DECLARE-based parallel hierarchical state machine for computer games

Junlin Luo
Computer Games Engineering MSc, Computer Sciences, Newcastle University
Newcastle, Great Britain
J.Luo17@newcastle.ac.uk

Abstract

Finite state machines have an extensive range of applications in the field of games. Using state machines can make the program structure clearer and avoid a lot of logic holes in logically complex systems. Also, using state machines solves the problem of duplicate definitions caused by over defined branching statements. However, as the system's complexity increases, state machines' drawbacks are exposed. At the working level, it cannot be analyzed and corrected based on events or actions that occurred in the past. At the design level, a single state machine causes an "exponential explosion" of definition problems, which is an obstacle for designers.

In this paper, we will review the existing effective linear temporal logic-based process control schemes, parallel hierarchical state machine implementation schemes, and the rationale for communication system implementation. Finally, I will give an outlook on the project, comparing the results before and after the use of process control.

Keywords: Parallel hierarchical state machine;Communication system;Linear temporal logic;DECLARE;

1 Introduction

1.1 Motivation

State machines are widely used in games, mostly together with behavior trees, in the game's animation controllers. Usually, we always combine state machines and behavior trees to build game logic together. State machines are not only used in game animations, but also in software architectures. A computer system can be built in a state machine architecture.

However, as the number of states in a state machine increases, it becomes more and more complex, and the large number of states in the state machine makes it very large and clumsy. From the design side, we will design similar modules over and over again. From a practical point of view, when we have a design blueprint and implement concrete code according to it, the large number of trivial states and transition conditions is undoubtedly frustrating for writers. If we could divide the state machine in such a way that parts of it could be reused, this would greatly reduce the work time of designers and writers.

State machines also face the problem that if all states are written to a state machine, then this will create multiple transition conditions. As the number of states increases, we need to add more conditions for each additional state, which is an "exponential explosion" of increments that designers do not want. This motivated me to design a state machine with a parallel effect. This would effectively reduce the need to define transition conditions.

At the same time, a state machine cannot track past events or states. It simply transitions between a limited number of states according to a set of transition conditions, and does not have the ability to analyze and self-adjust. This motivates me to extend the state machine by imposing time constraints.

1.2 State Machine

Finite State Machines are an effective tool for modeling the behavior of solid objects. A state machine describes the process of transitioning an entity from one behavior to another (sometimes keeping the state unchanged). The modeling process consists of the following three main elements:

- The current state the entity is in.
- The detection conditions for the state transition to occur.
- The state to which the current entity will be transformed.

Usually we use a graph to represent a state machine in which vertices represent states and edges represent transition conditions between states[3]. "When the transition condition f is true, the entity will transition from the current state to the target state", such a form of transition is considered by much literature as a rather natural way to describe the behavior of complex systems[6].

Hierarchical state machines are an extension of finite state machines, using different levels to partition the state machine. The main purpose of this division is to allow some common parts of the state machine to be reused. It is conceivable that a hierarchical state machine is a state machine composed of many finite state machines, some of which can be reused by different entities.

Parallel state machines are also an extension of finite state machines. Due to the nature of state machines, if all states are modeled by just one state machine, there will be an "exponential explosion" of state transition conditions that need to be defined. The main reason for this result is that a transition condition is required between two transformable conditions. Parallel state machines are composed of no less than two finite state machines, ensuring that they run in parallel to reduce the number of transition conditions for the designer's convenience.

1.3 Communication system integral

Communication system integral (CCS) is a process algorithm that can simulate the previous communication of a system. The main approach is to describe the parallel behavior of the system from an abstract level using a small number of basic primitives. In contrast to general programs, there are no variables and no variable types in such an approach. The main purpose of using CCS is to prove the correctness of the design before the system is built.

Such a production method has a wide range of applications in other fields, for example in construction engineering[5]. Every system has data inputs and outputs, and CCS expresses the process of system communication in algebraic form through a variety of different inputs and outputs. There is a set of symbols "-" (pronounced bar) that are commonly used to express inputs and outputs. For example, if there is a buffer that is only one grid size (that means it can only hold one thing at a time), we can express it in CCS semantics as :

$$B = in.o\bar{u}t.B$$

Because for Buffer, only two actions are accepted, one is to put in the content and the other is to output the content. A similar coffee machine can be expressed as follows:

$$CM = coin.co\bar{f}fee.CM$$

Not only for a single system, but multiple systems can communicate with each other through "Relabelling". Imagine we have a coffee vending machine, and now I have a milk vending machine, can we abstract the two machines by renaming them, since they are both essentially coins that give goods.

1.4 Linear temporal logic

Linear temporal logic (LTL) is a modal temporal logic. It extends time infinitely, modeling them as sequences of states that express the progression of the system forward in time. Linear temporal logic is an encoding of the future path, which contains a number of propositions. These propositions can be used as an expression of constraints on the development of the system and can be used to verify that the development of the system is on the development path expected by the designer[7].

