# An Object-Oriented Multimodel Design for Integrating Simulation and Planning Tasks

Jin Joo Lee
jl1@cis.ufl.edu

William D. Norris II
wdn@cis.ufl.edu

Paul A. Fishwick
fishwick@cis.ufl.edu

Department of Computer and Information Sciences
University of Florida

June 3, 1993

**Abstract**

Object oriented simulation methods have promoted a structured way of building models based on the mapping between physical objects and the logical objects. Dynamics within the traditional object oriented framework is presented in the form of finite state machines embedded within data flow functions. That is, as data and control flow within a data flow graph, the dynamics are represented by internal state changes. While this paradigm is powerful, we present an extension to the object oriented approach to simulation and planning by allowing model components to be defined in terms of different model types. This extension supports the idea that different parts of a model are defined differently depending on the level of abstraction. Our extension to the object oriented paradigm is described with a truck depot example that demonstrates how reactive control and deliberative planning are integrated within a *multimodel* framework. We found that the *multimodel* method is powerful for system simulation and planning when employed within an object-oriented infrastructure. [**Key Words: Autonomy, Intelligent Control, Multimodeling, Planning, Object Oriented Simulation**]

# 1 Introduction

The object oriented approach to simulation is discussed in different literature camps. Within computer simulation, the system entity structure (SES) [Zei90] (an extension of DEVS [PAG93, Pra91]) defines a way of organizing models within an inheritance hierarchy. In SES, models are refined into individual blocks that contain external and internal transition functions. Within the object oriented design literature [RBP+91, Boo91], the effort is very similar in that object oriented simulation is accomplished by building 1) a class model and 2) dynamic models for each object containing state information. Harel [Har88, Har92] defines useful visual modeling methods in the form of "state charts" so that the dynamics may be seen in the form of finite state machines. From our perspective the object oriented approach provides an excellent starting point when deciding how to organize information about dynamical systems:

1. Start with a concept model of the system.
2. Create a class model using a visual approach such as OMT [RBP+91]. This phase should involve creating all relationships among classes.
3. Specify the dynamics for each class instance where state transition is a factor. Note that some classes will not contain state information and some relations may not be of a dynamic nature.
4. Construct a *multimodel* to build a network of models each of which defines a part of the overall system.

In the usual object oriented approach, phase three translates to creating methods for an object that alter the state of that object. The problem is that phase three can be quite complex depending on the scale of the system being modeled. There needs to be a way of developing multi-level models that specify the phase three dynamics. Our approach is to use *multimodels* [FZ92, Fis91, Fis92, Fis93b, Mil93] for this purpose.

Multimodeling is a paradigm for designing and executing models. We use several well defined model types and connect them, so that the lower levels refine the higher levels. Due to the hierarchical structure of the multimodel approach, the object oriented paradigm is natural for implementation. Each of the model types are executed using the same methods, *Initialize()*, *Input()*, *Output()*, *State()* and *Update()*. Therefore, by

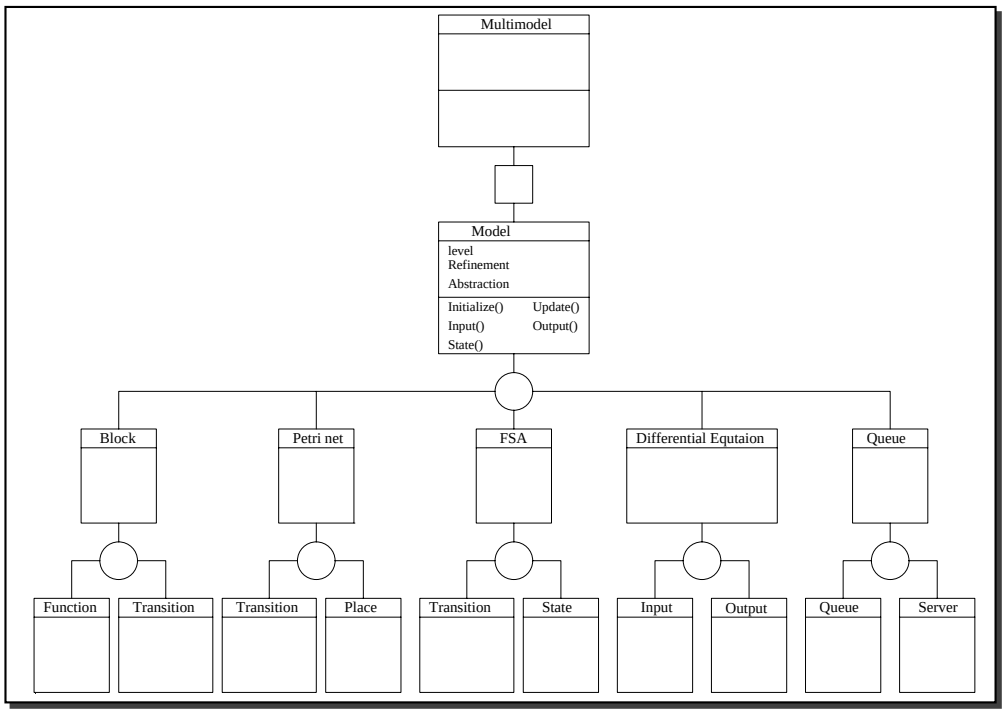


Figure 1: Object model of the multimodel objects

having a method of communication between each model type, the models are executed regardless of which model types are used. A class model of the modeling types is shown in Fig. 1.

The paper is organized as follows. In section 2, we discuss the issues involved in integrating simulation and planning along with some of the existing problems in each domain. An example involving the operation of a truck depot is described in section 3. Using the example, we show how simulation and planning are combined using multimodeling to model intelligent objects in section 4. Section 5 defines how a multimodel is designed and executed. In section 6, we describe how the multimodel of the non-intelligent objects from the truck depot example is implemented and executed.

# 2 Integrating Simulation and Planning

To illustrate the use of object-oriented multimodel design, we decided to integrate the normally separate tasks of simulation and planning. The idea of integrated simulation and planning developed from our efforts to overcome the many problems inherent in artificial intelligence (AI) and simulation. Traditionally in AI planning, researchers are concerned in building a planner that produces a symbolic plan (order of primitive actions) which will achieve certain tasks. However, the planner is not responsible for the actual execution of these plans. The planner is unable to control the execution or modify the plan in order to guarantee success of the proposed plan. Traditional AI planners were therefore unrealistic. First, the planners did not account for uncertainties or dynamic changes in the environment. Second, the planners had no notion of planning in real time. Basically, these *deliberative* planners assumed perfect knowledge of a very static world. Taking the other extreme, other researchers developed purely *reactive* planners (sometimes called systems due to the purely reactive nature of the planner) that can react to the environment by executing actions without extensive reasoning. Such systems are sometimes discussed under the umbrella of "automatic control" due to lack of intelligence.

A planner must have the ability to perform some form of long-range planning while reacting accordingly to any changes of the environment that may need immediate action. The next logical step is to combine planning and control to produce a planner that is both deliberative and reactive. Dean [TW91] provides a good overview of the various problems and techniques available in these two areas. Most of the traditional planners

which have been built so far are either purely deliberative or purely reactive. Recently, there have been some efforts to develop a *combined* planner [Kae87, Bro86, FG90]. Due to the divided research between deliberative and reactive planners, the technology of the two fields has also been divided. We believe the major difficulty in trying to build a *combined* planner is integrating the different methods of each area. For instance, a typical deliberative planner may use some type of first-order predicate logic to model knowledge and reasoning, whereas a reactive planner or controller may use methods from control theory such as differential equations. Thus, multimodeling allows the integration of these different techniques as submodels.

