Component-Based Hierarchical State Machine -A Reusable and Flexible Game AI Technology

Wenfeng Hu, Qiang Zhang, Yaqin Mao
Institute of Computer Science
Communication University of China
No.2 Dingfuzhuang East Street, Chaoyang District,
Beijing, 100024, P.R. China
0086-010-65783396
email: huwf@cuc.edu.cn

***Abstract*—Finite State Machine (FSM) is the most common used technique to create intelligent character behaviors in video games. But conventional FSM technique has many limitations in game development. The main innovation of this paper is the introduction of software component technology to the implementation of FSMs in game development, which modularizes the states and transitions of FSMs completely. Compared with the conventional FSMs, this technique has three advantages. First, high-level and complex intelligent behaviors can be constructed from a set of low-level and simple behavior rapidly. Second, high-level game designing can be decoupled from low-level game AI programming. Third, game characters equipped with this AI system can exhibit more flexibility and adaptability to the changing game environment.**

keywords: component-based hierarchical state machine, finite state machine, video game, artificial intelligence, game AI, software reuse, component.

I. INTRODUCTION

Traditionally, computer games developers use a small set of techniques over and over again in the implementation of artificial intelligence (AI) functionalities in video games, especially Finite State Machine (FSM) [2][10] is the most frequently used one. In video games, FSMs are typically used to model the behavior of computer-controlled game characters, also called no-player characters (NPC), to make NPCs to react to game events seem as intelligent and natural as possible. FSMs consist of a set of states, which represent some kind of actions or behaviors, and a collection of transitions for each state, which specify NPC's reactions to game events.

In fact, FSM is the most popular approach for game AI algorithms for many reasons including: ease of understanding, ease of implementation, efficiency, etc. Learning to use FSMs is relatively easy; developer familiarity helps with rapid prototyping. Ease of implementation also allows rapid prototyping and enables more development iterations. FSMs are lightweight and fast, which is always important in real-time systems. Compared to other control schemes FSMs are easier to test and validate.

However, traditional FSM technique has many disadvantages too. First, FSM approaches require the programmers to anticipate the full range of expected states and transitions between them. These states and transitions, often specified at compile time, cannot be readily modified once a game is built. As a consequence, FSM technique is limited specially by *combinatorial explosions* (As the environment complexity grows, FSM states and transitions sets grows too). Finally, it is very difficult to reuse existing states and transitions due to their context (or application)-specific nature. As a result, game AI programmers have to continually reinvent and recode those states and transitions in different contexts or in different game projects. This needless reinvention causes costly and time consuming development iterations.

The limitation of *combinatorial explosions* was overcome with the creation of Hierarchical State Machines (HSMs) [6][8][11], an extension to traditional FSMs that allows the creation of composite states, which contain, inside themselves, other states and transitions resulting in a hierarchy of FSMs. Using this approach, called generalized transitions, redundant transitions could be prevented.

Another technique called Object Oriented Hierarchical State Machine (OOHSM) [7] introduces the object-oriented programming technique to implement HSM. In [7], the author proposes the use of *inheritance* of OO technique, which allows a new state (machine) to be defined rapidly by reusing the behavior from an abstract state (machine). To a certain extent, this approach improves the productivity of development FSM in video games.

However, OOHSM technique is still not able to eliminate the two disadvantages of FSM above mentioned. First, the states and transitions, created with OO technique, are still very difficult to be reused as a building-block in different contexts. Second, OOHSM is still restricted by its fixed configurations (states hierarchy and related transitions), which leads to low adaptability to the changing environment at run time.

In this paper, we propose a new technique called Component-Based Hierarchical State Machine (CBHSM), which introduces software component technique to the implementation of hierarchical state machine. This technique overcomes the limitations of OOHSM and has three significant advantages:

–Compile Time Composability: At compile time, game

The paper is supported by the "Engineering Planned Project of Communication University of China (XNG0920)".

978-1-4244-8625-0/11/$26.00 ©2011 IEEE

AI programmers can create a new high-level and complex state by composing other prefabricated and simple states and transitions as building-blocks (whose source codes keep untouched).

–Design Time Configurability: CBHSMs are no longer completely fixed at compile time by programmers, and game designers have a chance to configure them at design time according to the game's high-level design.This feature decouples game designing from game AI programming, and a minor change of a game's design no longer needs a recompile.

–Run Time Flexibility: At run time, CBHSMs can be reconfigured as needed. This feature frees CBHSMs from fixed hierarchical structure and greatly improves their flexibility and adaptability to the changing game environment.

