

AN OBJECT-ORIENTED APPROACH FOR HIERARCHICAL STATE MACHINES

WILLIAM MALLOUK¹
ESTEBAN CLUA²

¹ Wet Productions LLC
william@wetproductions.com

²Departamento de Informática – PUC-Rio / VisionLab
esteban@inf.puc-rio.br

Abstract

Finite State Machines have been widely used as a tool for developing video games, especially as pertains to solving problems related to AI, input handling, and game progression. In this paper, we introduce a practical technique that can highly improve the productivity of development of Finite State Machines in video games. This technique consists of adapting concepts of OOP for use with Hierarchical Finite State Machines in an entirely visual system. An open source implementation of the technique that can be used as middleware in most games is also provided.

Keywords: *Artificial Intelligence, Hierarchical State Machines*

1 Introduction

FSM (Finite State Machines) is a technique that appears in many different forms and is used in major game titles such as *Quake*, *Fifa Soccer*, and *Warcraft*. In fact, FSM is the most popular approach for AI algorithms used in games [4]. In this paper, we demonstrate how Hierarchical Finite State Machines can be combined with Object-Oriented Programming techniques to obtain productivity and HFSM reuse. We also provide an open-source implementation of the technique as middleware that can be used in nearly any game.

1.1 Previous Works

Although our approach to Finite State Machines is original, many of the ideas were inspired by [4] and [5]. Particularly, the implementation of the visual system for designing state machines is based on the ideas introduced in [2]. Our visual state machine XML parser, VDX2HSM, is an improved version of the tool created by the referenced author. See section seven of this article for more details on it.

2 Hierarchical Finite State Machines In Games

We recall from [1] and [4] the idea of *Finite State Machines* and *Hierarchical State Machines* as used in games. An HFSM is a FSM in which states can be decomposed into other FSMs [4], as depicted in Figure 1.

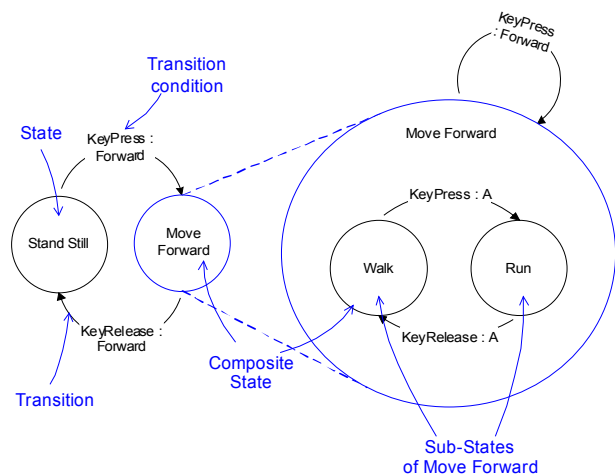


Figure 1. Hierarchical Finite State Machine

In the figure above, the decomposition of the state *Move Forward* is shown. In HFSMs, each sub-state is capable of being a composite state as well, forming a tree of state machines. This approach is convenient, especially when the state machine being modeled is huge and, therefore, presents the potential for confusing representation.

3 Object Oriented Approach

This paper introduces an hybrid system, where HFSMs are merged with concepts of OOP to create the *Object-Oriented Hierarchical State Machines* technique. Particularly, we adapted the concepts of *Class Hierarchy*, *Inheritance*, *Class*, *Method*, and *Data Abstraction* for use with HFSMs in an entirely visual system. In this paper, we assume that the reader is familiar with these concepts. Other related works are presented in [1] and [6].

4 Object Oriented Hierarchical State Machines

Object-Oriented Hierarchical State Machines, also known by the acronym, OOHSM, the result of combining *Hierarchical State Machines* with the Object-Oriented Programming concepts mentioned above.

The OOHSM technique consists of the following elements combined with each other: *State Machine*, *Base Machine*, *Sub Machine*, *Abstract State*, *Abstract State Machine*, and *Concrete State Machine*. We will see that each of these elements is analogous to an OOP element as we define them and provide examples of use in the next subsections.

To simplify the task of defining these elements, we have used properties of the UML (Unified Modeling Language) class diagram notation [7] because of its semiotic.

