Event-driven parallel hierarchical state machines for video games and the re-planning in running time

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

Abstract

Video games are modelled as a complex discrete-event system, usually described via DFA as both compact and expressive. However, they cannot express many complicated scenarios, as they suffer from *insomnia* and *amnesia*.

We investigate these two problems and propose a scheme for building two new types of DFAs: *parallel hierarchical state machine* and *debug state machine*. As graphs such as debug state machines can also be used to detect deadlocks in a game, we propose to solve the deadlock problem and move the DFAs into a desired live state. This approach will use the re-planning approach via A*, which is fairly common in current literature.

On this basis, we evaluate the effectiveness of both the DFAs above and the re-planning algorithm. We compare the game's performance with and without the two new DFAs and the re-planning algorithm. Consequently, the parallel hierarchical state machine effectively reduces the sum of states and transitions, and debug state machines allow the DFA to check earlier partial matches. The game finally enters the expected state using re-planning algorithms. It concludes that the two new DFAs and re-planning algorithms can effectively mitigate the two diseases.

Keywords: Video game, complex discrete-event system, parallel hierarchical state machine, re-planning, A*

1 Introduction

Video games are complex interactive systems involving a large number of dynamic behaviours described via states and events: Because the exchange of information in an interactive system can always be abstracted as "*when an event E occurs (activity label), the system changes from state A to state B if the transition condition C is true (pre-condition)*"[9]. E.g., when a collision occurs, the airbag in the car will be ejected if the impact force is more significant than a certain threshold.

Nowadays, most video games employ AI technology [11]. Nevertheless, video games are real-time compared to other AI scenarios as players generally demand the frame rate for the game, which means that video games require that the processing speed of the AI must be fast enough. There are Character AI, Meta AI and Navigation AI in the games. Character AI is the "brains" of the NPCs in the game, controlling their behaviour; Meta AI controls the process of the game; Navigation AI is for path-finding. In this paper, we will deal with the first two categories. Identifying, making choices, and acting is the flow of AI in a game. State machines have been trendy in making choices from the 1990s until now. State machines mediated by discrete events are ideally suited for expressing certain parts of the AI in games. The reason why events are discrete is that more than one event cannot occur in the game at the same time.

State machines generally exploited in video games are DFAs. A DFA is a specific type of labelled transition system, where special nodes are marked as either starting points or acceptance/desired states. These systems can be equivalently represented graphically (*graphs*), via algebraic rules (*process algebra*), or using logic formalisms (*LTS* and *DFA*). Process algebra, with its straightforward syntax and expressive semantics, is the language for describing LTS, while DFA can be compactly represented as REGEXes. Process algebra is more convenient than traditional state machines and REGEXes, as they provide process composition and synchronization solutions. A use case for this is given by complex video games, where multiple agents interact and reciprocally wait for incoming events. Despite these requirements being crucial in modern video games, the usual AI for this is only exploiting traditional DFAs, which cannot characterize waiting conditions and synchronizations between processes as LTS and process algebra. Therefore, we will use process algebra to describe more complicated state machines.

Still, a complex system cannot be efficiently expressed by a traditional DFA/LTS. Moreover, state machines are always loaded with states and transitions that are not currently used (*insomnia*) and always ignore those early matching conditions (*amnesia*) [1]. These errors lead to very well-known bugs in video games.

We found that in the current game field, some solutions are proposed in the literature for *insomnia*, but nothing related to *amnesia*, so we propose a solution to build two new types of state machines. We also found that using only the new state machine still leads to the problem of the game entering deadlock. As Alur et al. [1999] have concluded from several thematic analyses, debugging video games and AI is hard, and we have also found it very difficult to debug new state machines. To address this problem, we propose a re-planning algorithm.

Figure 1. Stuck NPC Charles in Red Dead Redemption 2

We think those game designers will be interested in this series of solutions because these problems currently plague certain game masterpieces. E.g., there is a bug in Red Dead Redemption 2 [13] where the NPC Charles will order the player to bypass an aggressive bear. However, the player can kill it in some way (the usual game flow is not allowed to kill it), which will eventually lead to Charles being stuck in place and the game progressing to a deadlock [7].

Another example of a similar bug exists in World of Warcraft [3]. The third boss of the Grimbalto dungeon will summon a dragon and enter invincibility when its health is below 30 per cent, and when the dragon dies, the boss will become attackable. However, players can kill the dragon when it appears, causing the BOSS to remain in an invincible state indefinitely [4].

Nevertheless, not all bugs are harmful. E.g., In GTA5's[12] Heist mission, four players are asked to travel to a mountain top after looting the Pacific Standard Public Deposit Bank. Once at the top of the mountain, players must ditch their motorbikes and open their parachutes to glide to an escape craft. During the actual playthrough, many players did not follow the intended parachuting process but instead headed straight for the escape craft. By taking advantage of this bug, players can reduce the time to the escape craft.

The reality of these problems in games motivates us to study them. In this paper, we focus on two diseases of state

Figure 2. The immortal boss in World Of Warcraft.

Figure 3. Robber who abandoned motorbike in Grand Theft Auto 5.

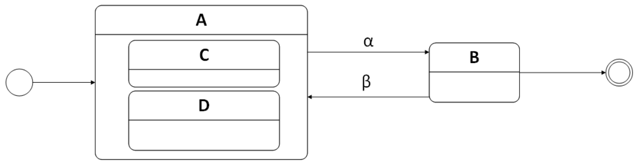

Figure 4. Hierarchical State Machine (A) and Parallel State Machines (C and D) by Harel

machines and how to automatically re-plan the game to our expected state when it is in a "bad state".Our new state machine and re-planning algorithm can help the game to alleviate the pain of disease and avoid the generation of bugs in the running state. We have done simulations on it. The simulation results show that our scheme improves the game's win rate and solves the game's deadlock problem.