In the reminder of this paper, section 2 explores the obstacles on the way toward CBHSM. Section 3 introduces the infrastructure of CBHSM system. In section 4, we elaborate the implementation details of CBHSM system. Section 5 shows the advantages of CBHSM technique. Finally, section 6 points out some potential direction of future research based on this paper.

II. The Obstacles on the Way towards CBHSM

The software technique of *Inheritance reuse* utilized in OOHSM, is often called "white-box reuse" and generally considered to be defective [1].Over *class inheritance*, *object composition*, another common technique for reuse, is favored by "Gang of Four" [3], who advocate that it's the best practice to reuse the functionalities of a class by assembling or composing its object. Figure 1 illustrates an ideal CBHSM making use of *object composition* technique.

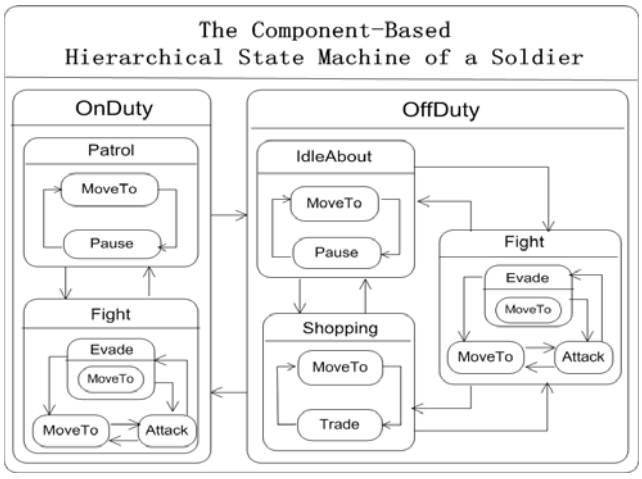

Fig. 1. An Example of CBHSM

In Figure 1, the composite states, such as *OnDuty*, *Patrol*, *Fight*, *Evade*, *OffDuty*, *IdleAbout* and *Shopping* etc, act as states containers. Some other states, such as *MoveTo*, *Pause* and *Attack*, can be composed as building-blocks into different contexts (up level composite states). The *object composition* technique obviates the need to create a new class by inheriting from a state to reuse it. A composite state can only "employ" an instance of the state and delegate operations to the "employee" when receiving a request. We call this ideal HSM "Component-Based".

How to obtain an ideal CBHSM like above? In the following of this section, we show three obstacles on the way towards a CBHSM.

A. One-Way Encapsulation of OO Technique

Class inheritance is straightforward to use, since it's supported directly by the object-oriented programming language. But OO programming languages do not directly support component-based programming. That is, an object normally created by an OO programming language is far from a component unless it is designed carefully to follow some software design rules.

Why is an object normally a non-component? Why the *encapsulation mechanism* of OO doesn't work? According to OO technique, *encapsulation mechanism* conceals the implementation details of an object from its context, shown in Figure 2.

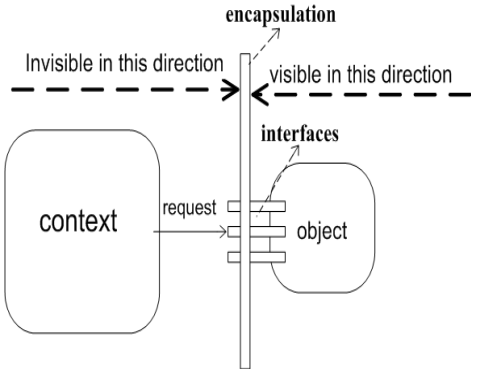

Fig. 2. One-Way Encapsulation of OO Technique

Figure 2 shows that OO technique only ensures a one-way encapsulation. That is, the context is independent on the implementation of its delegation objects. As a consequence, a context object can contains and delegates other objects with compatible interface regardless of their implementation details. However, encapsulation mechanism does not ensure an object is independent on the implementation of its context. Obviously, an object whose implementation is dependent on a special context cannot inhabit in other heterogeneous contexts. In the case of FSM, states created by ordinary OO technique tend to be coupled with their contexts (container state, NPC, or game environment) and cannot be composed into other heterogeneous contexts.