4.1 State Machine

Using our new notation, we visually redefined state machines. Figure 2 contains a state machine with two states represented using our new notation.

For the sake of simplicity, we can omit the details of the sub-states of a composite state, as

was done with the *Move Forward* composite state of figure 3. Note that a state machine can have as many composite states as necessary.

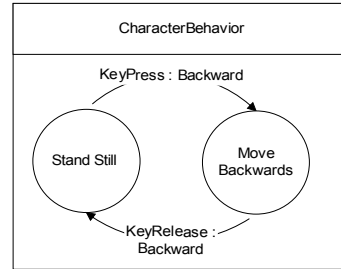


Figure 2. A State Machine

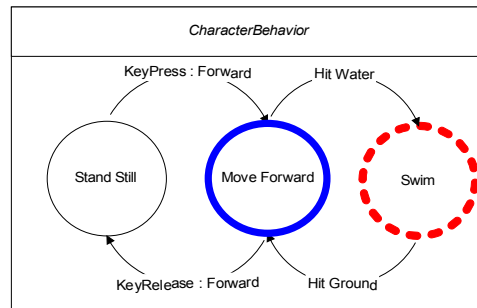


Figure 3. Composite State with omitted Sub-States

4.2 Base Machine and Sub Machine

In OOHSM, state machines can inherit properties from other state machines, forming an inheritance hierarchy that is similar and analogous to OOP *inheritance*. In Figure 4, the state machine *CharacterBehavior* is a super-machine of *PaladinBehavior*, and the state machine *PaladinBehavior* is a base machine of *CharacterBehavior*.

The difference between sub-machines and sub-states should be clear. Each composite state has one or more sub-states, and each state machine can have one or more sub-machines.

From the definitions above, it becomes apparent that a *state machine* is analog to a *class*, and that a *composite state* is analog to a *method*.

In some programming languages we can also compare an *object* with OOHSM *instance*, which is the terminology that will be used in the implementation of this work, which will be discussed later this article.

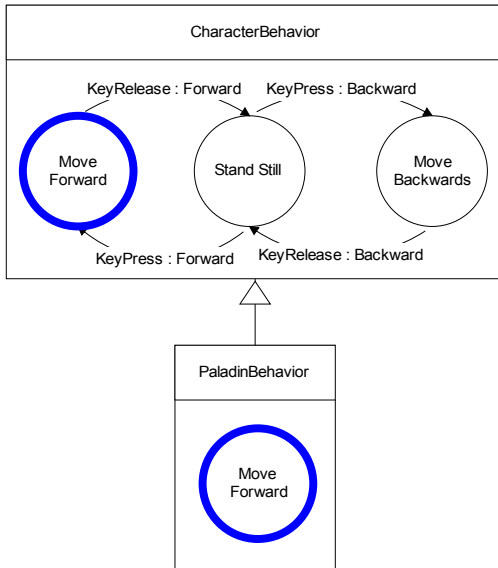


Figure 4. OOHSM Inheritance

4.3 Abstract State

An Abstract State is an undefined composite state. Throughout this paper, a dotted circle, such as the one in figure five, will be used to describe *abstract states*. The concept of *abstract state* is analogous to that of *abstract* or *virtual method* in OOP.

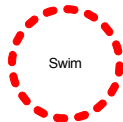


Figure 5. Abstract State Notation

4.4 Abstract State Machine

A state machine containing at least one *abstract state* is called an *abstract state machine*. *Abstract state machines* cannot be instantiated or used directly because at least one of its composite states is undefined. Abstract state

machines can be used when extended by state machines that contain the definitions for the *abstract states* contained therein. A sample abstract state machine is illustrated in figure 6.

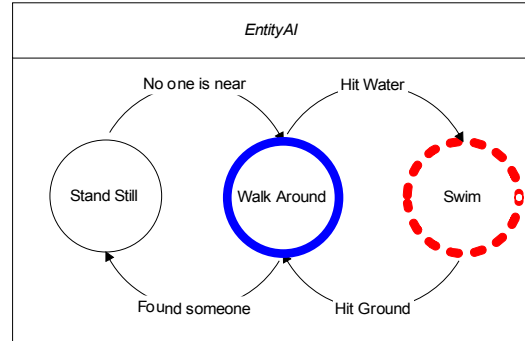


Figure 6. Sample Abstract State Machine

4.5 Concrete Machine

A concrete machine is defined as an OOHSM that has no *Abstract States*. The state machines shown in figures 2 and 3 are examples of concrete state machines. Concrete state machines are analogous to concrete classes in OOP.