First of all, LTL is a modal logic. One of the important features is to have both propositions and actions in the syntax of the statement. For example, the soup is not ready now, and when the alarm clock goes off, the soup is ready. "The alarm clock goes off" is the action that occurs, and when this action occurs, the proposition "the soup is ready" is true. The smallest elements of linear temporal logic are atomic propositions, and there are many combinations between them. Due to the introduction of time, four expressions describing time are introduced. The capital letters G, F, X, and U stand for "globally", "in the future", "next time", and "until" . G means that the proposition will be true for the rest of time, F means that the proposition will be true at some time in the future. X means that the proposition will be true from the next moment. u is a double operator. For example, AUB means that A will remain true until B is true. Using these four expressions, we can express some time-dependent constraints. If we want to express that the traffic signal will always flash in the sequence of "red, yellow, green, red, yellow, green...", we can use the following way:

$$[]\,((gr \bigcup ye) \bigvee (ye \bigcup re) \bigvee (re \bigcup gr))$$

1.5 DECLARE

DECLARE was developed as a constraint-based system and uses a declarative language based on temporal logic to develop and execute process models[9].

One problem with existing process modeling languages is that they can allow designers to over-define the execution process. DECLARE utilizes a declarative approach instead of an imperative one, and uses constraint definitions instead of positive definitions. Allowing it to support loosely structured process models that use many constraints but essentially provide greater flexibility for the system. From the design side, it also allows the system to be designed to focus more on structure rather than specific process control.

DECLARE provides some constraint templates. For example, the "Response" template stands for "Whenever A happens, B should happen after A". Using these templates, we can build all the constraints in DECLARE for validation, similar to building a state machine. Once the models are created,DECLAR will first review the defined models to check for unexpected cases such as invalid constraints. Finally, DECLARE will start its service by accepting information from other systems that need to be validated, and after DECLARE has processed the validation information, it will send the results back to the user.

1.6 Major Problem

The current use of state machines in games faces the following three problems:

- The state machine module cannot be reused.
- State machines require a large number of transition conditions to be defined.
- The state machine cannot analyze past events.

In the design process, state machine module reuse is a very common way to simplify the design. For example, various NPCs in the game have many sub-state machines with the same structure among them. The purpose of state machine module reuse is achieved by using a hierarchical state machine to mark the state machine modules that need to be reused.

As the complexity of the state machine increases, it is assumed that each new state added can be transformed with the previous state, which means that the number of condition transitions added subsequently will become more and more troublesome for the designer. By using parallel state machines and splitting the state machine into parallel small state machines, the number of transition conditions can be effectively reduced, making the structure of the state machine clearer and the designer can design more conveniently.

Due to the nature of state machine, the state machine always runs according to the transition conditions that have been defined, and it cannot analyze the past events. However, in practice, there are a lot of cases where the events need to be analyzed and the state of the state machine needs to be corrected. This also leads to over-definition of the process by the designer. Since the designer has to specify the flow of the system in great detail in order to ensure that the system runs correctly. This is extremely difficult in some cases, e.g. when using probabilistic expressions of transition conditions, the uncertainty of the transitions leads to the designer having to deal with all possible combinations of states

2 Related Work

2.1 Parallel Hierarchical State Machine

For the extension of state machines in computer games, theoretical foundations and implementations have been available in previous literature as well as in realistic production tools.For example, sub-state machines can be defined in unity by grouping them, which is one implementation of hierarchical state machines. There is no mention of parallel state machines in the official unity manual, and the common implementation is to combine a hierarchical state machine with a behavior tree, adding the tasks that need to be paralleled in the behavior tree. The problem is that the designer may not be able to abstract the common state of parallel tasks well. For example, if an AI agent is dancing while eating a snack and doing many other things that are not logically related, it is difficult to assign these parallel tasks to the corresponding common state. If dancing and snacking can be classified as playful, what if the AI is still talking on the phone? It is not appropriate to classify talking on the phone as playful

The state machine has amnesia, amnesia, and insomnia. Amnesia is the inability to consider multiple parts of the progress at the same time. Amnesia is the inability to perceive sub-expressions. Amnesia means that useless state is not unloaded from memory[8].

An effective way to address amnesia is to add additional states[10].The process of doing so leads to an over-definition of states. Splitting the state machine into sub-state machines avoids defining super states and also avoids defining a large number of transition conditions[1].

For designers, the use of thumbnails can be used to express the hierarchical hierarchy of state machines, and the use of dashed lines to split from the state-chart can be used to express the parallelism of state machines[6].

However, for a state machine that actually appears in the game, this parallelism does not actually exist, because the seemingly parallel two state machines are also done by sequential updates, which means that they actually have a running order as well. From the practice of state machines, we also want to allow designers to follow the drawing of the design drawings directly in the state machine. So in a sense parallelism can also be classified as a kind of hierarchy, splitting a task that needs to be parallel into two small sub-state machines.

2.2 Communication Systems And State Machines

Since there are different entities in the system, information needs to be passed between the state machine and the state machine, such as when a player dies another player should know. This requires a communication system to act as a bridge for information transmission between the state machine and the entire game. Using CCS algebra to express the system can carry out state transition check to realize the completeness of the system before the system.[5] Process algebra would be a very advantageous mathematical tool. Process algebra forms a framework for formal reasoning about processes and data, which can be used to detect undesirable properties and formally derive desirable properties for system specifications[4] CCS is essentially a labelled transition system that expresses the state transitions that occur when the system is running in a manner similar to a state machine graph. Communication between state machines will be taken over by a system. Each module provides input and output information. CCS gives us an abstract ability to visualize the communication exchange process of the system.

