Coordination mechanisms in an MMO

Internship Report
Business Analytics

Corné Sprong
1396056

VU supervisor
Tibor Bosse

Second reader
Sandjai Bhulai

Company supervisor
Gert-Jan van Rootselaar

Table of Contents

1. Introduction

1.1. Company

Connected Dreams B.V. is a video game internet company that is currently developing and marketing Tailplanet; a unique on-line video game experience. Connected Dreams is based in Amsterdam and currently consists of a team of 5 people. Tailplanet is Connected Dreams' first project and has been in development since its founding in 2008. [7]

1.2 Tailplanet

Tailplanet is a 'continuously on-line' Massively Multiplayer Online Game (MMOG) It utilizes state of the art graphics- and internet technology, with an internally developed game engine. At the beginning of the game, players get a house with several rooms that they can decorate, as well as a pet (known internally as 'Slurfie') to train and play with. Tailplanet caters to different demographics by offering both caretaking (feeding, washing and playing with your pet) and action (training your pet and teaching it tricks to prepare for competitions) activities. As players progress they unlock new regions and new possibilities, for example holding a Trick Duel or participating in a beauty contest. At a later stage, it will also be possible to cross slurfies to form new breeds. Children are expected to make up a large percentage of the playerbase; with this in mind features for child safety have been implemented. For example, by default players can only chat to people that they have added as friends. A friend can only be invited by knowing his email address, or by having him as a friend on Facebook. Anyone can register and play for free, but a player can obtain advantages or premium items using in-game diamonds, which players (or their parents) can purchase in a webshop for real money. These micro-transactions will form the bulk of Connected Dreams' income.

Illustration 1: Typical screenshot of the game.
The pet Slurfie stands on a training mat, learning his first trick, a basic jump. At the top, from left to right: chest that holds player's items (currently empty), diamonds and in-game money, the player's level and experience, a portrait of the pet and its name, a summary of its healthiness and social aspects and a chat window for feedback. At the bottom, the icons are a house (for moving to different rooms within the house), the chest (for opening the inventory), the mission book (which holds the available and completed missions), the passport (holds information about the pet's skills

and achievements) and the trick controller (now obsolete). The icons on the right are the world (giving access to other locations, not yet unlocked here), a shopping bag to open a shopping window, a help button that gives short information about the interface and a button that toggles the chat stream overview. In the center we see the pet after successfully trying an upward jump.

1.3 Paper outline

For this assignment – explained in section 2 - I have taken a use-case based approach. This means that the required protocols (Section 4) were chosen based on the common elements in sample of the potential use-cases (section 3). I evaluated these protocols on the criteria of robustness, ease of use and quality of behavior (section 5), and compared my findings with existing literature on the topic of coordinating state machines and state machines in AI (section 6).

2. Problem description

In this section I will talk about Multi Agent Systems and describe the technology underlying TailPlanet, leading into the problem description.

2.1 Multi Agent Systems

A Multi Agent System is a system in which autonomous agents interact with each other and the environment. MAS have applications in logistics, robotics and video games among others. They are particularly useful when the situation is complex and dynamic, and central decision making is unpractical. Each agent has its own goals and access to information about its own area of responsibility. Even though each agent has its own activities, it should contribute to solving the full problem through coordination. Thus when applying MAS techniques, both the design of the agents and the design of their possible interactions is important. The appropriate architecture is context-dependent.

The challenge is that the agents need to operate in a coordinated fashion, but at the same time we do not want to fully centrally specify their behavior (with a central state machine). In other words we need a balance between autonomous decision making and centralized control. The level of autonomy (versus control) depends on the use case (the task at hand). In MAS literature, various mechanisms have been established to coordinate task allocation and plan generation. These fit in the categories of organization structuring, contracting and multi-agent planning [4]. Most of these mechanisms are based on some form of negotiation.

In the video game Tail Planet the agents represent virtual pets that interact with each other. Part of their behavior is autonomous and part of their behavior needs to be coordinated. The behavior of objects in Tailplanet is modeled in Enterprise Architect using Hierarchical Finite State Machines (HFSM). These are then exported to XML and compiled into JavaScript by an in-house compiler. This setup makes it easier for behaviors to be implemented by modelers without a Computer Science background.

2.2 Statemachine modelling

I will give a short introduction about the variant of HFSM modeling that Connected Dreams is using. The HFSM that runs the pets uses a basic form of goal-oriented behavior. At a high level of the hierarchy (known internally as 'the GOB'), there are branches for behaviors such as eating and sleeping, each with an associated score (called a promise). At the deeper levels this behavior is

further refined, for example a pet in 'hunger' behavior can walk to a foodbowl, or chooses to beg for food instead if no full foodbowl is present.