With planning and control combined, we now want to integrate the different modeling types that exist in AI and simulation. There has been previous work done in the integration of AI and Simulation [Nie91, O'K89]. However, combining different modeling paradigms and techniques is often a difficult task. Because of the ability to combine different modeling paradigms at multiple levels, we consider the object-oriented approach of multimodeling [Fis91, Fis92] to be a natural approach to solving the problem.

There are advantages to be obtained by integrating simulation and planning. Because both the intelligent and non-intelligent objects are being modeled and simulated under one simulation, we are able to test and evaluate the performance of the overall system. From the planning perspective, testing, evaluation and modification can be done without connecting the planner to an actual physical device or object.

The Truck Depot problem was originally taken from [TW91]. Since the problem contains both non-intelligent objects (e.g. basin, trucks, valves) and intelligent objects (e.g. robots or people) in equal emphasis, the problem inspired us to find a solution to optimization by combining the two fields of simulation (simulating non-intelligent objects) and AI (simulating intelligent objects) under a unifying modeling paradigm.

# 3 A Truck Depot Example

Fig. 2 shows the aerial view of the truck depot, which represents the concept model of system. The depot contains one basin with two input pipes $P_1$,$P_2$ and one output pipe $P_3$. The two input pipes $P_1$, $P_2$ carry two different chemicals. In the basin, a mixture is made from these two chemicals. Empty tanker trucks arrive at the depot and wait until they can move under valve $V_3$ to be filled with the mixture from the basin. When

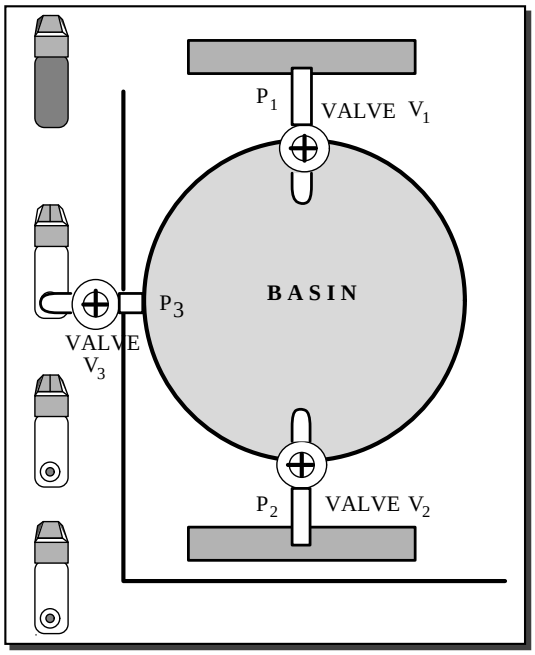


Figure 2: Concept model of truck depot (Aerial view)

each truck leaves the basin, its cargo is tested. If the truck has been filled with an acceptable mixture, it leaves the system; otherwise it dumps the cargo and returns to be refilled. In our version of the problem, the capacity of each tanker trucks is constant.

When viewed from the simulation perspective, the simulation of the truck depot contains both intelligent and non-intelligent objects, as shown in the class model Fig. 3. The intelligent objects control the non-intelligent objects, see Fig. 4 trying to achieve their goal of maximizing profit. The intelligent object could be a human or a robot. In our case, the intelligent object is the human operator of the truck depot. The operator plans and controls the valves in the depot, while achieving the goal of maximizing the profit of the depot by maximizing the number of trucks filled while minimizing the cost. The depot is charged for the total amount of chemicals that flow through the input pipes during the period in which it is open. The basin, the trucks and the valves are the non-intelligent objects in our system. The trucks are independent objects which arrive according to an exponential distribution over the period of time when the depot is open.

We also have the notion of time to consider in our simulation. The start of simulation