B. The Coupling between Event Senders and Event Consumers

In a game system, game objects, such as, avatar, NPCs, trees, missiles, etc, interact to each other by game events. A game object only acts when an event is produced and sent to it. When a game object receives an event, a transition rule may be

satisfied and then be triggered to change its current behavior (state). In most cases however, game events are simply synchronous events (method calls) which is most straightforward to use. But a synchronous event has a disadvantage that it couples the event senders and the event consumers. In the case of FSM, synchronous event causes a state to be dependent on its context and then tends to be un-reusable.

C. The Coupling between States and Transition

In FSMs, each state has a collection of transitions, which specify the NPC's reactions to various game events. It is a common practice to hardcode the logic of transitions into the corresponding states, often taking the form of a series of statements of "*if-else*", which causes the coupling between a state and its transitions. However, a state interweaved with hardcoded transitions tends to be un-reusable, because the same state in different contexts may have different reactions to the same event.

Detailed techniques to build a CBHSM system are presented in the following sections. To explain this software implementation technique clearly, we use the *Unified Modeling Language* (UML) [5] to illustrate the systems architecture, components and relationship.

III. Asynchronous Event-Driven System - The infrastructure of CBHSMs

As mentioned in section 2.2, to decouple a state and its context, the state should receive asynchronous events. Since most of conventional game engines don't support asynchronous game events directly, we must establish an *Asynchronous Event-Driven System* (AEDS) based on a conventional game engine first.

We learn the ideas from [4] to establish our AEDS. In the implementation of our AEDS, *GEvent* (an abstract class) and *IEventReceiver* (an interface) play the key roles. *GEvent* is the ancestor class, which represents the game events and can be generated, sent, cached and received. The interface of *IEventReceiver* declares the functionality of receiving game events. A component in our AEDS called *EventDispatcher*, which implements the interface of *IEventReceiver*, contains the game events ordered by time that are going to happen in game, and dispatches these events to their destinations in time sequence. Any game objects do not send the events they generate directly to the event destinations, but to the *EventDispatcher* (see Figure 3).

In our implementation of CBHSM system, all game objects in nature interact by events exchange. Any game objects only act or change their behavior because of the event arrival. For an example, when the avatar says hello to a NPC, the avatar, rather than directly calling some method of the NPC, generates and sends a event object (e.g.*GSayHelloEvent*) to the NPC indicating the salutation in that very moment. This event includes all necessary parameters (*time*, *launcher*, and etc). When the NPC receives this event, its state machine reacts to it, and establishes how to change its states (*GSayHello*, *GEscape*, *GFight*, or do nothing).

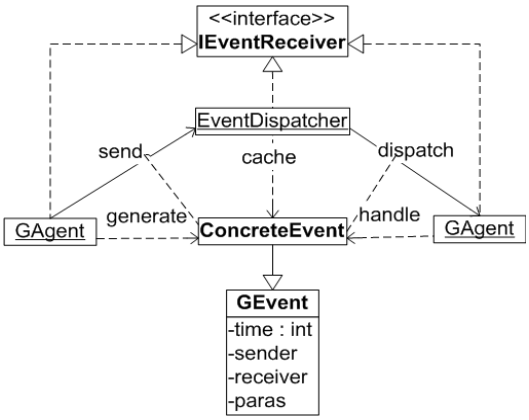

Fig. 3. Asynchronous Event System

Because of AEDS, game agents and their CBHSMs are left only a very narrow interface to interact with the game world, which makes them to be less dependent on their contexts.

IV. The Component-Based Hierarchical State Machine

A. The Architecture of CBHSM System

In our implementation of CBHSM system, there are several key classes, which collaborate with each other and form the system structure. Figure 4 shows their class inheritance relationship, and figure 5 shows their object reference relationship.