2 Background and related work

2.1 Insomnia

A system is affected by *insomnia* when it loads useless states and transition conditions. There have been several studies that have offered constructive comments, and some that are even complete theories. As traditional state charts cannot represent hierarchies [9], we cannot reuse those repeated states and transition conditions, which leads to redundancy in the system. The duplicated parts of the state machine are loaded into memory, indirectly aggravating the insomnia of the state machine. Harel [1987] proposes Hierarchical State Machines (HSM) and Parallel State Machines (PSM) for solving the problem. HSMs reduce the system's redundancy and provide a level-wise description of states, which are more natural and human-friendly when the system is large and complex. The pros of HSM motivate the importance of exploiting HSM for complex systems. To implement HSMs, the same author proposes a refined design idea. Figure 4 describes a state A, which can be refined into C and D.

Alur et al. [1999] believe that we can achieve a more straightforward design through a granular approach and

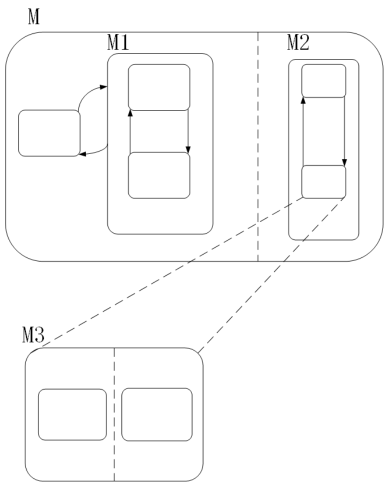

Figure 5. Width and Depth in PHSM

that it is possible to look at the entire state machine at different levels, making the design process step-by-step. He also argues that the modularity of state machines can effectively reduce state reuse. For this, he merged HSMs and PSMs into Parallel Hierarchical State Machines (PHSM, see Figure 5). Ultimately he also proposes two indexes for eliciting the representation's compactness of PHSMs when compared to other graph representations:

- Width: the maximum number of components in the product nodes.
- Depth: the length of the longest path in the graph.

His use of width and depth as evaluation indexes inspired us to choose to use *size* as the final evaluation index. $size = |V| + |E|$ where

- V denotes all vertices (states) in the state machine.
- E denotes all edges in the state machine.

Harel [1987] also mentions the exponential growth of states and transitions since traditional state graphs cannot express state machines in parallel. This is exemplified in Figure 7: there are two state machines, A and B, with $|V_A| = 3$ and $|V_B| = 4$ states. The former comes with $|E_A| = 3$ edges while the latter comes with $|E_B| = 4$ edges. We must combine the two state machines without parallel state machines, which means we must combine every state in A with every state in B. Here we could use the Multiplication Principle in Permutation and Combination. There would end up being $|V_A| \cdot |V_B| = 12$ states and $|E_1||V_2| + |E_2||V_1| = 24$ edges. However, if we use a parallel state machine, we only need to list the states in a and b separately. The final state machine will have $|V_A| + |V_B| = 7$ states and $|E_1| + |E_2| = 7$ edges. See Figure 6 and Figure 7. A multiplicative relationship like $|V_A| \cdot |V_B| + |E_1||V_2| + |E_2||V_1|$ is called an exponential explosion in the state machine.

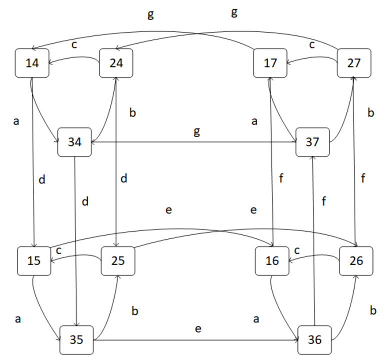

Figure 6. An example for $A \times B$ without PSM

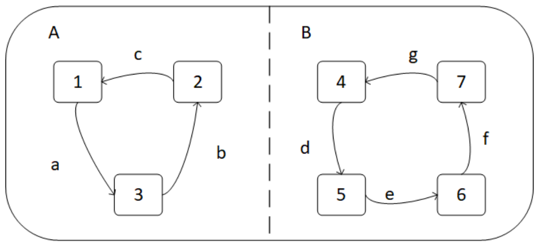

Figure 7. An example for $A \times B$ with PSM

Alur et al. [1999] emphasise atomised actions as a constituent element of parallel systems, affirming the outstanding ability of state machines in this respect. In response to the reuse of state machines, they argue that *process algebra* should be a tool to express parallel systems using CCS (acronym of *Calculus of Communicating Systems*). As atomised, indivisible activities are needed to describe the whole system, a state machine would be a suitable choice. Essentially, a state machine is a finite LABELLED TRANSITION SYSTEM (LTS), expressed as a *triple* relation. Here we need to state some definitions:

- Q is a finite non-empty set of all *states*.
- A is a finite set of *actions*.
- $\{\tau\}$ is the set of invisible *actions*.
- $\rightarrow \subseteq Q \times A \times Q$ is a finite *transition relation*, where $(q, a, q') \in to$ describes that we can reach $q' \in Q$ from $q \in Q$ via action $a \in A$.

Based on the above definitions, if we take a simple game NPC vendor as an example in Figure 8, the delivery system can be defined as a $LTS = (Q, A, \rightarrow)$ where $Q := \{q_1, q_2\}$, $A := \{money, boosts\}$, and $\rightarrow := \{(q_1, money, q_2), (q_2, boosts, q_2)\}$. Any LTS can be equivalently expressed as a system of process algebra formulæ:

$$\begin{cases} q_1 := money.q_2 \\ q_2 := \overline{boosts}.q_1 \end{cases}$$

If we apply the behavioural semantics associated with this system, we then obtain back the graph in Figure 8, thus

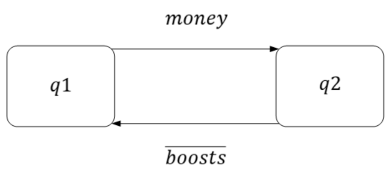

Figure 8. A vendor model in LTS

showing that both representations are equivalent. Furthermore, process algebra allows us to express both parallel state machines via the | operator, so that the passive state in Figure 13 can be expressed as Attack|Move, where Attack := $\beta.\beta'$.Attack and Move := $\gamma.\gamma'$.Move. As we will see in greater detail in §3, such algebra also allows synchronizing actions between state machines via the ν operator. Further details and discussions are postponed to that section due to the lack of space.