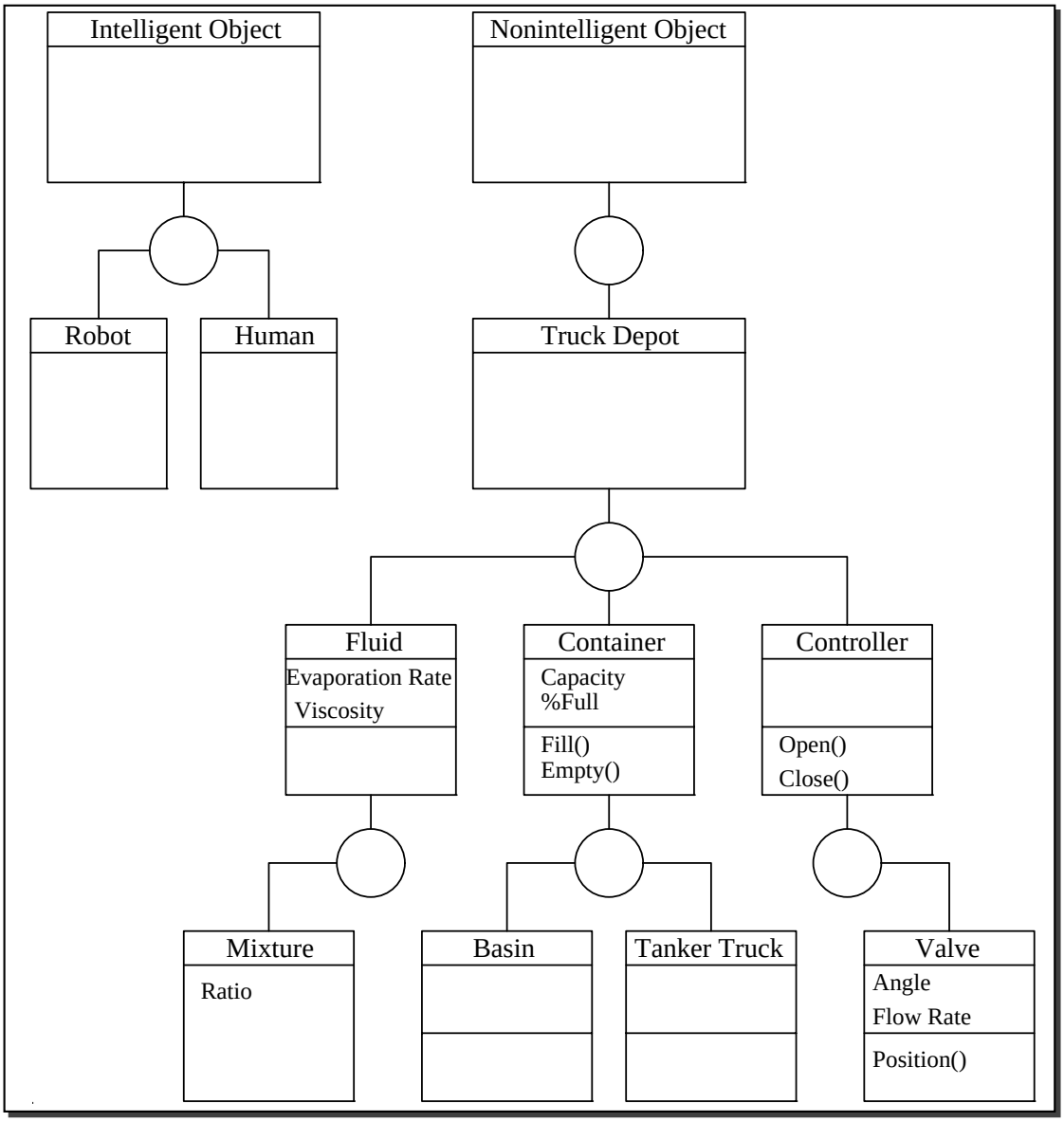


Figure 3: Class model of truck depot

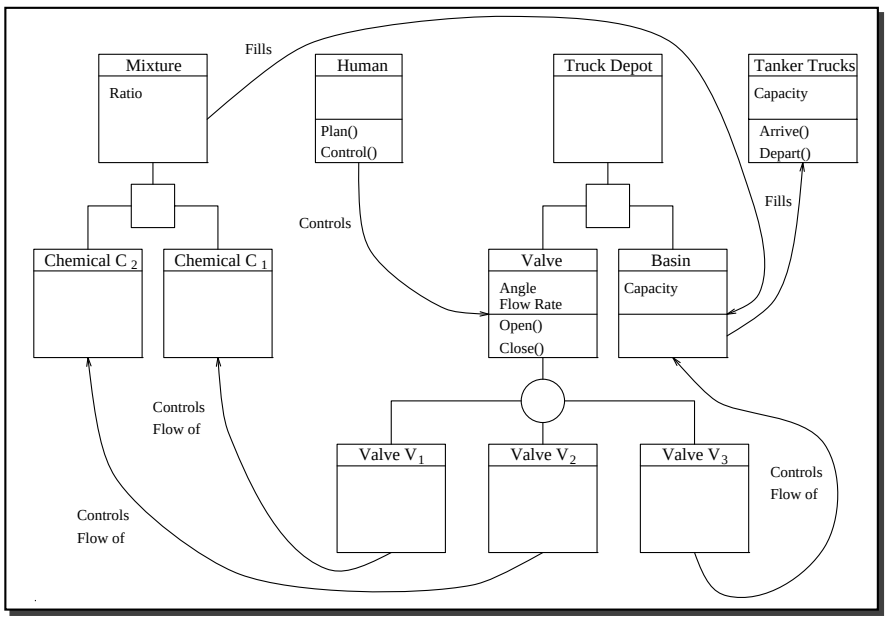


Figure 4: Instance model of truck depot

time corresponds to the start of a workday (opening time of the depot) and the end of simulation time corresponds to the end of the day (closing time) in the real world. Fig. 5 shows the control system of our simulation.

Let us examine the basin and the effect of the valves in more detail. Consider the basin containing a mixture (as illustrated in Figure 6). The two input pipes $P_1$, $P_2$ carry two different chemicals. Pipe $P_3$ fills each tanker truck with a mixture of the two chemicals. Each pipe has a valve ($V_1$, $V_2$ and $V_3$) which controls the chemical flow. Each valve has a servo motor attached that is controlled remotely and simultaneously by the intelligent object. The intelligent object controls each of the valves by opening or closing them.

The primary goal of the intelligent object is to fill the tanker trucks with an acceptable mixture while minimizing the system cost. If we further extend this problem to a real world situation, we can consider that the owner of the depot is paid for each truck properly filled and is charged for the total volume of chemical that flows through $P_1$ and $P_2$. Thus, the owner's goal is to use the least amount of chemicals while filling as many trucks as possible. All mixture which overflows from the basin or the truck is treated as waste. Also, any remaining mixture in the basin after some deadline (e.g.

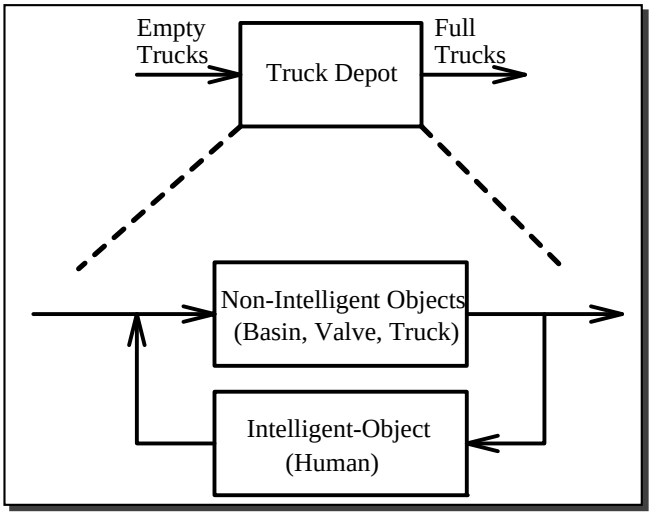


Figure 5: Top view of system

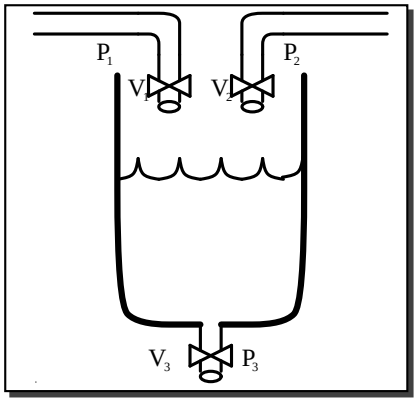


Figure 6: Basin containing mixture

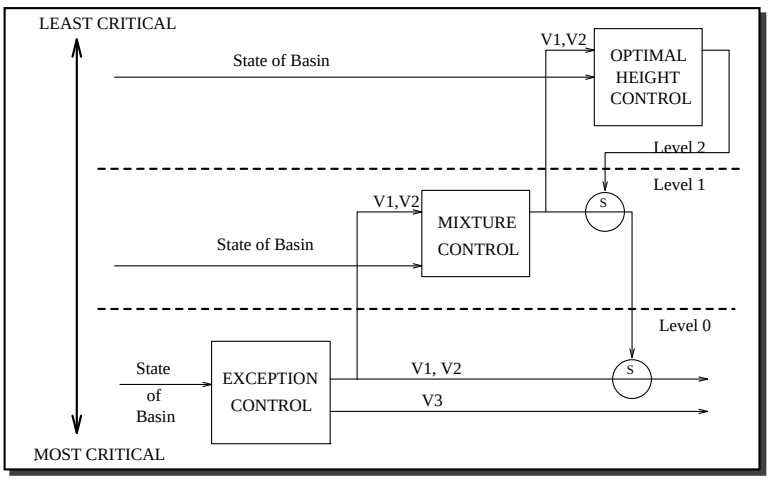


Figure 7: Multimodel planner

end of the day) will also be waste. And finally, when a truck contains an unacceptable mixture, the contents are dumped and refilled later. The mixture is not acceptable (or *bad*) when the proportions are off by more than 10% from the correct ratio. It is considered acceptable (or *good*) otherwise.

# 4 Intelligent Objects

From the definition of the problem, we can identify aspects which have to be dealt with reactively or deliberatively. Because the arrival of the trucks is dynamic and there is no determined number of trucks *a priori*, no type of offline planning is possible. Therefore, we require a planner that is able to plan and react in parallel.

As illustrated in Fig. 7, the overall architecture of the planner is a multimodel. Due to the reactive nature of the problem, we have adopted Brook's subsumption architecture [Bro86] to integrate the different level modules. Our hierarchical approach is different from a conventional hierarchical planner in the sense that each of the levels have access to input and output. Since it is possible to have conflicting output commands, we need some type of coordination or mediation [Kae87] among them. Adopting the subsumption architecture's method of mediation, the outputs are suppressed

by a higher level when the higher level makes an overriding decision. In the original version of the subsumption architecture, a time period is specified, during which the output will be suppressed. However, because our simulation is discrete, we will only allow the output to be suppressed for one time step until the next event arrives and causes another output.

The most significant aspect of this planner is, however, the multimodeling concept. Under multimodeling, each module can use different type models that best suit the task to be accomplished. The multimodel planner has multiple levels that are divided based on how critical the reaction of each level is to the overall success of the planner. The levels also reflect the reactivity of the control in that the lowest level module is the most reactive module whereas the highest level is the least reactive. Thus, the two properties of reactivity and criticality coincide in their degrees at each level. In general; however, this may not always be the case. The least reactive module may contain the most critical level of control in the system. The following hypothetical example illustrates a typical case where the more reactive modules are also more critical to the overall success. For example, when they plan deliberatively to decide where to turn while driving or controlling the car at the same time. If the car reaches a red light before the turn, it is more critical that the driver react and stop at the red light than worry about the turn.

