substate cycle is not finished or not at a stable point). Therefore, the system waits until the state is completed (designated by entering an exit-safe substate) and then continues by processing the pending events. Therefore, a transition is enabled when: (i) The event(s) associated with the transition occurs, and (ii) the guard evaluates to True, and, (iii) all current substates are exit-safe (i.e. $\Omega(s) = True$). In order to handle transitions with exit-safe states properly the following algorithm is used. This is illustrated as a flow chart in Figure 3.

For an event, e ,
 Foreach

$$t \in T : in(X(t)) \wedge ((L_E(t) = e) \wedge L_G(g)(t))$$

Then If $\Omega(X(t)) \vee ((e \in E_\chi))$

Then $in(Y(t))$ Else $pend(e, X(t))$

EndIf

EndForeach

The order of execution of the potential transitions is non-deterministic. A transition will occur if:

$$\exists (t \in T) : in(X(t)) \wedge (e = L_E(t))$$

It is not guaranteed which transition will take place, unless the specified system itself has a mechanism for guaranteeing the order of transition execution. After entering an exit-safe state and performing the action associated with that state, the pending events need to be processed. Handling pending events is application dependent. Some possible ways of handling the pending events are described in Section IID. Section V provides a simple example of a system containing exit-safe states.

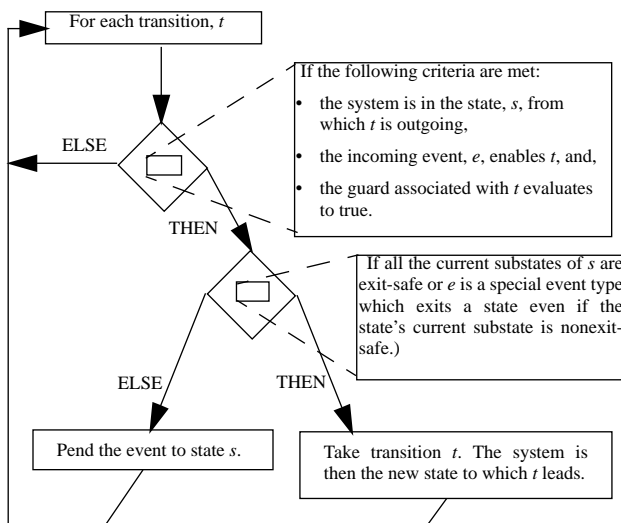


Figure 3. Flowchart of Transition Algorithm

IV. IMPLEMENTATION

The architecture has been implemented in C++. C++ provides the necessary inheritance features of the information model as well as virtual functions for changing base class methods in derived classes.

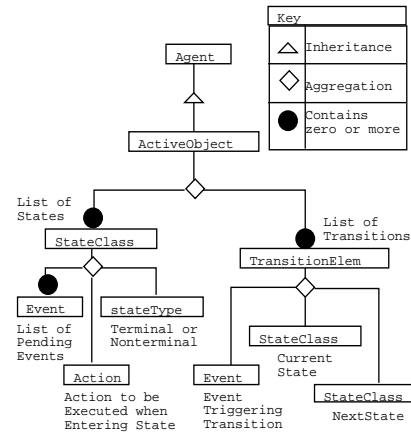


Figure 4. Basic Object Notation for Architecture

The *InterAgent* toolkit for distributed system development has been chosen to provide the backbone for communication management within our system [10]. It supports many platforms including Sun, Silicon Graphics, and PC providing an open system development environment across those systems. *InterAgent* also supports point-to-point and multi-cast message passing.

The architecture is structured as several classes: *Agent*, *ActiveObject*, *TransitionElement*, and *StateClass*. A simplified model of the construction of these classes is shown in Figure 4 using Rumbaugh's Object Model Notation [11].

Figure 4 illustrates that the *Agent* class is the most basic class and is common to all the rest of the classes in the architecture. The *Agent* class contains the information and methods to handle the message communications between distributed processes.

The *ActiveObject* is the next most basic class. It is comprised of several states and several transitions which combine to make up the state lifecycle of the object. The *ActiveObject* class contains the information and methods to handle the processing of events received by the agent. Processing the events involves: (i) determining whether the event is legal for the current state of the *ActiveObject*'s, (ii) either pending the event or evaluating the transition to the next state, and (iii) evaluating any guards placed on a transition.

The state is represented in the *StateClass*. The *StateClass* defines the state of the *ActiveObject* as well as the methods of the *ActiveObject* to be invoked when entering that state.