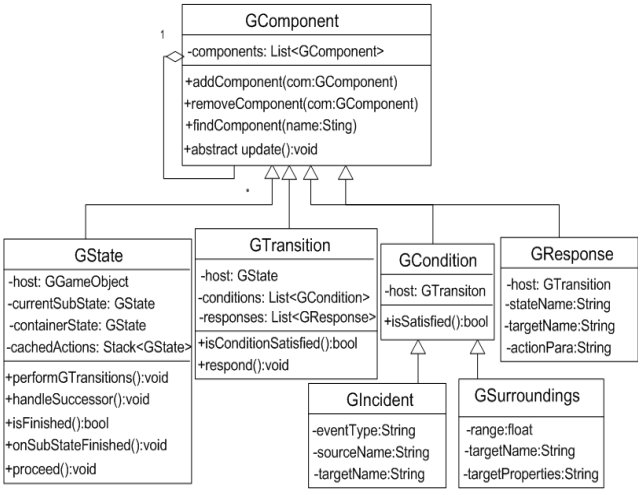

Fig. 4. Class Inheritance Hierarchy of CBHSM

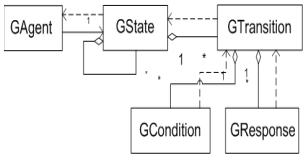

Fig. 5. Object Reference Relationship of CBHSM

GComponent: This class is the ancestor of all other classes in our implementation of CBHSM, which provides the basic functions of any component systems, such as containing,adding, removing and searching other components at run time. Besides, ***GComponent*** has a abstract method *update*, which is called every frame by game engine to update the properties of ***GComponent***.

GState: In our implementation of CBHSM, this class represents a NPC's (***GAgent***) current state and behavior. As a descendant of ***GComponent***, a ***GState*** object (For simplicity, we use lower-cased and italic class name represents an instance of this class below) may contain other *states* and (or) *transitions*. In the hierarchy of CBHSM, a *state* may act as the roles of both *containerState* and *substate* simultaneously.

GTransition: CBHSM technique encapsulates conventional transitions rules into s series of ***GTransition*** objects. A *transition* can only inhabit in a *state*, whose responsibility is to check whether some conditions have been satisfied and then to make responses accordingly. Note that a *transition* works only when its *host state* is just a current active state.

GCondition: This class encapsulates the conditions, which can trigger responses in transition rules. A *condition* can only exist in a *transition* that has a special container to hold them. Each *condition* is responsible to check itself whether it has been satisfied. ***GCondition*** provides an interface named *isConditionSatisfied*, which is must be implemented by all descendents of ***GCondition***. Only when all *conditions* contained in a *transition* are satisfied, this *transition* is triggered.

GResponse: This class encapsulates the reactions (state transition) of a NPC in a *state* to some *conditions*. It specifies the *name* (or *type*) of next *state* and other necessary parameters. A *transition* may contain more than one *responses*, which means that game NPCs can carry out a series of, rather than one, actions to react to some event. It's important to note that the series of *states* triggered by a *transition* are first cached into a stack (a property of ***GState*** named *cachedActions*) hold by the *containerState* of this *transition's host state*. In next round of *update*, the element at the top of *cachedActions* will be pop up and set up to be *currentSubState*.

GIncident: This is a subclass of ***GCondition***, which defines a special trigger condition: the NPC has received a specified ***GEvent*** object.

GSurroundings: This is another subclass of ***GCondition***, which defines another special trigger condition: a particular game object has come within the NPC's range of vision.

B. The Dynamic Behavior of CBHSM System

To understand the dynamic characteristics of CBHSM system, some key code snippets are excerpted from CBHSM System (figure 6). Although the process of algorithm illustrated in figure 6 is quite straightforward, the following points are worth noting.

First, two conditions result in state transitions: (1) External Environment, which is handled by *transition* objects (line 1 to line 3). (2) Intrinsic Termination Conditions, which is differentiated into two situations and handled accordingly: (i) When

```
0: GState::void update(){
1:   foreach(GTransition t in components){
2:     t.update();
3:   }
4:   if( interupted ) return;
5:   handleSuccessor();//desianate successor, if has
6:   if( isFinished() ){
7:     containerState.onSubStateFinished(this);
8:   }else{
9:     if( currentSubState!=null ){
10:      currentSubState.update();
11:    }else{ //no transition and no substate, then
12:      proceed(); //proceed to execute own task
13:    }
14:  }
15: }
16: GState::void onSubStateFinished(GState fs){
17:   GState next = getNextSubState();
18:   if( next != null ){
19:     setCurrentSubState(next);  //state transition
20:   }else{
21:     fs.setActive(false);
22:     setFinished(true);
22:   }
23: }
```

```
24: GState::GState getNextSubState(){
25:   return cachedActions.pop();
26: }
27: GState::addCommand(GResponse com){
28:   pushAction( getComponent(com.name) );
29: }
```

```
30: GTransition::void update(){
31:   if( conditionSatisfied() ){
32:     respond();
33:   }
34: }
35: GTransition::bool isConditionSatisfied(){
36:   foreach(GCondition c in components){
37:     if( !c.isSatisfied() ) return false;
38:   }   return true;
39: }
40: GTransition::void respond(){
41:   if( host.needCaching() ){
42:     host.containerState.pushAction(host);
43:   }
44:   foreach(GResponse r in components){
45:     host.containerState.addCommand(r);
46:   }
47:   host.containerState.onSubStateFinished(host);
48: }
```

Fig. 6. Code Snippets of CBHSM System

a *state* has completed its own task without interruption by external environment, and it exactly knows what kind of *state* is its successor, then it informs its *containerState* to perform this transition. Line 5 in the above code snippets handles this situation. The method of *handleSuccessor* is abstract and any subclasses of ***GState*** may override it if desired. (ii) When a *state* has completed its own task successfully, and it doesn't have any idea of its successor, then it has to leave this decision upward to its *containerState*. Line 6 to line 7 in the above algorithm handle this situation.