In figure 7, we present an *instantiable*, and *concrete* state machine based on the EntityAI abstract state machine shown in figure 6.

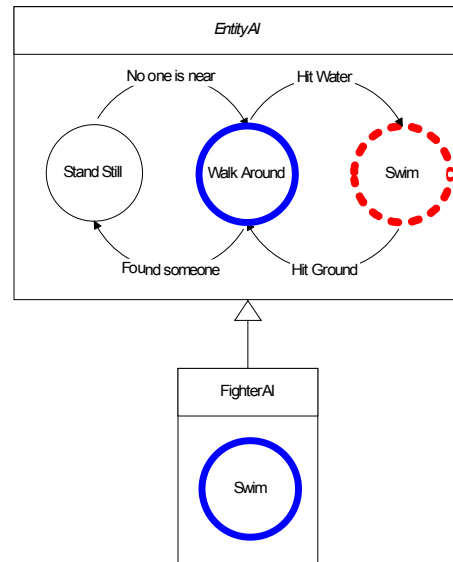


Figure 7. An inherited abstract state machine

The FighterAI state machine shown above has only one composite state called *Swim*. The *Swim* composite state is the definition for the abstract compound state *Swim*, which, in turn, is declared in the abstract state machine EntityAI. FighterAI inherits all the concrete states from EntityAI and provides an implementation for the Swim abstract state. Therefore, FighterAI is a concrete state machine, and it can be used or instantiated.

5 Notation Details

The Unified Modeling Language provides a graphical representation of class that permits us to visualize an abstraction regardless of the programming language being used. The UML *class* element is a description of a “set of objects that share the same attributes, operations, relationships, and semantics.” [7] Also, class diagrams provide several types of relationships such as dependencies, refinement, associations, and generalizations.

The notation used in this paper borrows elements from the Unified Modeling Language and from the standard state machine notation that is used by many authors, including [3]. We also introduced two new elements: the *composite state* symbol and the *abstract state* symbol. The diagram shown in figure 8 labels all the elements used throughout this paper.

The UML *Class* symbol was used to represent a state machine, as discussed in section 4.1. *Abstract* state machine names are written in italics. *Concrete* state machine names are written in normal text. *Generalization* was the only relationship type borrowed from the UML and it is used to model inheritance.

Elements that were borrowed from the standard state machine notation are *state*, *transition*, and *transition condition*.

Finally, the newly introduced symbols are the *abstract state* symbol and *composite state*

symbol. The latter is also used when overriding composite and abstract states.

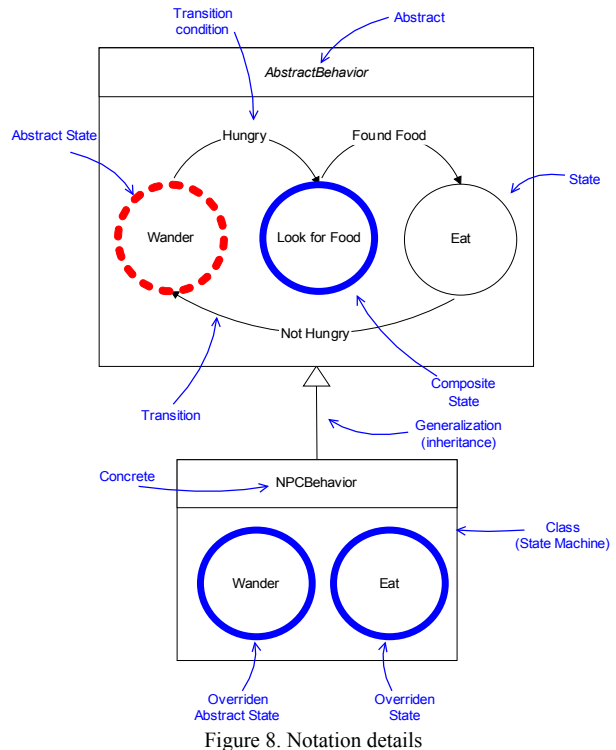


Figure 8. Notation details

6 Comparison With Other Techniques

Below we provide a comparison with Finite State Machines and the State Design Pattern, which are both related and relevant to this work.