The transitions are represented by the *TransitionElem* class. The *TransitionElem* class defines how the *ActiveObject* responds to events. When constructed, the active object specifies the event, the state in which the event is legal, the destination state, and optionally, a guard.

All user-defined *ActiveObjects* should be derived within the *ActiveObject* class. The developer should keep in mind where abstractions can be made to promote reuse throughout the systems development life. A top-down specification

defines the basic state flow for an active object in a class. Derived classes from that basic state define details of the sub-states for state specified. If the substate definition changes but the basic high-level state definitions remain intact, the base class can be reused. An example of this is provided in Section V.

V. ASSEMBLY ROBOT EXAMPLE

In order to illustrate the capabilities of the system, this section discusses an example program development. This example deals with a simple assembly robot. The example develops the class hierarchies of the information model and the state life-cycles and shows some output of the monitor program.

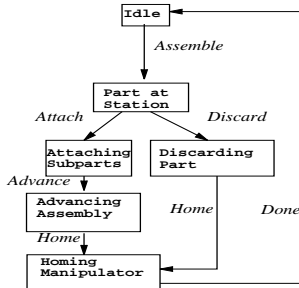


Figure 5. Basic State Model of Assembly Operation

A. Assembly Robot Base Class

The assembly robot example consists of an assembly robot which goes through six basic states: 1) Idle, 2) Part at Station, 3) Assembling part, 4) Advancing part, 5) Discarding part and 6) Homing manipulator. These states and the corresponding event transitions are shown in Figure 5.

B. Derived Assembly Robot Classes

Two implementation scenarios are discussed. The first is when inspection of the part is added to be handled within the manipulator agent and the second is one in which inspection is handled by a separate agent. Each scenario involves the six basic steps outlined above, so a base class will be developed to handle the six basic states and their transitions. Two derived classes will inherit the information and state characteristics of the base class but will add different sub-state actions to handle the different scenarios.

The case in which inspection is handled directly by the assembly robot agent involves adding sub-states to the *Part at Station* state. During this state the robot will inspect and then grab the part and attach it to the subassembly. The sub-states are shown in Figure 6.

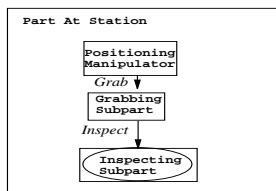


Figure 6. Substates of Part At Station State for Scenario 1

After inspection, either event *Attach* or event *Discard* is generated and the robot moves into the appropriate state as in

Figure 5. After reaching this state, the assembly robot can then transit out of the parent state, *Part At Station*, and continue to the appropriate state. The other scenario involves a separate agent which does the inspection. Again, the basic states of the assembly robot are the same, and only the sub-states of *Part At Station* are different. The base states are thus inherited. The derived assembly robot class will then have substates of *Part At Station* which could look like those in Figure 7.

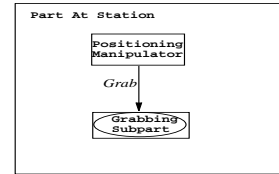


Figure 7. Substates of Part At Station State for Scenario 2

The separate inspection agent then sends either the *Attach* or the *Discard* events while the assembly robot is in the *Part At Station* state. That event pends until the assembly robot finishes the *Grabbing Subpart* state and is exit-safe. Figure 8 - Figure 12 provide snapshots of the implementation.

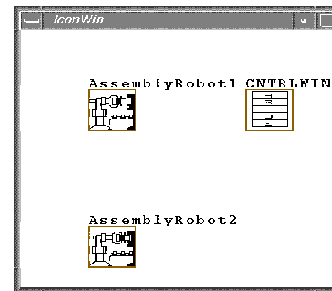


Figure 8. Agent Icon Popup Window

Figure 8 displays the different agents within the system. When an agent process is started, it announces this fact to the monitor which then places an icon in the window. In this example, there are two assembly robots, *AssemblyRobot1* and *AssemblyRobot2*, and a control window, *CNTRLWIN*. The two assembly robots are implementations of those discussed in the previous example, and the control window is used to manually send events to the processes.