We are now ready to draw some final considerations. Harel [1987] and Alur et al. [1999] show us the importance of introducing a new type of state machine in terms of both parallelism and hierarchy of state machines. Their study provides the theoretical basis for implementing parallel hierarchical state machines in this paper. Alur et al. [1999] introduce the communication system as a whole, which becomes a valuable tool for proving system parallelism in this paper. In summary, hierarchical and parallel state machines can alleviate insomnia by synchronising repetitive states and transitions. Process algebra can alleviate insomnia by exploiting behavioural semantics, as states never reachable from the initial configuration are never converted into an LTS. Furthermore, the hierarchical and parallel state machines help us reduce system redundancy via reuse.

2.2 Amnesia

A system is affected by *amnesia* when it ignores partially established parts of the transition condition. The relevant literature with application to the field of video games is scarce. Game designers always try to ensure that the game follows the intended behaviour, so we must detect if the game is on the right track. These detections can be defined *forward* and *backwards*. The *forward* detection needs first to enumerate all scenarios where the game is off track. The *backward* definition requires the satisfaction of given constraints. "If one of the constraints is not satisfied, then the game goes off track" can be written as

$$\exists p \in P, \neg p \rightarrow q. \tag{1}$$

such that p is a constrain and q="the game goes off track", where P is the set of all constrains. The contrapositive of proposition 1 is

$$\forall p \in P, \neg q \rightarrow p. \tag{2}$$

With equation 2 we want to show that the *backward* definition intends to find a statement that generalises all constraints. As the enumeration may lead to a state explosion, as noted in Figure 6, it is better to follow the backward approach. *Conformance checking* is used to check the compliance of a set of temporally ordered events σ (i.e., *trace*) generated by a system (e.g., a state machine) to a temporal behavioural rule φ. For example, in a game where we want to avoid the player dying and his pet continuing to fight, we may specify that when the player dies, the pet must die immediately. When a system violates a constraint, we call such behaviour *deviant*; a constraint is a *process modal*, and Linear-time Temporal Logic for Finite traces (LTL$_f$) is a compact tool for describing such models. Kröger and Merz [2008] define such logic for the first time, stating that temporal logic is composed of terms (atomic propositions p), classical boolean operators, and temporal operators (in bold) as in the following syntax:

$$\varphi := p \mid \neg\varphi \mid \varphi \vee \varphi' \mid \varphi \wedge \varphi' \mid \varphi \rightarrow \varphi' \mid \mathbf{G}\varphi \mid \mathbf{F}\varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi'$$

where ne**X**t ($\mathbf{X}\varphi$) denotes that the condition φ should occur from the next state, **G**lobally ($\mathbf{G}\varphi$) denotes that the condition has to hold on the entire subsequent path, **F**uture ($\mathbf{F}\varphi$) denotes that the condition should occur somewhere on the subsequent path, and **U**ntil as $\varphi\mathbf{U}\varphi'$ denotes that φ has to hold at least until φ' becomes true, either at the current or a future state. Linear temporal logic is composed of propositions over discrete time, which fits well with our definition of a discrete event system.

LTL$_f$ can guarantee that the sequence of events generated by our game satisfy some quality constraints. Since we have a sequence of events in the game, we can perform *Conformance checking* based on this sequence. The sequence of events trace σ comprises a finite number of discrete events. An event σ_n is a pair $\langle A, p\rangle$, where A is the activity label, and p is the associated *payload*. The *payloads* are a mapping of key-value pairs representing the events' parameters. However, because of amnesia, we need some mechanism temporarily store matching conditions. Bergami et al. [2021] mention that any LTL$_f$ formula combined with *payloads* can always be expressed as a state machine. These state machines naturally keep track of the current "matched state" for later consistency detection. In summary, using LTL$_f$ for generating game state machine might solve the amnesia problem.

Bergami et al. [2021] also provide three steps for transforming process modeling into LTL$_f$:

- Express process model in LTL$_f$
- Provide data propositions
- Combine all propositions

Again using the previous example referring to the player and the pet, we can express it as the following LTL$_f$ formula synchronizing the player with the pet:

$$\mathbf{G}(\mathit{player\ die} \rightarrow \mathbf{X}\mathit{pet\ die})$$

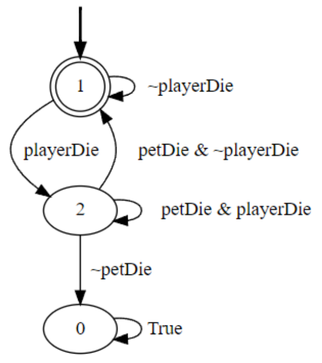

Figure 9. Scenario B from my dissertation's use case.

Using the aforementioned three steps, such a formula can be also represented as the graph in Figure 9. We encourage the reader to refer to [6] for propositions and logical operators, as well as tools describing discrete mathematical structures.

2.3 Re-planning algorithm

Cashmore et al. [2019] categorize actions as either *interruptible* or *non-interruptible*, and proposed corresponding re-planning approaches. They point out that it is not appropriate to classify all behaviours as non-interruptible ones, which may cause *deadlock*. E.g., if a character is in the path-finding process and we need to re-plan the character due to some undesirable condition, this could be recalling the character. However, suppose we classify the path-finding process as uninterruptible: the character must reach the final destination before being recalled. If the undesirable state is that the character is stuck on the path, then the character will never reach the end of the path-finding process and cannot be recalled later. Bergami et al. [2021] describe in detail the process of the final transformation of LTL_f into a DFA using *Planning Domain Definition Language* (PDDL) and provided the method of re-planning in conjunction with the A* algorithm for correcting undesired process behaviour. PDDL can express a (re-)planning problem $p = (I, G, P_D)$, where I is the initialization state of the system, G is the goal configuration, and P_D is the domain we are concerned with that needs (re-)planning. Ω is the set of a series of (re-)planning actions, which means that when we have executed all of Ω, the system will satisfy our pre-set G. We use a to represent a single sub-action, where $a \in \Omega$. a can be defined as $\langle Para_a, Pre_a, Eff_a\rangle$,where:

- $Para_a$ denotes the input parameter.
- Pre_a denotes the precondition that determines which a will be executed.
- Eff_a denotes the effect of the re-planning action.