Second, when a *state* is interrupted by external environment, before the series of reactive *states* are pushed into the stack (*cachedActions*) of *containerState*, the interrupted *state* should be checked first if it needs to be cached. If necessary, the interrupted *state* is pushed into the stack first. Line 43 to line 48 complete this process. The order is very important, which ensures that a game NPC can continue its interrupted mission after dealing with environmental incidents.

C. The Component Quality of CBHSM

To modularize FSMs thoroughly, the composition interface of ***GComponent*** is just a beginning, CBHSM technique takes the following three steps to decouple *state* from the other parts of CBHSM.

First, as mentioned in section 2.2 and section 3, CBHSM adopts an asynchronous event-driven system, which makes a *state* to interact with game environment through a specified and narrow interface, thus reduces a *state's* dependence on its environment. Second, transition rules in *states* are no longer in the form of "*if-else*" statements, instead are encapsulated into a set of ***GTransition*** objects. As results, *transitions* can be configured into different *states* at game design time, and can be added to or remove from *states* flexibly at run time.

The third step, decoupling a *state* from its *containerState*, is most crucial, which can ensure a *state* to be composed in other complex *states*. Because a *state* may acts the roles of both *containerState* and *substate* simultaneously, clarifying the responsibilities between the two roles in one class definition is

most crucial and most difficult. Figure 7 clarifies the allocation of responsibilities between *containerState* and *substate*.

Allocation	Method Name	Annotation
common responsibilities	*update*	This is the main control flow of this algorithm. The update method of containerState will recursively call that of substate (if exist).
	isFinished	To check if this state has finished its task.
	performGConditions	Dealing with state transitions caused by external factors specified by *transitions*.
	proceed	Executing the intrinsic process of this *state*.
responsibilities of containerState	*onSubStateFinished*	This method is invoked when *currentSubState* has finished. Figure 6 gives its default implementation. When creating those complex and application-specific states, overriding this method is one of the key tasks.
	getCurrentSubState	Obtaining the current active *substate*.
	getNextSubState	Obtaining next active *substate*. By default, it returns the top element of stack (*cachedActions*).
	setCurrentSubState	Setting the parameter to be the *currentSubState*.
responsibilities of subState	*handleSuccessor*	To check if the state has finished, if so, try to designate its successor, if it knows.

Fig. 7. Allocation of Responsibilities Between *ContainerState* and *Substate*

To improve the reusability of a subclass of ***GState***, along with clarified interface between *containerState* and *substate*, another important design criterion should be kept in mind: "*Keeping a State as Simple as Possible*". A *state* should be oblivious to everything but its own special business.

According to the division of labor specified in figure 7, when creating a new composite state class, the game AI programmers need only do two things: (1) overriding the *isFinished* method to specify the new state's end conditions; (2) overriding the *onSubStateFinished* method to process the termination of its *currentSubState*. The game AI programmers can compose prefabricated states into a new state freely without worrying about if the composed states can adapt to the new context. Furthermore, the above newly created composite state, which follows our design specification, becomes a composable building-block again.

V. Advantages of CBHSM Technique

A. Compile Time Composability

At compile time, if a game AI programmer wants to define a complex intelligent behavior that comprises some more fundamental behaviors (*substates*), (s)he can create a new class by composing those basic *states* and *transitions* as building-blocks. In the implementation of a new state class, the programmer only focuses on the top-level behaviors, and leaves low-level and detailed behaviors to its *substates*. Obviously, reusing *states* and *transitions* as components is more flexible and then is more productive. Based on a limited number of simple *states*, a multiple of complex states can be constructed by this approach rapidly. CBHSM technique makes it possible to characterize a game NPC's behavior on a higher level.