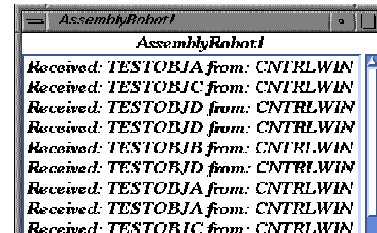


Figure 9. Event History Popup Window for Assembly Robot 1

Figure 9 illustrates that when the user clicks the middle mouse button on an icon, a popup window appears displaying all the messages sent from and received by that agent. In this case, *AssemblyRobot1* has received a series of events from

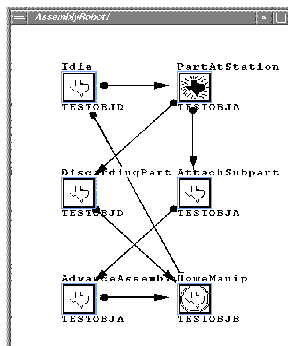


Figure 10. Top Level State Transition Diagram Popup Window

Figure 10 shows the top level state transition diagram window which pops up when the user clicks the left mouse button on an icon. In this case, it shows the top level state transition diagram of *AssemblyRobot1*. All the states are represented by the Texas state icons. The current state of the agent is shown by a dark Texas state.

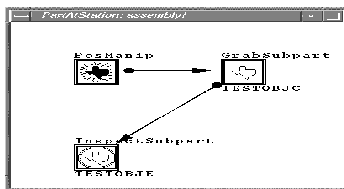


Figure 11. SubState Transition Diagram Popup Window

Figure 11 shows part of the substate popup window for *AssemblyRobot1*. This window pops up when the user clicks the left mouse button on a state. Here, since the first substate is the current substate of the system (notice that the current state is **PartAtStation** substate **PosManip**). The next state in the system, **GrabSubpart** is a exit-safe state and is denoted by the circle around inscribed in the state square.



Figure 12. PartAtStation Transition List Popup Window

Figure 12 shows a window which pops up when the user clicks the middle mouse button on a state. This window defines the legal transitions from and to that state. Here the **PartAtStation** state is illustrated.

VI. CONCLUSIONS AND FUTURE WORK

The current implementation of the support architecture described provides a useful transition mechanism from analyzing to developing object-oriented distributed systems. It supports state hierarchy specification and implementation, inheritance of state life-cycles within active objects, guarded transitions on events. The notion of exit-safe states provides a useful extension to hierarchical state transition diagrams.

A graphical tool and an automatic code generator will be developed to specify the system and to generate the appropriate C++ code as required by the architecture. With the addition of this tool a complete design, implementation, and monitoring system will have been developed.

REFERENCES

- [1] Grady Booch, *Object Oriented Design with Applications*, Redwood City, California: The Benjamin/Cummings Publishing Company, Inc., 1991.
- [2] Bernard T. Barcio, *OARS: An Object-Oriented Architecture for Reactive Systems*, Masters Thesis, University of Texas at Austin, Fall 1994.
- [3] B. T. Barcio, S. Ramaswamy, K. S. Barber, "An Object Oriented Modeling and Simulation Environment of Reactive Systems Development", *Submitted to the IEEE Transactions on Software Engineering*, Feb. 1995.
- [4] G. W. Cherry, "Stimulus-Response machines: A New Visual Formalism for Describing Classes and Objects," *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 2, Apr. 1993, pp. 86 - 95.
- [5] D. Coleman, F. Hayes, S. Bear "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design," *IEEE Transactions on Software Engineering*, vol. 18, no. 1, Jan. 1992, pp. 9 - 18.
- [6] S.Gossain, B. Anderson, "An Iterative-Design Model for Reusable Object-Oriented Software," *ECOOP/OOPSLA 1990 Proceedings*, Oct. 1990, pp. 13 - 27.
- [7] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman, "On the Formal Semantics of Statecharts," *Proc. 2nd IEEE Symp. on Logic of Computer Sci.*, 1987, pp. 54 - 64.
- [8] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Sci. Comput. Program.*, vol. 8, no. 3, June 1987, pp. 231 - 274.
- [9] H.M. Jarvinen, et al., "Object-Oriented Specification of Reactive Systems," *Proceedings of the 12th International Conference on Software Engineering*, 1990, pp. 63-71.
- [10] Modulus Technologies, *Interagent Toolkit User's Manual*. Modulus Technologies, Inc. 1993.
- [11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.
- [12] Sally Shlaer, Stephen Mellor, *Object Lifecycles: Modeling the World in States*. Englewood Cliffs, New Jersey: Yourdon Press, 1992.
- [13] Robert C. Sharble, Samuel S. Cohen, "The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods," *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 2, Apr. 1993, pp. 60 - 73.