## 4.1 Exception Control

The lowest level in the hierarchy contains the module, Exception Control, that is the most reactive and the most critical. In our problem, there are two critical situations. The first is when the mixture is overflowing or is about to overflow from the basin or the truck. This must be avoided at any expense, since spilled mixture can never be recovered. The second situation arises when the basin is empty. This situation may not be as critical as overflow but it is just as important. Unless it is close to the end of simulation time, the planner should avoid having an empty basin since no truck can be filled. The Exception Control layer is responsible for controlling the forementioned situations. The Exception Control takes the state of the basin as input, which is the volume of the mixture in the basin, $B_{vol}$ and the volume of the mixture in the truck, $T_{vol}$.

With $B_{vol}$, fuzzy logic [Zad88] is used to infer whether the basin is in an OVERFLOW or EMPTY state. With $T_{vol}$, fuzzy sets are used to decide if the truck is in an OVERFLOW

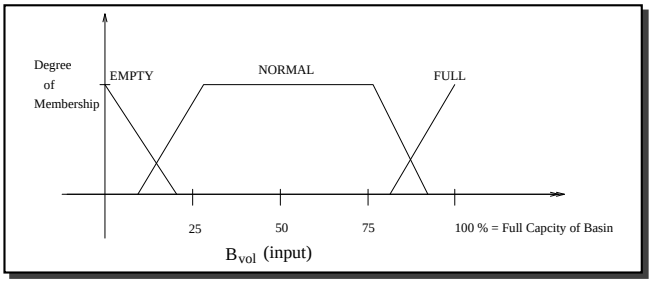


Figure 8: Fuzzy set for mixture height

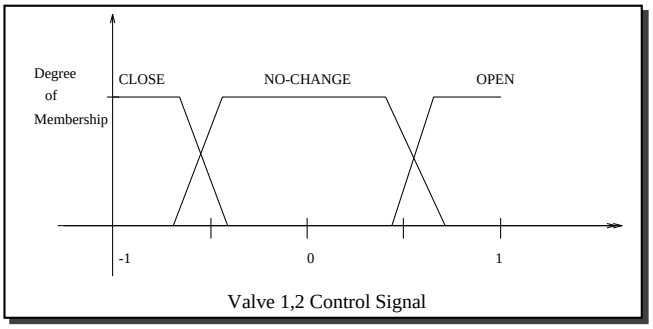


Figure 9: Fuzzy set for valve control

state. Only when these conditions arise, does Exception Control react and send an output command. A null command is sent otherwise. The output commands are valve settings that can be either CLOSE, OPEN or NO-CHANGE (which corresponds to a null signal) valve $V_n$. The fuzzy input and output sets for OVERFLOW CONTROL of the basin are illustrated graphically in Fig. 8 and 9.

The output set is identical for both valves $V_1$ and $V_2$, because when an overflow occurs or is about to occur, both of the valves need to be closed at the same time. To control the overflow of the truck, the volume of the mixture in the truck is monitored and inferred by Fig. 10 to be either in the state NOT_FULL or FULL. Note, since we are only concerned when the truck is full and therefore close to overflow, all the other states (e.g. EMPTY, NORMAL) correspond to one superstate NOT_FULL. The output set for valve $V_3$ is similar to the one for valves $V_1$ and $V_2$.

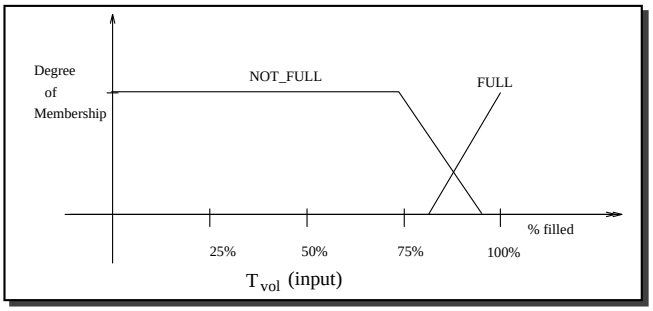


Figure 10: Fuzzy set for truck height

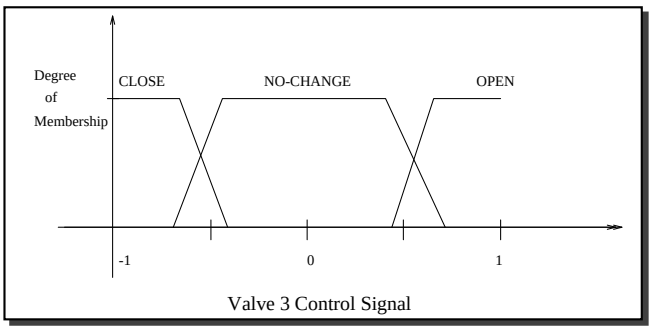


Figure 11: Fuzzy set for valve 3 control

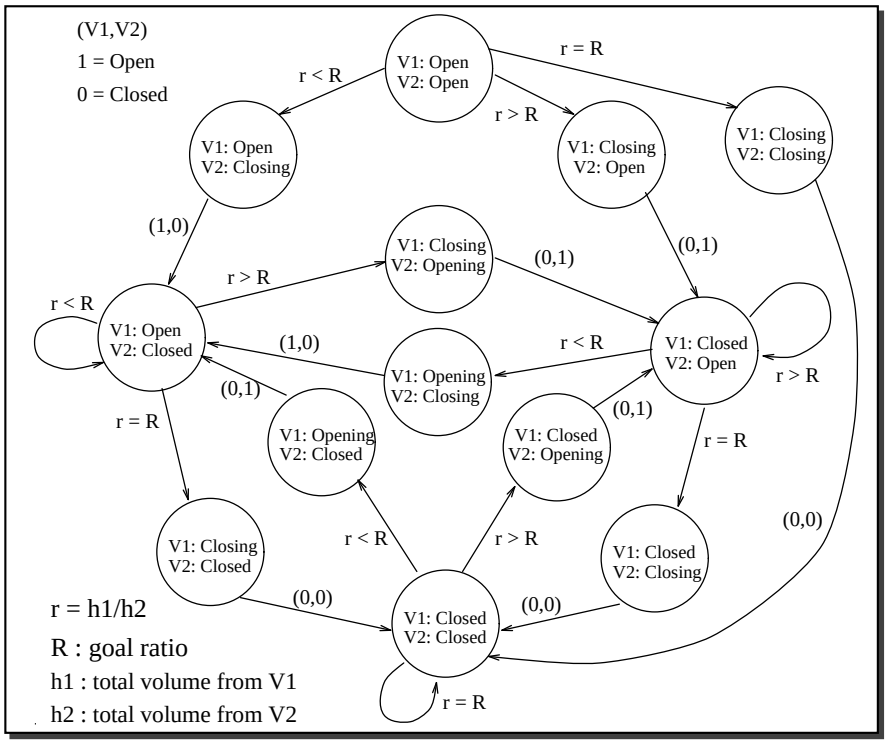


Figure 12: FSA for Mixture Control

As discussed in [SK92, Cox92], fuzzy control has proved to be successful in many practical applications. Because the point where the basin overflows or becomes empty is exact, fuzzy logic may seem unnecessary. However, we are not concerned precisely when these events occur, but rather the appropriate time to start monitoring and controlling to prevent overflow. The fuzzy logic model best mimics the actual performance of the human operator.