B. Design Time Configurability

The configuration of conventional FSM of a NPC usually is hardcoded at compile time by game AI programmers. Thus, FSM programmers get involved in the things relating to the NPC's high-level design that should be handled by game designers. Obviously, it's much more difficult to tweak a game's design if a recompile is needed after every minor change.

The CBHSM technique prevents game programmers from hardcoding every NPC's FSM at compile time and postpones this work to game design time. Then, game designers obtain chances to practice their profession to create character's various behaviors necessary for game plot by configuring their CBHSMs. Thus, the tasks of game AI programming and game designing are decoupled, game programmers and game designers can then focus on their own affairs respectively without worrying about interference with each other.

The configuration of a CBHSM is usually recorded into a file. When a game startups and initializes, game engine will construct the CBHSM by loading and reading that file. With CBHSM technique, game designers can make four kinds of configuration. First, they can add some *states* into a composite *state*. Second, they can specify some *transitions* into a *state* to model the NPC's reactions to specific game events. Third, *transitions* can also be configured with different *conditions* and *responses*.Finally, those public properties defined in the above components can be configured in the file and initialized by the *introspection mechanism* supported by the mainstream OO programming languages (such as Java and C Sharp).

C. Run Time Flexibility

To make NPCs more unpredictable and believable, the NPCs' behaviors should not be determined completely at compilation time, or even at design time, and should be adaptive to the changing game environment at run time [9]. The component technology opens the door for run-time changeable FSMs and greatly improves FSMs' flexibility and adaptability.

According to above analysis of this paper, in our CBHSM system, any *state*, whose design strictly follows the specified interface and design criterion, can be added to or removed from other *states* at run time. So the nested *states* hierarchy can be changed dynamically. Furthermore, *transitions* in a state, conditions and *responses* in a *transition*, can all be changed dynamically. The flexibility of CBHSM provides the possibility for game NPCs to adapt to their environment. Finally, the mechanism of *states* caching, as mentioned in section 4.2, makes a game NPC to have two abilities simultaneously: "Adaptability to changing environment" and "Persistence in fixed tasks".

VI. Future Works

The initial CBHSM system appears as an AI subsystem of ***Pureland*** game engine, which is a component-based game engine developed by the author of this paper. Currently, CBHSM system has been migrated to ***Unity3D*** game engine which is a very popular commercial 3D game engine.

At present, in order to create a complex game character, a large amount of configuration must be done at game design time. A potential direction of future work based on this paper is to explore the strategies of generating the configuration plans according to game environment at run time [12].

VII. Conclusion

Finite state machine is a traditional technique which has both strengths and weaknesses and is widely used in game AI development. This paper makes a great improvement to traditional FSM technology and makes it more reusable and more adaptive.

References

[1] Alan Snyder. Encapsulation and inheritance in object-oriented languages. In Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, pages 38-45, Portland, OR, November 1986. ACM Press.

[2] Thomas, Andy Hunt, "State Machines," IEEE Software, vol. 19, no. 6, pp. 10-12, Nov./Dec. 2002.

[3] E. Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Boston, 1995.

[4] Garcia, I., Molla, R., Barrala, A., GDESK: Game Discrete Event Simulation Kernel, Journal of WSCG, 2004.

[5] G.Booch, J.Rumbaugh, and I.Jacobson. The Unified Modeling Language for Object-Oriented Development, Version1.0, 1996.

[6] GIRAULT, A., BILUNG, L., LEE, E., 1999. Hierarchical finite state machines with multiple concurrency models. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 06 August 1999 Berkeley. California: IEEE Press, 742-760.

[7] MALLUK, WILLIAM; CLUA, E. An Object-Oriented Approach for Hierarchical State Machines, Full Paper Computing Track, in SBGames 2006 Proceedings, ISBN 85-7669-098-5, Recife, PE, Brazil, november 2006.

[8] R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In Proc. of the 26-th International Colloquium on Automata, Languages and Programming, ICALP'99, LNCS 1644, pages 169-178. Springer-Verlag, 1999.

[9] R.M. Young et al., "An architecture for integrating plan-based behavior generation with interactive game environments," Journal of Game Development, vol. 1, 2004.

[10] Wagner, F., Schmuki, R., Wagner, T., Wolstenholme, P., 2006. Modeling software with Finite State Machine: A practical approach, United States: Taylor Francis Group.

[11] Valery Sklyarov, Hierarchical finite-state machines and their use for digital control, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, v.7 n.2, 222-228, June 1999.

[12] Weld, D. An Introduction to Least-Commitment Planning. AI Magazine, 27-61, 1994.