The scenario in Bergami et al. [2021]'s research is slightly different from the game scenarios mentioned in this paper. The former is static, as it does not involve real-time detection

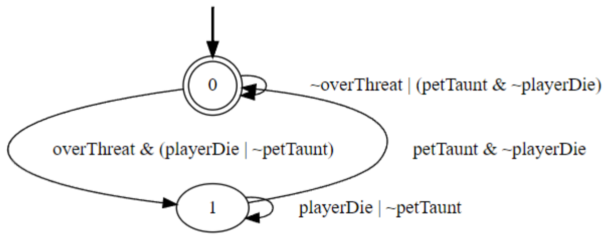
0
~overThreat | (petTaunt & ~playerDie)
overThreat & (playerDie | ~petTaunt)
petTaunt & ~playerDie
1
playerDie | ~petTaunt

Figure 10. Scenario A

of whether the system has entered an undesirable state while the actual game environment is dynamic. To illustrate this difference, we will show another game scenario in the next section: to give the player a better gaming experience, we desire that once the player over-threat, the pet taunts before the player dies (Figure 10), which can be represented in LTL_f as follows:

$$\mathbf{G}(player\ overthreat \rightarrow (\mathbf{F}(pet\ taunt \wedge \neg player\ die))$$

Deadlocks in the game can be easily detected when state machines reach a state having no outgoing edges. However, in some other scenarios, all states have outgoing edges (Figure 10): in such cases, the only way to define a "bad" state is to specify state 1 as a bad state by hand.

Goyal et al. [2014] analyse the differences and connections between A* and Dijkstra's path-finding algorithms. We find that the most significant difference between A* and Dijkstra's algorithms is the heuristic for pruning the state space. As A* exploits heuristic for path traversing, we might exploit states' expectation values as a heuristic for our re-planning algorithm when and exploit A* for re-planning to escape the bad or deadlock states reach the desired state configuration. The cost function exploited by A* is expressed as $f(n) = g(n) + h(n)$, where:

- $f(n)$ is the combined priority of node n. When we choose the next node to traverse, we always pick the node with the highest combined priority.
- $g(n)$ is the cost of node n from the starting point.
- $h(n)$ is the expected cost of node n from the endpoint, which is the heuristic function of the A* algorithm.

Introduction to conformance checking reveals that we can use linear logic to express constraints to alleviate the amnesia of state machines. It also shows that LTL_f can always be expressed as a constratint automata , showing us the possibilities of implementation, and providing detailed implementation steps[2].

We are now ready to draw some final considerations. The description of interruptible and non-interruptible behaviour shows us that modelling behaviour always as non-interruptible is undesirable in some cases. Here, since our state machine is a detection state machine, it can be modelled as an interruptible state machine [5]. Bergami et al. [2021] combine the re-planning approach with the A* algorithm, thus providing a solution that might be transferred to videogames. At the same time, the description of the PDDL jointly with the behavioural semantics associated to process algebra sheds light on how we should implement the game's event system.

3 Design and implementation

We designed and implemented a small game demo to simulate the general situation in WOW described in Figure 2. We sketched a game scenario consisting of four entities: a player teamed with a pet, monsters and dragons as antagonists. This is similar to the flow of the game in the dungeons. When the player attacks the monster, the pet assists the player based on its own AI logic. The pet has three strategies:

- Assist mode: the pet imitates the player's actions and becomes a shadow of the player.
- Protect mode: the pet will fight back when the player is in a dangerous situation.
- Passive mode: there will be no response unless the player gives a command.

When the monster's health falls below 30%, it will become immortal and start summoning a dragon. The monster will then move to a fixed location with the dragon. Eventually, after the player has killed the dragon, the monster will leave the immortal state and become attackable by the player. The player wins once the player has killed the monster and the game reaches a desirable state. The game reaches an undesirable state when the player loses.

3.1 Game Constraint

We have designed three constraints for the game:

$$\mathbf{G}(player\ overthreat \longrightarrow \mathbf{X}pet\ taunt) \tag{3}$$

$$\mathbf{G}(player\ die \longrightarrow \mathbf{X}pet\ die) \tag{4}$$

$$\mathbf{G}(summon\ dragon \longrightarrow (\mathbf{F}dragon\ die \wedge (\neg dragon\ die \mathbf{U} arrival))) \tag{5}$$

The advantage of scenario A (Figure 10) is that the game's correctness rate can be an index to evaluate this scenario, as killing the monster is the game's correctness condition. Theoretically, the game's correctness rate would increase if the pet could be guaranteed to protect its owner effectively. Scenario A, however, never has a deadlock condition, which means re-planning is useless for scenario A. Scenario A makes a no-limit commitment to time. A pet satisfies the constraint as long as it taunts the player over-threat, and that time is acceptable no matter how late.

We are looking at Scenario B because, as in Fig.9, the state machine deadlocks when it enters state 0, and we need to re-plan it immediately. Scenario B, however, is difficult to measure using a certain index. In summary, we have chosen to express Scenario A as Scenario B i.e.