## 4.2 Mixture Control

The Mixture Control module is responsible for maintaining the *correct* ratio of the two chemicals in the basin. Initially, the planner is given a ratio $R$ which is to be maintained throughout the simulation. The mixture is considered acceptable and to be of the *correct* ratio if the actual ratio $r$ falls between the range $R - 5\%$ and $R + 5\%$. The type of model used for Mixture Control is a finite state automata as shown in

Fig. 12. Of the 13 states, 4 states represent all possible combinations of valve openings for the two valves - $V_1$ open or closed and $V_2$ open or closed. The remaining 9 states are *transition* states that connect the 4 states depending on the input. Transition states represent the transitions that the valves are assuming. For instance, the transition state Opening switches the Closed state to Open state. Stated in more syntactic terms, transition states are states that contain one or more valve settings ending with *ing*(e.g. participles).

The module takes as input from the basin model the current ratio $r$ and the current state of the valves. The valves can either be Open, Opening, Closed or Closing. In the figure, the current state of the valves is represented by the tuple $(V_1, V_2)$ where $V_n$ can be one of the following values: $1$ = Open, $0$ = Closed, $2$ = Opening, $-1$ = Closing[1]. Depending on what the current state is in the FSA, the next state is reached by choosing the appropriate arc. If the next state that is reached is a transition state, an output command is sent that will actually create the event to change the physical state of the valves.

For example, at time $t$, the current state is (Open,Open). Then at time $t+1$(i.e. the next even time), the input is $(r, 1, 1)$. If $r < R$, the Mixture Control module will switch the current state to (Open,Closing). The current state of the fsa will stay at this state until the input is $(r, 1, 0)$. There is an implicit self loop (which is omitted in the figure) that transitions to itself when the input is anything other than $(r, 1, 0)$. When the input becomes $(r, 1, 0)$ at some time $t + n$, where $n$ is the time delay for the valve to be totally closed, the current state switchs to (Open, Closed). The input $(r, 1, 0)$ is a confirmation from the basin model that the commands were properly carried out and that valve $V_2$ has successfully closed.

Once the ratio of the mixture becomes acceptable and close to its goal ratio $R$, Mixture Control will send commands to close all the valves to maintain the steady state. However, this command may not reach the basin model since the higher level module can suppress the command. When sending the output command, the Mixture Control module should also consider the output coming in from the Exception Control module. Taking this into account, the Mixture Control module must decide which output is more critical. If the Mixture Control module decides that its output is more critical, it will suppress the lower level output and replace the command with its own. For more detailed explanation of the suppressor function, refer to [Bro86].

[1]Note, these numbers are not related to the fuzzy set values in Exception Control.

## 4.3 Optimal Height Control

Finally, the Optimal Height Control module controls the optimal height in order to maximize the profit. Because this module is less reactive and involves more symbolic knowledge and reasoning (heuristics) than the other lower level modules, we believe rule-based reasoning is well suited for the task. This module uses production rules to reason what the next action should be to maintain an optimal height.

The notion of optimal height is time dependent. Since the simulation has a start and an end time, the intelligent object can have different strategies for maintaining the height at different times. As soon as the number of trucks waiting in queue is greater than 1, the Optimal Height Control is able to predict, at that point in time, the minimum amount of mixture needed to fill the trucks waiting. Of course, the prediction may have to be changed at the next time step if more trucks arrive or a truck leaves the queue after it is filled. When the time draws near to the end of simulation time, the Optimal Height Control may decide it will only keep enough mixture in the tank to fill those trucks that have high probability of getting filled, given the remaining time available.

Another consideration that Optimal Height Control module takes into account is the speed of chemical flow. The flow rate of valve $V_3$ depends on the height of the mixture in the basin. If the height of the mixture is high, there will be more pressure and the flow rate will be faster for valve $V_3$. Since our Optimal Height Control module uses heuristics, optimality is not guaranteed.

## 4.4 Evaluator

The Evaluator evaluates the overall profit of the system, during and after the simulation is over. This function is included as part of the Optimal Height Control module.

$$Profit = N * (totalvolumeofgoodtrucks) - C * (totalvolumeofinput)$$

where $N$ is the amount of reward per unit of volume and $C$ is the amount of money charged per unit of volume.

# 5 Simulating Multimodels

## 5.1 What are Multimodels?

Models that are composed of other models, in a network or graph, are called multimodels [Fis93a]. Multimodels allow the modeling of large scale systems at varying levels of *abstraction* without loss of accuracy, and they combine the expressive power of several well known modeling types such as FSAs, Petri nets, block models, differential equations, and queuing models. By using well known models we avoid creating a *new* modeling system with a unique syntax. Each of the base model types are unchanged. The multimodeling system is more accurate than any one of the individual model types since at every level of abstraction the model which provides the most information is used. When the model is being executed at the highest level of abstraction, the lowest level (representing the most refined parts of the model) is also being executed. Each high level state duration is calculated by the refined levels which have a finer granularity.

Multimodeling allows the model designer to define the system from an object oriented approach. The designer defines the system specifying high level concepts, such as synchronization, random arrivals and autonomous objects. Then, at a later step, the designer has the ability to describe each object's state transition using a refined method, such as differential equations or block models. The lower levels of the system are refinements of the components of the higher level models, which give the model more accuracy. Each level of the model has a phase space that is created by the state trajectories; the higher levels are abstractions or partitions of the lower level phase space.

## 5.2 Why use Multimodels?

For the Truck Depot example the question arises, "*Why use multimodels?*" The non-intelligent objects in the system could be modeled using control functions since classical control theory would provide an optimal solution. Therefore, using a multimodel may not seem necessary. However, multimodels do offer several advantages over using classical control theory.

- *Extendiblity*: A multimodel is extendable. Making a change to the model, (e.g. adding evaporation of the mixture while in the basin, having variable capacity of the tanker trucks being filled, or adding uncertainty to the model by not allowing the planner to know the exact valve positions) is difficult, if not impossible, to implement when using classical control theory.

- *Replaceability*: Any of the objects in the system can be replaced by another object that accepts the same input and gives the same output. For example, the two input valves can be replaced by three input valves, or the intelligent object can be replaced by another type of model, such as a neural network, or a case based reasoner.

- *Reusability*: While closely linked to replaceability, another strength of the multimodeling approach is the reusability of objects. The planner, the basin or the *entire* truck depot could be used within the context of a much larger model containing the depot asa component.

- *Comprehensibility*: Any physical system can be modeled using classical control theory, but it has drawbacks. For example, the equations for large scale systems the become prohibitively complex and unsolvable and when small changes are made to the system often the model must be recreated and resolved, a loss of some or all previous work.

## 5.3 How to use Multimodels

Each type of model (e.g. Petri net, block, differential equation, FSA, queuing, etc...) has similar features, *input*, *output*, *state*, and a *transition* from one state to the next. Some model types have lower level components, that encode information about the model. In an FSA, each state holds all of the information that is needed to answer any question about the model. In contrast, a transition and place in a Petri net each only have information about a subset of the system, although when combined, they describe the complete system.

To simulate a multimodel, it is necessary to have the input from one model type accept the output from another type of model. Each model must be able to recognize the output of any refining model as *valid* input and the input should cause the system to change state.

To synchronize the system at its highest level, a coordinator is used to process the external inputs and to call the appropriate model. The coordinator also creates and initializes each model and its components, and then organizes the models into the specified hierarchy. During execution, if a model has a refining model, the output of the refined model is used to update the system state. The coordinator executes the refining models as deep as necessary, but allows only external output from the levels above the specified level. The coordinator uses a future event list (FEL) to keep track of: 1) the next event, 2) to which model or model component the event should be sent, 3) which token caused the event, and 4) the global time of the simulation. Each level of the model must wait for an event to begin execution, then it posts its new state to the FEL.