As in previous studies, I will first verify the communication process between the state machine and the event system using process algebra. If one of the state machines and the event system are viewed as a whole, the system will be expressed as:

$$SYS = (SM[out- > com]||ES[in- > com]) \setminus com$$

The event system and the state machine are in parallel, and the state machine sends the events that occur to the event system. The event system forwards it to the object that is listening to the event for further processing.

2.3 Full Support for Loosely-Structured Processes

Until now, we have found that state machines can express logical equations such as "or", "and", "equal", but not something like "This is due to the lack of predicates and quantifiers[2]. The expression of LTL compensates for this lack to some extent, for example, G acts as a full The role of quantifiers is to express the proposition as being true from now on.

The previous state machine, whether it is a hierarchical state machine or a concurrent state machine, cannot analyze what happened in the past. Even if we have a state machine like "Push down automata" with some "memory", it just records the parent state. If we need to analyze the past, first we need to record all the events through a system, and then use an analysis system based on sequential logic analysis.The DECLARE system offers the possibility of analysis, which is based on sequential logic and checks the entire system by means of verification constraints.[9] Applying DECLARE's analysis system to the game state machine will give the state machine the ability to analyze events and correct itself. Amendment requests can be divided into two categories, those that are non-mandatory to accept, and those that must be accepted.Not only can the NPC's heavy plan be realized, but also reasonable suggestions can be made to the user.

A parameter in DECLARE can be either a proposition or an event. If we want to use the game as the object of validation of the constraint, we will record the events of the game through the event system and then pass the events (and some important parameters) together to the DECLARE system for validation.

3 Project approach

Based on the narrative of previous work, I envisioned a game case. The simulated game consists of three main characters, the player, the player's pet, and the monster. The entire game process will start with the player entering the battle field. The player's pet will fight together with the player, and the pet will have three different strategy modes for the player to choose from.

The entire game will go through four main stages, starting with the initial stage when entering the field, followed by the normal battle stage, where the player and pet will face powerful enemies together. When the monster's blood level falls below thirty percent, the monster will jump to the platform and summon a dragon. Next, the monster will land in the battle area together with the dragon. When landing, the player can only attack the dragon can not attack the monster, when the player successfully killed the dragon, the monster will fall from the back of the dragon. Eventually, the player kills the monster game over. The code for the whole game engine will come from the CSC8503 base code provided by Rich.

To implement this case, I will first model the game state using process algebra to ensure that the game state is feasible and reliable. I will extend the game engine code with state machine related content. The hierarchy and concurrency will be implemented mainly by way of arrays of vectors. I will have a separate state machine on each character and also two external state machines. One state machine is the event system state machine for event messaging, and the other is the debugging state machine for verifying LTL. The purpose of the debugging state machine is to show that LTL expressions can be expressed in a state machine, but the internal structure of the state machine can be very complex and a significant burden to the developer.

Eventually I will make two control groups, one with and without LTL constraints, and the other with state machine validation constraints and with LTL validation constraints.

4 Project Plan

I have now successfully extended the parallel hierarchical state machine and the event passing system into the previous state machine. The general functionality of the game has been implemented. The player is now free to choose targets around him and give attack commands to attack them. For pets, the three main strategy modes work well, and players can switch between them on their own. The part involving LTL time constraints is not yet complete, but the debugging state machine can already simulate some uncomplicated constraints.

The next step will be to verify the LTL constraints on the state machine at this stage, which I think will take a few weeks on this part.

References

[1] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. 1999. Communicating Hierarchical State Machines. *LNCS* 1644. https://doi.org/10.1007/3-540-48523-6_14

[2] Susanna S Epp. 2010. *Discrete mathematics with applications.* Cengage learning.

[3] AB Ferrentino. 1977. State machines and their semantics in software engineering. (1977).

[4] Wan Fokkink. 1999. *Introduction to process algebra.* springer science & Business Media.

[5] Roberto Gorrieri and Cristian Versari. 2015. *Introduction to concurrency theory: transition systems and CCS.* Springer.

[6] David Harel. 1987. Statecharts: A visual formalism for complex systems. *Science of computer programming* 8, 3 (1987), 231–274.

[7] Jesper G Henriksen and PS Thiagarajan. 1999. Dynamic linear time temporal logic. *Annals of Pure and Applied logic* 96, 1-3 (1999), 187–207.

[8] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. 2007. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems.* 155–164.

[9] Maja Pesic, Helen Schonenberg, and Wil M.P. van der Aalst. 2007. DECLARE: Full Support for Loosely-Structured Processes. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007).* 287–287. https://doi.org/10.1109/EDOC.2007.14

[10] Luis Quesada, Fernando Berzal, and Francisco J Cortijo. 2012. Parallel Finite State Machines for Very Fast Distributable Regular Expression Matching.. In *ICSOFT.* 105–110.