$$\mathbf{G}(player\ overthreat \longrightarrow \mathbf{X}pet\ taunt) \tag{6}$$

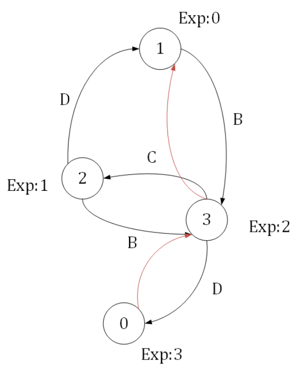

Figure 11. Scenario C

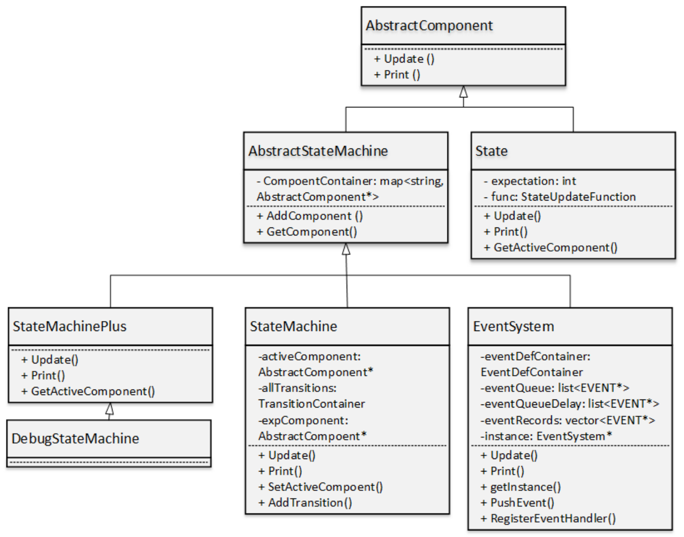

Figure 12. UML Diagrams representing the different state machines

The reason for scenario C is to ensure that the player can kill the dragon only after the monster and dragon have reached their intended location. As shown in the Figure 11, Scenario C still suffers from a deadlock where B stands for summon dragon and C stands for arrival. The state goes into deadlock when the player kills the dragon before the monster and dragon reach their intended location, manifesting in the game as the monster will be immortal forever. We must re-plan scenario C to get the monster out of its immortal state eventually. In summary, the actual constraints simulated in the game are Equations 6 and 5.

3.2 State Machine and Communication System

There are four categories of abstract state machines: StateMachinePlus, StateMachine, Event System, and DebugStateMachine (Figure 12). We use *payloads* to combine component names and objects to make it easier to enquire about components. The choice is to use a `std::map` as the container for all abstract components, where keys can be used for querying the component's name. It is worth noticing that abstract state machines also inherit from abstract components, which means that a state machine can also be a sub-state machine of a higher-level state machine as a component. At this point, the state machine can express hierarchical nestings. For parallelism, we have designed **StateMachinePlus**, a "vector" state machine. The abstract component inside this state machine is the **StateMachine**, and when StateMachinePlus executes the `update` method from the main game event loop, all its sub-state machines will execute the `update` method in sequence.

The **StateMachine** class consists of states and transitions. As a primary system component, the state machine is mainly responsible for detecting condition matches and switching states. As the state machine holds all transition conditions, the state machine detects all conditions from the currently active state to the destination state by executing the update method. If the condition holds, the source state will switch to the destination state. The transitions condition uses a lambda function that returns a boolean value. The lambda function is used because it can capture external variables. As with our previous abstraction of interactive systems, conversion conditions also need to capture events. When it is necessary to detect whether a state transition needs to occur, the state machine will first detect whether there is a record of the relevant event in the event queue. If so, the transition condition verifies its lambda function, and if the lambda function returns true, the state machine changes from source state to destination state. In addition, the currently active and expected components are recorded in the state machine. The active component is used to identify what state the state machine is currently in, and will also be used as a source state for the next transition verification. The expected component is used to identify what state the state machine needs to be in after re-planning.

EventSystem enables the communication between multiple systems by passing internal game events dispatched as messages. Events within the game become a medium of information. Considering that the same event should be distinguished if it occurs at different times (these events may have different parameters), we divide the events into EVENT-DEFINE and parameters. In the event definition, we define whether the event is a delay event and the response function (handler). A callback function is known as a handler. Before the event system reacts to an event, two prerequisites need to be satisfied. The event, first and foremost, needs to have EVENT-DEFINE. It is pointless to talk about an event if the event system doesn't specify what it is. Second, in order to process the event, we need to register the event handler, which is a lambda function that takes an event pointer. This is necessary because we might need information about the event's parameters. In order to properly handle an attack

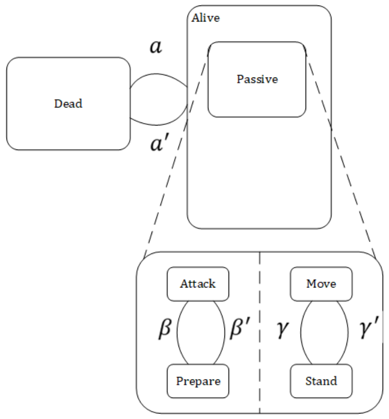

Figure 13. Character state machine

event, the event system needs to be aware of both the attacker and the target of the attack. The assault event, which carries the game world ID parameter, can indicate both the attacker and the defended player. It is important to note that many objects can register handlers for the same event to achieve collaboration between multiple systems. E.g., when the player dies, the sound system needs to respond to the event, and the player's pet needs to respond. EventSystem contains an *event* queue and the *delay event* queue: the former contains the events generated within the game update loop and the latter the delayed events. An EventRecords object can record all events for later analysis in our re-planning system.

Character is the base class for all four entities (NPCs and Player). Since pets have three policy modes, we must reuse the state machine to the maximum extent possible. We can assume that all other entities are in passive strategy mode so that the character state machine can be modelled as Figure 13. The *passive* state is refined into a parallel state machine, which includes a state machine that controls the player's attack and a state machine that controls the player's movement. However, the only synchronized transition conditions are α,α' and β. Figure 14 shows the LTS graph generated by the behavioural semantics. To simplify the representation, the name of states in the figure uses numbers. E.g., The formula $(\vee push|pop)((\vee g|b)(attack|move|sync)|die)$ is named as state 1. Those silent transitions($\{\tau\}$) should be removed for synchronisation (Figure 15). After renaming the states, the CCS formula is finally transformed into a traditional state machine (Figure 17 and Figure 16).