The following methods are used to simulate each of the model types. In general, they comprise all the methods that are necessary to simulate each of the models that are used. However, the individual implementations are quite different internally. The goal of each method is listed below:

- *ReadModel()* reads the model specifics, (the type, topology, transition functions, and refining models) and creates the model instance.
- *Initialize()* describes how the model and its lower level components are set in the initial state.
- *Input()* collects all of the input data that a model receives during execution. Input() represents both *internal* and *external* events.
- *State()* returns the current system state.
- *Update()* causes the input or event to be applied to the current state and the next state to be scheduled on the FEL.
- *Output()* returns the output of the current state.

### 5.3.1 Petri Nets

Petri nets are composed of transitions and places, each must be managed both as individuals and in parallel. Petri nets are used to model conditions, resource sharing, synchronization problems, and other discrete spaces. We use the extended Petri nets

with colored tokens and timed transitions in our implementation. For the multimodeling refinement process, places never have a refining model, only transitions. Places are used only to show if proper conditions have been met to allow any of the output transitions to *BEGIN_FIRE*. A transition may have a refining model that is executed every time a transition fires. It determines the time to produce a token. If the transition does not have a refining model, then the transition time is sampled from a normal distribution with a specified mean and standard deviation.

- *Initialize()* calls each place and transition and runs their Initialize() functions. The place Initialize() function sets the initial number of tokens for that place, and schedules all of the output transitions if the place has any initial tokens. The transition Initialize() tests to see if it contains a refining model, if it does then it calls the refining model's Initialize() function.
- *Input()* collects external inputs that cause a transition to begin firing. When a Petri net is created, external inputs are connected to some transitions. When the external input is received, the connected transition begins to fire. Input() for a place is the number of tokens to be added when an input transition has finished firing.
- *State()* is the number of tokens in each place.
- *Update()* is null for the Petri net model, it is handled internally by the transitions and the places. Update() for transitions is called with either a *BEGIN_FIRE* or an *END_FIRE* event, with any other event sent to the refining model.
- *Output()* is a vector containing the number of tokens in each of the places that are marked as output places.

### 5.3.2 Queuing Model

Queuing models are used to show how different numbers of servers affect production, and how objects move through a system where they are forced to wait. It is different from the Petri net, since the order of the objects in the queue can change and the objects can have priorities, whereas the Petri net has no control over the ordering of its tokens. The queuing model has two components, the queue and the server. The queue is not refined, instead it keeps track of the tokens waiting and how long they have been waiting. The server can have a refining model to to determine the exact service time. If

there is no refining model then the service time is sampled from a normal distribution with a specified mean and standard deviation.

- *Initialize()* sets the number of objects waiting in the queue and in the servers.
- *Input()* returns tokens to the queue that wait for service. Some types of input may interrupt the server and cause it to delay processing one object in favor of the new object; this is a *PREEMPT* event.
- *State()* is the number of tokens in the queues and being serviced.
- *Update()* recognizes only four events: *ARRIVING*, *PREEMPTING*, *DEPARTING*, and *BEGIN_SERVICE*.
- *Output()* is the token that has been processed through the servers.

### 5.3.3 Finite State Automata

Finite state automata (FSA) are useful when modeling the discrete states a system can enter. FSA have two components, states and transitions. States are unique and fully describe the system, while transitions are inputs that move the system from state to state. States can be refined by other types of models, that when executed determines the transition time taken in the state. If no refinement exists the transition time is sampled from a normal distribution with a specified mean and standard deviation.

- *Initialize()* sets the current state of the model. It also calls the Initialize() function for each state. If a state has a refining model, the refining model is initialized.
- *Input()* sets the current input for the system. It does not differentiate between external and internal inputs.
- *State()* is the current state of the system.
- *Update()* determines the next state by sending the input to the current state. The current state returns the next state.
- *Output()* returns an event defined for each state, the output is valid when the state is entered.

### 5.3.4 Block Model

Block models are a type of functional modeling where each block represents a function. The function is applied to the input to produce an output. The block models may be coupled together to form block networks. Block models have many components, some of the frequently used components are: adder, multiplier, and integrator. But the function could be complex and can only defined by a procedure. The difference between using a block model and an equation is that the blocks generally correspond to physical components or software routines that perform the function.

- *Initialize()* sets the initial value for all of the block outputs and for any integrators that are defined in the system.
- *Input()* sets the signal level for each of the inputs to the function block.
- *State()* returns the output of each of the function blocks.
- *Update()* performs the specified function on the input signals and returns an output signal.
- *Output()* is the outputs of the selected function blocks.

### 5.3.5 Differential Equation

When a model requires continuous time/continuous state description, then we must specify that part of the model with differential equations [Fis93a]. The general form for a differential equation is either of the following:

$$\frac{dx(t)}{dt} = f(x,t) \quad (1)$$

$$\dot{x} = Ax \quad (2)$$

Differential equations are used at (or near) the lowest level of abstraction in multi-models, to provide information about those objects in the system that have continuous states. The only components for a differential equation is the state vector $x$ and the matrix $A$ that defines the system.

- *Initialize()* sets the the state vector to the initial state vector $x_0$.
- *Input()* sets specified variables in the state vector.
- *State()* is the state vector $x$.
- *Update()* solves the equation over the specified interval.
- *Output()* is the state vector $x$.

# 6 Non-Intelligent Objects

## 6.1 Model Design

Modeling the basin poses several challenges. The model state includes continuous and discrete variables, constraints, and functional relationships.

The volume of the mixture in the basin changes continuously throughout the simulation. The input signals that control the three valves give continuous outputs, but they change at discrete times. The trucks move through the system as discrete objects, and are constrained when waiting to be filled because: the signal for valve $V_3$ must open the valve; the basin must have enough mixture to fill a truck; and the filling area must be empty, (since only one truck can be filled at a time).

### 6.1.1 Level One

The main constraint in the model is when the basin fills the trucks; since this activity has many conditions. Based on this constraint, we choose a Petri net to model the top level (Level One), see Fig. 13. The inputs to the model are tanker trucks arrivals and control signals for opening and closing each of the valves. The output from the model is statistics showing the number of trucks that were filled properly, and the volume of each chemical that is poured into the basin during the simulation. Each of the transitions is explained in detail as follows:

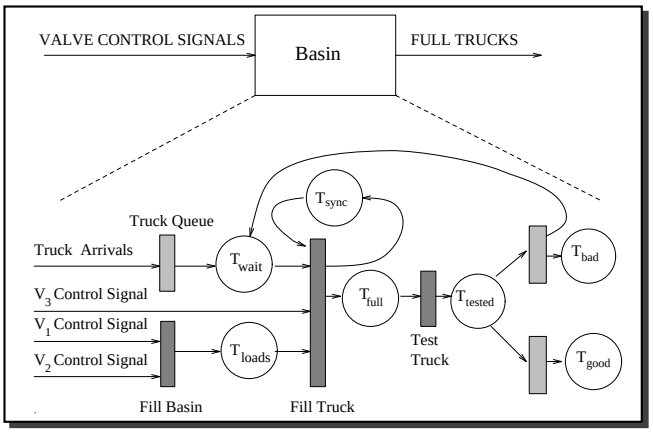


Figure 13: Petri net model of the basin

- *Truck Queue* is where all trucks must wait until the basin is ready to fill a truck. Truck arrivals are received as input and each truck waits until it can be filled. When the filling area in the basin is empty the next tanker truck is allowed to leave the queue to begin to be filled. This transition is refined by a queuing model, (see Fig. 14).

- *Fill Basin* takes the control signals for valves $V_1$ and $V_2$ as input, and outputs tokens which represent how many tanker trucks can be filled with the volume of mixture currently in the basin. This transition is refined by a block model, (see Fig. 15).