6.1 Finite State Machines

One of the design goals of OOHSM is that all properties pertaining to Finite State Machines should also be applicable to them. More notably, the following constraints are applicable to both FSM and OOHSM.

- Operations are synchronized by discrete clock pulses.
- There is a finite number of states that the machine can attain (even though it is easier to create an OOHSM with a large number of states)
- At any given moment the machine is in exactly one of its states. What the new state

will be depends on the input, as well as what the current state is.

- The machine is capable of output, through user-defined functions.
- Finite State Machines are deterministic.

OOHSM also suits the formal definition of Finite State Machine provided by [3]: $M = [S, I, O, f_s, f_o]$ is a finite-state machine if S is a finite set of states, I is a finite set of input symbols, O is a finite set of output symbols, and f_s and f_o are functions where $f_s: S \times I \rightarrow S$ and $f_o: S \rightarrow S$. The machine is always initialized to begin in a fixed starting state s_0 .

OOHSM are in essence finite state machines, with the difference that OOHSM provide notational tools that give several advantages in the design phase, allowing, for example, the construction of very large state machines, and reuse of these state machines, as it was discussed above.

6.2 The State Design Pattern

The *State* pattern [11] is a behavioral software design pattern used to represent the state of an object. It is a solution to the problem of how to make behavior depend on state. It consists of defining a “context” class that presents a single interface to the world, a State abstract class and several subclasses of it, each of which defining a different behavior.

The main difference between the State Pattern and OOHSM is that in the pattern each state should be modeled as a separate class, and thus requires a separate implementation. In OOHSM, states do not require implementation. Each OOHSM state can have data or code related to it, but none are mandatory.

Another difference is that the State pattern does not specify where transitions should be defined whereas OOHSM transitions should be fully defined in OOHSM diagrams.

7 Sample Uses

Good examples of game genres that can take advantage of OOHSM are *Fighting Games* and *RTS (Real-Time Strategy) Games*. In many RTS games, different unit types share the same AI pattern repeatedly. The shared AI elements can be defined in base machines, and the AI elements that are specific for each unit type can be defined in separate state machines that inherit from the previously defined base machines.

We observed that in fighting games such as *Street Fighter*, *Tekken*, and *Dead or Alive*, multiple characters share the same set of basic movements such as “kick,” “jump,” and “punch,” as well as a set of character-specific moves. We concluded that in these types of fighting games, the shared moves can be defined in base machines, and the character-specific moves can be defined in sub-machines.

8 Benefits of using OOHSM

There are two main advantages for using OOHSM at the implementation of games. These advantages are described below:

- **Reduced Development and Maintenance Time:** OOHSM can save development time considerably when developing games that have HFSM with repeated patterns. Taking the genres defined in section five as examples, a great amount of development time can be saved by working only once on the behaviors for characters and units as defined in base machines. Maintenance can also be expedited because every character or unit which inherits from a state machine in a game can be updated by changing that state machine alone.

- **Ease of Use:** As a result of using a visual system to create the state machine diagrams, it becomes easy for non-specialized personnel (non-programmers) to create OOHSMs to describe the behavior of characters in games. This is especially important, because many times game designers are responsible for the definition of AI. A visual OOHSM system obviates the need for additional programming, even at scripting level.

[5] Jacobs, S. Visual Design of State Machines, Game Programming Gems 5.

[6] The free dictionary of computing: <http://foldoc.org/>

[7] Booch, G; Rumbaugh, J. and Jacobson, I. The Unified Modeling Language User Guide, pp 82, 1999, Addison-Wesley.

[8] www.cadabra3d.org

[9] <http://www.gnome.org/projects/dia/>

[10] <http://office.microsoft.com/visio/>

[11] Gamma, E; Helm, R; Johnson R. and Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software, pp 231, 2001 Addison-Wesley.