3.3 State Machine Parser

The role of the state machine parser is to translate LTL$_f$ formulas represented as strings into Debug state machines. In particular, the state machine parser translates formula 3 and 4 into the state machine in Fig.10 and 9, respectively. The

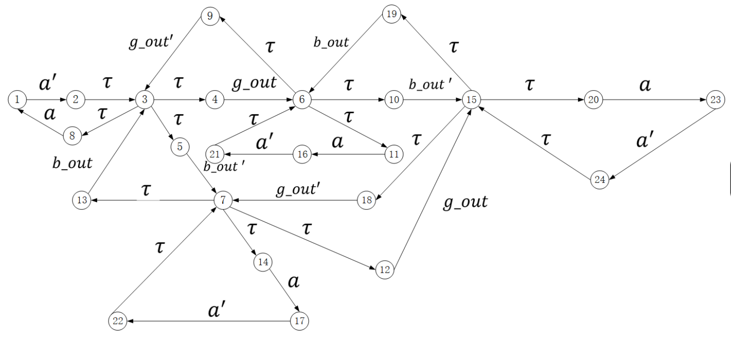

Figure 14. LTS generated by the behavioural semantics before closure

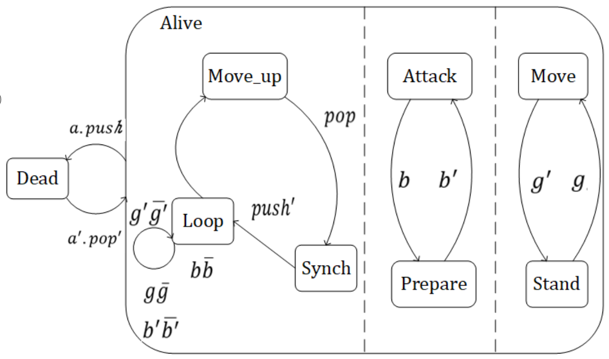

Figure 17. Character state machine synchronization

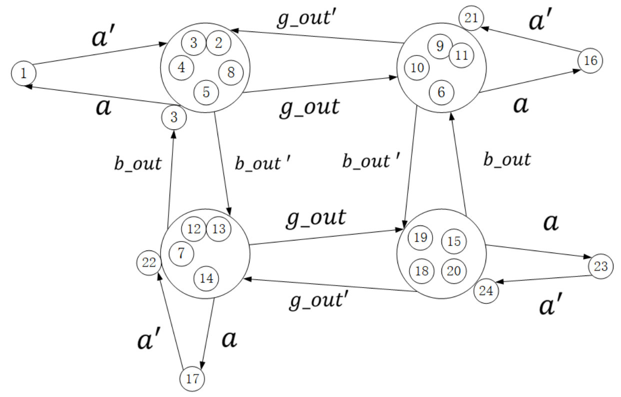

Figure 15. LTS after closure and minimization

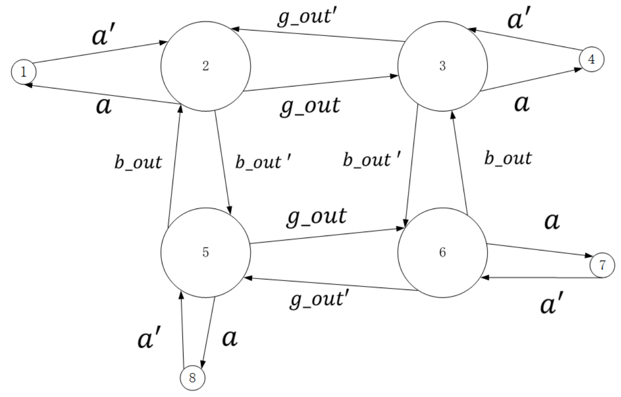

Figure 16. The same state machine with node renaming

whole translation process has two parts. First, the $\mathrm{LTL_f}$ formula is transformed into a `FlexibleFA` object representing a finite state machine. Then the object is translated into the final Debug state machine. We take these states and edges and construct the **State** object and the **StateTransition** object, respectively. State objects consist of a state name and an `update` function. If the state machine is currently in a **State**, the `Update` function defined under the **state** is called continuously. A state transition consists of a *source* state, a *destination* state and a *condition*. The state machine determines whether a transition is valid by testing whether an event has occurred and whether the condition is satisfied. StateTransition can be enabled or disabled. To parse the state, we set the label to the state name and the update function to `NULL`. We set the label to the event to parse the transition condition and assume that the condition always holds. Finally, as the red line shown in Figure 11, **Parser** adds all the one-way translation conditions to the inverse translation conditions and sets them not to be enabled, which makes it easier for the A* algorithm to find paths later on.

3.4 Adaptive Debug System

The adaptive system detects if the current game state is in a deadlock state. If the game enters a deadlock, this system automatically executes a re-planning algorithm on the relevant state machine. Ultimately, the system ensures that all monitored state machines operate in the desired state. Inspired by the PDDL language, we introduce two concepts: *environment* and *agent*. An *agent* is the observed Debug state machine, which contains the currently active state and the desired state. The *environment* is similar to P_D in PDDL and refers to the domain we need to re-plan. E.g., in scenario C, only the monster state machine in the environment needs to be observed. The critical point in detecting whether a state machine is in the deadlock is whether the currently active state has an outgoing edge. As in Figure 11, when the state machine enters state 0, the state machine will stay in state 0 forever as there is no outgoing edge. When a deadlock occurs, the adaptive system first finds the corresponding environment. Then for each agent in this environment, the A* algorithm is executed to find its shortest path from the currently active state to the desired state. Red lines in the same image depict the shortest repairing path from the current deadlock active state to the desired state of this state machine. Finally, the state machine moves to the desired state by executing the relevant transition function according to the shortest path derived from the A* algorithm. At the same time, all transition conditions on the path are set not to be enabled to prevent the state machine from entering a deadlocked state again. The re-planning path can also be exploited by the designer to correct the behaviour of the system later on and might provide valuable information to the final game tester.

4 Result and Evaluation

The simulation aims to reproduce bugs encountered in real games and verify that parallel hierarchical state machines can alleviate insomnia and debug state machines can solve amnesia. The adaptive debug system can detect deadlocks in the state machine in dynamic situations and re-plan them.