- *Fill Truck* is fired when: 1) enough mixture is in the basin to fill at least one truck, 2) the signal controlling valve $V_3$ is *OPEN_V3*, 3) there is at least one truck waiting to be filled, and 4) no truck is currently being filled. A full truck is output to be tested to check if it contains an acceptable mixture. This transition is refined by a block model, (see Fig. 16).

- *Test Truck* takes the trucks as input after they have been filled and tests the mixture to check if it is an acceptable mixture, firing either a *good* or a *bad* token based on the results.

### 6.1.2 Level Two

At the second level each of the transitions in Level One was refined where necessary.

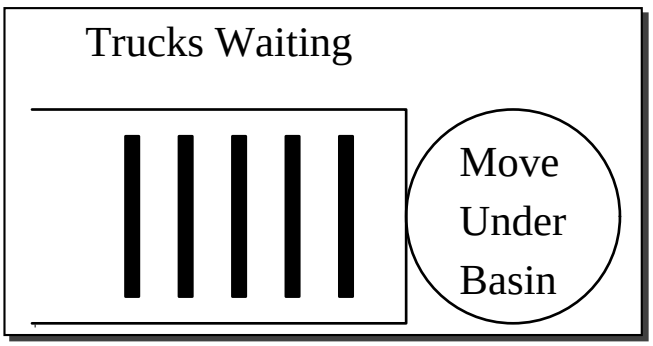


Figure 14: Refinement of transition *Truck Queue*

- *Truck Queue* is refined by an S/S/1 queuing model. It was chosen to model the trucks waiting to be filled. The queue is used to maintain the arrival order of the trucks. The model has one queue and one server. Preemption and prioritized tokens were not necessary in our implementation, though with the queuing model this extension is easy to implement.

- *Fill Basin* is refined by a block model. It was chosen to represent the relationship between the valves $V_1$ and $V_2$, and the volume of mixture in the basin. The control signals go into function blocks that pass the signals to a lower level that eventually returns with a value for the control to be applied due to the current state of the valves $V_1$ and $V_2$. The function block, *Mixture Volume in Basin* takes the control input and returns a value that states how many tanker truck loads the basin currently holds.

- *Fill Truck* is also refined by a block model that shows the relationship between valve $V_3$ and the volume of mixture filling the truck, see Fig. 16. Valve $V_3$ takes as input the control signals, *OPEN_V3* and *CLOSE_V3* and returns the amount of control to be applied due to the current state of valve $V_3$. The function block *Fill Truck* takes the control value as input and returns a value when the truck has been filled.

### 6.1.3 Level Three

Each of the function blocks from the second level of the model is refined by a finite state automata. The FSAs are used to *translate* between the differential equations that

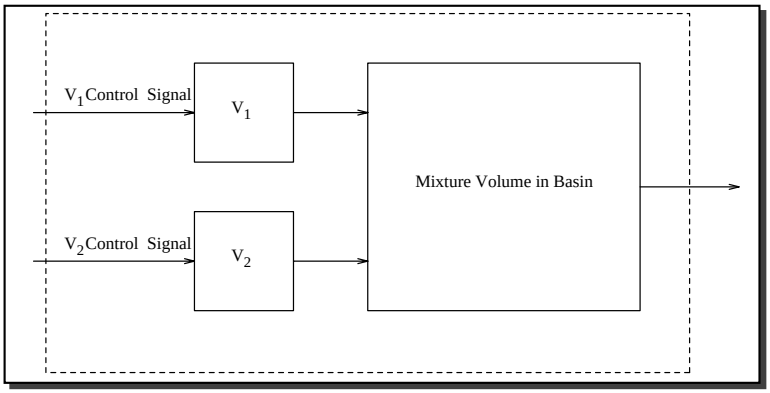


Figure 15: Refinement of transition *Fill Basin*

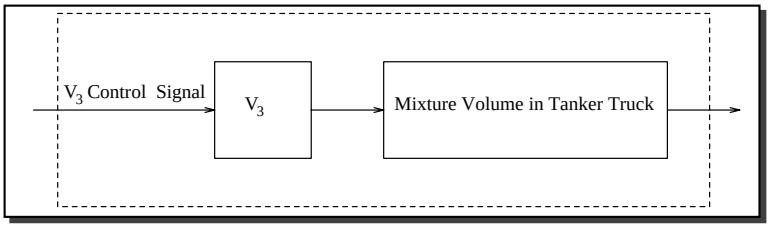


Figure 16: Refinement of transition *Fill Truck*

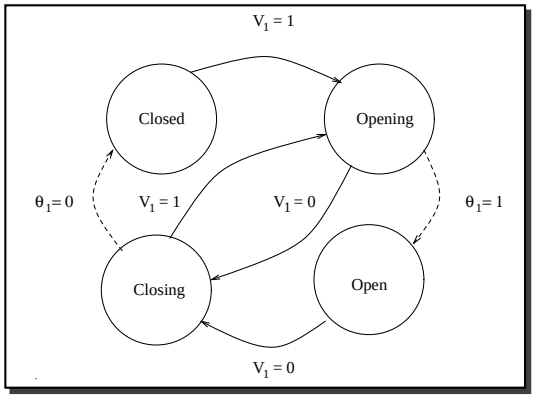


Figure 17: FSA model of valve $V_1$

are in level four and the Petri net from the first level. There are five FSAs, but all three FSAs that refine the valves are similar to the refinement of valve $V_1$.

- *Valve* $V_1$ is refined by the FSA shown in Fig. 17. The input for this FSA is the control signal for the valve $V_1$, ($V_1 = 1$ or $V_1 = 0$). The FSA can also change state if an internal transition is detected, ($\theta_1 = 1$ or $\theta_1 = 0$). The internal transition is produced by the refining differential equations. The FSA that refine valves $V_2$ and $V_3$ are similar to Fig. 17, the only difference is the events that cause the system to change states.

- *Fill Basin* is refined by the FSA displayed in Fig. 18. The input for this FSA is the amount of control that is being applied to the system by valves $V_1$ and $V_2$. The output is the state of the system, that is how many truck loads of mixture the basin contains. All of the state transitions are caused by internal events.

- The function block *Mixture Volume in Truck* is refined by the FSA illustrated in Fig. 19. The input to this FSA is the amount of control applied by valve $V_3$, the system output is the state of the truck, whether the truck is being *FILLED*, *FULL* or *OVERFLOWING*. The transitions from *FILLED* to *FULL* and from *FULL* to *OVERFLOWING* are internal transitions.

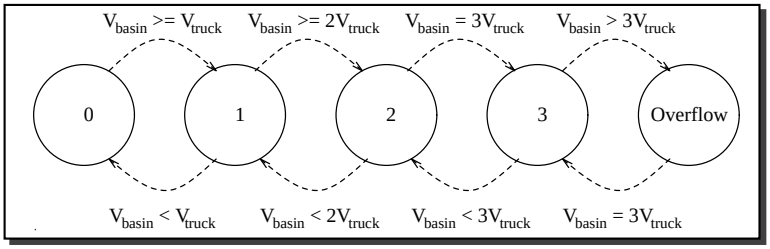


Figure 18: FSA model of volume of mixture in the basin

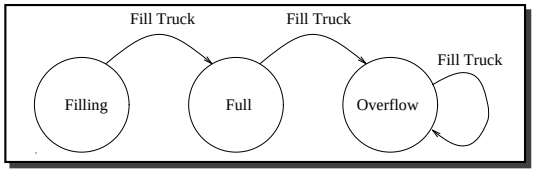


Figure 19: FSA model of volume of mixture in the tanker truck

### 6.1.4 Differential Equations

At the highest level of refinement (the lowest level of abstraction) the model must represent the continuous elements of the system, the volume of the mixture in the basin and in the tanker trucks. Differential equations model continuous change in state and time and can be applied at this level by use of the control function $\dot{x} = Ax + Bu$. Where $x$ is the subsystem state (the volume of mixture in the basin or in the tanker truck) and $u$ is the control from the valves, and $B$ is the amount of control being applied.

## 6.2 Model Execution

A coordinator is used to create and execute the multimodel system. After each level of the model is created, the levels are connected to their refining models, then each level is initialized. The initialization includes setting the initial states of the valves, the next arrival of a truck, and the initial volume of mixture in the basin. For our example we choose the basin to be empty, the valves to be closed and no trucks waiting to be

serviced. The coordinator's task during execution is to dequeue events from the FEL and direct it to the model specified.

The coordinator recognizes these external events: *TRUCK_ARRIVAL*, *OPEN_V1*, *CLOSE_V1*, *OPEN_V2*, *CLOSE_V2*, *OPEN_V3*, and *CLOSE_V3*. The coordinator must also handle the following internal events: *TRUCK_ENTER*, *TRUCK_EXIT*, *V1_IS_OPEN*, *V1_IS_CLOSED*, *V2_IS_OPEN*, *V2_IS_CLOSED*, *V3_IS_OPEN*, and *V3_IS_CLOSED*. Finally, there is a controlling event that is set to mark the end of the simulation, *END_SIMULATION*.

The coordinator is shown in the following pseudocode:

```
while (Model::GetTime () < END_TIME)
{
    event_list.next_event (event, token, model);
    if (model.norefiningmodel())
    {
        switch (event)
        {
        case TRUCK_ARRIVAL:
            event_list.schedule (TRUCK_ARRIVAL, 0, token, TRANSITION_0);
            token.putid (counter++);
            event_list.schedule (TRUCK_ARRIVAL, expntl (.3), token);
            break;

        case OPEN_V1:
        case CLOSE_V1:
            event_list.schedule (event, 0, token, TRANSITION_1);
            break;

        case OPEN_V2:
        case CLOSE_V2:
            event_list.schedule (event, 0, token, TRANSITION_1);
            break;

        case OPEN_V3:
        case CLOSE_V3:
            event_list.schedule (event, 0, token, TRANSITION_2);
            break;
```

```
            case V1_IS_OPEN:
            case V1_IS_CLOSED:
                event_list.schedule (event, 0, token, TRANSITION_1);
                break;

            case V2_IS_OPEN:
            case V2_IS_CLOSED:
                event_list.schedule (event, 0, token, TRANSITION_1);
                break;

            case V3_IS_OPEN:
            case V3_IS_CLOSED:
                event_list.schedule (event, 0, token, TRANSITION_2);
                break;
            }                       // switch
        }
        else
        {
            model.Input (event);
            model.Update ();
        }
    }                               // while
```

The coordinator loops through the simulation updating the simulation time after each event is dequeued from the FEL, (time is static in the model class, so each model shares the same simulation time) until the end simulation time is reached.

# 7 Conclusions

Through our truck depot example, we demonstrated how to integrate simulation and planning tasks under the object-oriented multimodel framework. The designing of the model for our system is performed through the 3 phases 1) concept model, 2) class model and 3) instance model with the relationships specified as discussed in section 1. In using multimodeling, we have provided general policies and some specifics involved in modeling commonly used model types. As shown, multimodeling cannot only be used to integrate different type models in a hierarchy but also used to integrate model

types coming from completely different background or disciplines. Integration of models from simulation, AI, planning and control is one good example. By using our proposed approach, we are able to overcome the problems of integrating simulation and AI planning. Even in AI planning, we overcome the difficulties of integrating planning and control, which are due to the vast differences of models coming from two separated fields of study.

For future work, we would first like to extend our truck depot example to include more dynamic properties. For instance, have varying tanker truck capacity and allow the planner to reorder the trucks after they arrive. We would also like to experiment with different intelligent objects, such as a mobile robot and adding more constraints to the problem such as asynchronous control of the valves. Using adaptive control (e.g. adaptive fuzzy systems) will be another extension. For multimodeling in general, we will add a graphical interface for creating the models and a distributed persistent object database to store all of the objects which have been created. Then a user could load any object from the Internet.

# 8 Acknowledgments

We would like to thank the Institute for Simulation and Training (IST) at the University of Central Florida for partial funding of this research in connection with the "Mission Planning" sub-contract. We would also like to thank Douglas Reece and David Lambert for their comments.

# References

[Boo91] G. Booch. *Object Oriented Design.* Benjamin Cummings, 1991.

[Bro86] R. A. Brooks. A robot layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14 – 23, 1986.

[Cox92] E. Cox. Fuzzy fundamentals. *IEEE Spectrum*, pages 58 – 61, October 1992.

[FG90] I. Futo and T. Gergely. *Artificial Intelligence in Simulation.* Ellis Horwood Limited/John Wiley and Sons, 1990.

[Fis91] P. A. Fishwick. Heterogeneous Decomposition and Coupling for Combined Modeling. In *1991 Winter Simulation Conference*, pages 1199 – 1208, Phoenix, AZ, December 1991.

[Fis92] P. A. Fishwick. An Integrated Approach to System Modelling using a Synthesis of Artificial Intelligence, Software Engineering and Simulation Methodologies. *ACM Transactions on Modeling and Computer Simulation*, 2(4), 1992.

[Fis93a] P. A. Fishwick. *Computer Simulation Model Design & Execution*. Prentice Hall, 1993. (to be published as a textbook).

[Fis93b] P. A. Fishwick. Discrete event dynamic systems: Theory and applications. (accepted for publication), 1993.

[FZ92] P. A. Fishwick and B. P. Zeigler. A Multimodel Methodology for Qualitative Model Engineering. *ACM Transactions on Modeling and Computer Simulation*, 1(2):52 – 81, 1992.

[Har88] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514 – 530, May 1988.

[Har92] D. Harel. Biting the Silver Bullet: Toward a Brighter Future for System Developmen t. *IEEE Computer*, 25(1):8 – 20, January 1992.

[Kae87] L. P. Kaelbling. An architecture for intelligent reactive systems. In *Reasoning About Actions and Plans*, pages 395 – 410. Morgan Kaufmann, Los Altos, CA, 1987.

[Mil93] V. T. Miller. *Heterogeneous Hierarchical Modelling for Knowledge-Based Autonomous Systems*. PhD thesis, University of Florida, 1993.

[Nie91] N. R. Nielsen. Applications of AI Techniques to Simulation. In P. Fishwick and R. Modjeski, editors, *Knowledge Based Simulation: Methodology and Application*, pages 1 – 19. Springer Verlag, 1991.

[O'K89] R. M. O'Keefe. The Role of Artificial Intelligence in Discrete Event Simulation. In L. E. Widman, K. A. Loparo, and N. R. Nielsen, editors, *Artificial Intelligence, Simulation & Modeling*, pages 359 – 379. John Wiley and Sons, 1989.

[PAG93] H. Praehofer, G. Auernig, and Reisinger G. An environment for devs-based modeling in common lisp/closs. (accepted for publication), 1993.

[Pra91] H. Praaehofer. *Theoretic Foundations for Combined Discrete Continuous System Simulation.* PhD thesis, University Linz, Austria, 1991.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, E. Frederick, and W Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[SK92] D. G. Schwartz and G. J. Klir. Fuzzy logic flowers in Japan. *IEEE Spectrum*, pages 32 – 35, July 1992.

[TW91] D. L. Thomas and M. P. Wellman. *Planning and Control* . Morgan Kaufmann, 1991.

[Zad88] L. A. Zadeh. Fuzzy logic. *IEEE Computer*, pages 83 – 93, April 1988.

[Zei90] B. P. Zeigler. *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems.* Academic Press, 1990.