4.1 Reuse

We now discuss how our communication system can be represented in process algebra. The reason for doing this is to obtain the associated state machine. After doing so, we can better compare the size of our implementation to the one that would have been required by exploiting traditional state machines. While modelling state machines, it is necessary to include certain system states to make certain transition conditions visible and invisible. Just for modelling purposes on process algebra, we add a new sub-state machine called *sync state machine* representing our EventSystem. As EventSystem is capturing the events that need to be dispatched to other state machines, this state machine can capture the events happening in the state machines while making the synchronised action visible to an external observer. This extension is needed due to the limitation of process algebra, where synchronised actions are made silent to an external observer. As the state machine switches between state *Dead* and state *Alive*, the Sync state machine will also need to move from the internal parallel state machines to the higher abstraction level (*alive*) via a *move_up* action. When the Sync state machine enters the state *loop*, the Alive state machine will synchronise *g* and *b*, and a will be invisible. Given this, we can now fully characterise the character state machine in Figure 13 jointly with our event system, capturing the events from the character as the following process algebra system:

$$\begin{cases} \text{move\_up\_state} := \mathit{pop}.\text{synch} \\ \text{sync} := \mathit{push}'.\text{loop} \\ \text{loop} := \gamma.\gamma\_\mathit{out}.\text{loop} + \beta.\beta\_\mathit{out}.\text{loop} + \gamma'.\gamma\_\mathit{out}.\text{loop} + \\ \qquad +\beta'.\beta\_\mathit{out}.\mathit{loop} + \mathit{move\_up\_state} \\ \text{move} := \gamma'.\text{stand} \\ \text{stand} := \gamma.\text{move} \\ \text{attack} := \beta.\text{prepare} \\ \text{prepare} := \beta'.\text{attack} \\ \text{internal\_alive} := (\text{attack} \mid \text{move} \mid \text{sync}) \\ \text{synch\_alive} := (\nu\gamma, \beta)\text{internal\_alive} \\ \text{die} := \alpha'.\mathit{push}.\mathit{pop}'.\alpha.\mathit{die} \\ \text{system} := (\nu \mathit{push}, \mathit{pop})(\text{die} | \text{synch\_alive}) \end{cases}$$

After this conversion, we can exploit process algebra's behavioural semantics to obtain the resulting state machine depicted in Figure 13. Now, we can exploit the proposed size index for evaluating the complexity of state machines. We can now compare the reduced complexity induced by the synchronisation mechanisms to an equivalent state machine without synchronisation. The dissertation advocates that the latter will have increased complexity, thus tampering with the designer's choices. Our implementation allows to just design seven states (vertices) and six transition conditions (edges) in the character state machine (Figure 13):

- $V = \{ \mathit{dead}, \mathit{passive}, \mathit{alive}, \mathit{attack}, \mathit{prepare}, \mathit{move}, \mathit{stand} \}$
- $E = \{ \alpha, \alpha', \beta, \beta', \gamma, \gamma' \}$

This gives that proposed state machines have a size $SIZE_a$ of $|V| + |E| = 13$. As outlined in Figure 14, our system requires 24 states and more than 30 edges while directly converted into LTS from the associated process algebra system, while it requires 8 states with 16 edges, thus obtaining that the $SIZE_b$ of the same system can be expressed without synchronisation mechanisms is 24. Therefore, we can observe that our proposed solution considerably increases the overall number of edges and, if no minimisation techniques are adopted, further increases the number of states.

We can then propagate this reasoning to the whole system. Four characters in the game scene are running in parallel: in process algebra, this can be represented as $\mathit{player}|\mathit{monster}|\mathit{pet}|\mathit{dragon}$. Given that this expression can be implemented via StateMachinePlus and that each character shares the same characterisation of Figure 13 by reused implementation through instantiation where actions are just instantiated with different label names, the $SIZE_a$ the system size is just $SIZE_a = 4 \cdot (|V| + |E|) = 4 \cdot (7 + 6) = 52$. On the other hand, given that parallel systems undergo the explosion problem sketched in Figure 6, the equivalent system with no synchronization mechanism will require $|V|^4 = 7^4 = 2401$ states and $|E_{123}||V| + |E||V|^3 = 8232$ edges, where 4 corresponds to the name of the characters, $E_{123} := |E_{12}||V| + |E||V|^2 = 882$, and $E_{12} := |E||V| + |E||V| = 84$.

The comparison of $SIZE_a$ and $SIZE_b$ shows that the SIZE of the game has decreased significantly with the use of parallel hierarchical states. The decrease in SIZE means that the complexity of the system has decreased. We have not eliminated all of the useless states and transition conditions that cause insomnia, but we have eliminated some of them. The insomnia of the state machine was somewhat alleviated.

4.2 Debug State Machine

The purpose of testing the Debug state machine is to demonstrate that Debug state machine can achieve game constraints and resolve amnesia. We propose a win rate as an evaluation measurement in formula 6. If the player kills the monster game wins, while a game is lost if the monster kills the player. To facilitate testing, we provide an AI controller for the player. The AI controller can simulate user input to automate the testing. Specifically, the AI controller will randomly change the strategy mode of the pet and control the player to attack the monster. We set the number of games in a test round to 10. The number of wins and losses is counted at the end of each of the ten games. In this experiment, we conducted three rounds of testing. We compared the win rate before and after the game with and without the Debug state machine, as shown in table 1. We found that using the Debug state machine increased the win rate to one hundred per cent in the three rounds, which confirms that the Debug state machine is working and has solved the amnesia problem.

Table 1. Win rate in three tests

Test	Debug off	Debug on
$Firsttest$	win/lose: 6/4	win/lose: 10/0
$Secondtest$	win/lose: 7/3	win/lose: 10/0
$Thirdtest$	win/lose: 7/3	win/lose: 10/0

4.3 Adaptive Debug System

The purpose of testing the adaptive system was to verify that the entire flow from the discovery of the deadlock state to the correction of the state machine was valid. In order to verify the validity of the Debug state machine, we chose not to play according to the established flow of the game. The player will kill the dragon before the dragon and the monster reach their intended location. We compared the final shape of the game process with and without the Debug state machine under the appeal behaviour. We found that without the Debug state machine, the monster would invariably be immortal, and the player would never be able to win the game. If we use the Debug state machine, after the player kills the dragon, the monster is automatically de-immortalised so that the player can kill it. By comparison, we find that the adaptive system comes into play. The adaptive system completes Debug state machine's re-planning process from discovering the deadlock problem to correcting the state machine in the formula 5.

5 Conclusions

In summary, the research in this paper has two parts. The first part of the paper investigates insomnia and amnesia suffered by state machines in video games. The state machine adds all states and transitions conditions to memory without judgement, which leading insomnia. The result of insomnia is that many states and transition conditions are present in the game. This paper proposes a parallel hierarchical state machine to alleviate insomnia. We introduce the SIZE to measure state machine complexity to verify the effectiveness of parallel hierarchical state machines. Experimental results show that parallel hierarchical state machines alleviate insomnia successfully.

The leading cause of amnesia is the state machine's inability to 'remember'. The state machine forgot those partially satisfied conditions, resulting in our inability to impose time-dependent constraints on the game. In this paper, we propose the Debug state machine as a solution to amnesia. To verify the effectiveness of the Debug state machine, we introduce a victory metric to measure whether the Debug state machine is adequate. Experimental results show that the Debug state machine solves amnesia.

Immediately following this paper, we examine the issue of rescheduling in running time. The game needed re-planning because the Debug state machine had a deadlock problem. Deadlocks eventually cause the game to stall indefinitely at some stage. In this paper, we propose an adaptive system to reschedule the system. We eventually found that the adaptive system solved the deadlock problem.

Harel [1987]'s extension of state graphs is the theoretical basis for our design of parallel hierarchical state machines. In terms of practical code, Davison's implementation of state machines is the base code for our implementation of parallel hierarchical state machines. Bergami et al. [2021]'s work inspired us to use LTL_f expression constraints and translate them into Debug state machines. Also, Bergami et al. [2021]'s work on re-planning inspired us to try replanting Debug state machines in running time.

Overall, our design of the Parallel Hierarchy State Machine and Debug State Machine allows for the reuse of state machines. On a design level, the new state machines ease the burden on game developers. They no longer have to deal with the complex transition relationships in the state machine and can devote that energy to the more complex game logic design. At the game software level, the new state machine reduces the memory usage of the software. Secondly, our re-planning algorithm reduces the burden on game bug debuggers. Typically most game bugs are exposed during the in-beta phase, with a few bugs appearing after launch. As most games now have a long lifeline, bugs will intermittently appear in late and continuous updates, and the game debugger is always in a cycle of receiving bug feedback and working on bug fixes. With the game iterates, fixing bugs may become more complex as more factors need to be considered. The most important feature of our re-planning is that it is done on a running time basis. If the game could self-correct some of the bugs at the running time, the debugger might not need to get caught up in a spiral of bug fixing.

5.1 Future work

Parallel hierarchical state machines alleviate insomnia by reusing states and transition conditions as much as possible. We are not getting to the root of insomnia completely. Suppose we want to solve insomnia for good. In that case, we need to remove state and transition conditions that are not currently in use, which means that the state machine can load with states that are likely to arrive and transition conditions that are needed based on the current state. Setting a visible range for the state machine might be a solution.

The adaptive system can only detect deadlocked states in the Debug state machine. However, some states fall between the desired and deadlocked states, similar to the previously mentioned state 1 in Fig.10, which means that perhaps the time to trigger a correction should not just be when the Debug state has entered the worst state. How to measure whether the current state is "bad" enough is a topic for the future. As the example above happened in GTA5, some bugs are harmless to the game. Even players are happy to use it. The example makes us wonder how we should evaluate the harm a bug can do to the system. Adaptive Systems should prioritise re-plan for bugs that are particularly damaging to the system and seriously affect gameplay.

Acknowledgement

I want to thank my supervisor, Giacomo Bergami, who provided me with great suggestions and priceless criticisms for the writing of the dissertation and whose theoretical and patient guidance has dramatically benefited me during the process.

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] Giacomo Bergami, Fabrizio Maria Maggi, Andrea Marrella, and Marco Montali. 2021. Aligning Data-Aware Declarative Process Models and Event Logs. In *International Conference on Business Process Management*. Springer, 235–251.

[3] Blizzard Entertainment. 2004. *World of Warcraft*. Game [Windows]. Blizzard Entertainment, California, United States. Played August 2022..

[4] BooJWood. 2022. *Grimbalto BUG*. https://www.bilibili.com/video/BV1aY411T7ji?spm_id_from=333.999.0.0&vd_source=a75a153ad25623a18647640b7fc96def

[5] Michael Cashmore, Andrew Coles, Bence Cserna, Erez Karpas, Daniele Magazzeni, and Wheeler Ruml. 2019. Replanning for situated robots. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 29. 665–673.

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

[7] Brother Foller. 2022. *RDR2 - What if you kill the bear that Charles asked you to bypass*. https://www.youtube.com/watch?v=e1Mr36vstIg

[8] Abhishek Goyal, Prateek Mogha, Rishabh Luthra, and Neeti Sangwan. 2014. Path finding: A* or dijkstra's? *International Journal in IT and Engineering* 2, 1 (2014), 1–15.

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

[10] Fred Kröger and Stephan Merz. 2008. First-order linear temporal logic. In *Temporal Logic and State Systems*. Springer, 153–179.

[11] Youichiro Miyake. 2017. *Current Status of Applying Artificial Intelligence in Digital Games*. Springer Singapore, Singapore, 253–292. https://doi.org/10.1007/978-981-4560-50-4_70

[12] Rockstar Games. 2013. *Grand Theft Auto 5*. Game [Windows]. Rockstar Games, New York, United States. Played August 2022..

[13] Rockstar Games. 2018. *Red Dead Redemption 2*. Game [Windows]. Rockstar Games, New York, United States. Played August 2022..