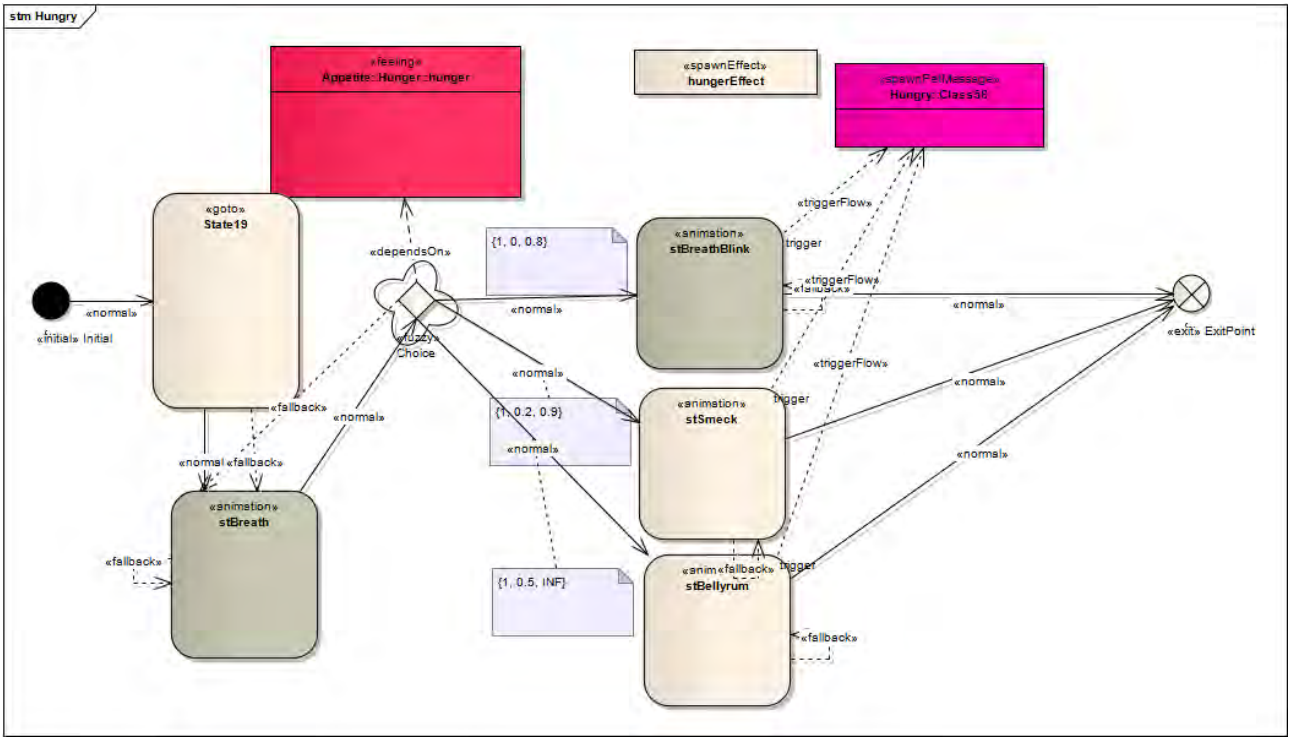

Illustration 2: Diagram Hungry

The diagram *hungry* (Illustration 2) describes the behavior of a pet when there is no food available. Execution starts at the black circle. The <<goto>> tells the pet to move towards the player (the front of the room). When it has reached its destination, it turns face the camera and plays the stBreath animation. At the diamond, it chooses a path based on the value of the variable 'hunger'. The tags on the outgoing transitions contain the range in which hunger may fall for that transition to be taken, and a weighing factor. That weight is used when ranges overlap, thus allowing for stochastic behavior. Outgoing triggerflow arrows (dashed) denote that an action should be performed upon entering or leaving the connected node. Possible actions include sending a trigger to another object or itself, spawning a new object or performing a search for an object that meets certain constraints. In this diagram, the <<spawnPetMessage>> causes a balloon to be displayed above the pet, with the icon for hunger in it. At the crossed circle, control is returned to a higher state machine (if there is no higher level, the object is destroyed instead). Rounded rectangles denote states, rectangles with a folded corner are constraints and rectangles have various meanings.

Initially, each object only had at most one associated state machine, but support for multiple state machines was added later. This is a convenient feature for the introduction of auxiliary FSM's. The state machine in Illustration 3 listens to START_GAME events (which can happen at any moment), and records this occurrence. This allows the animation state machine to process the event at a later moment. Typically we do not want to interrupt an action in progress, so to steer behavior flags are used which increase the promises for certain actions.

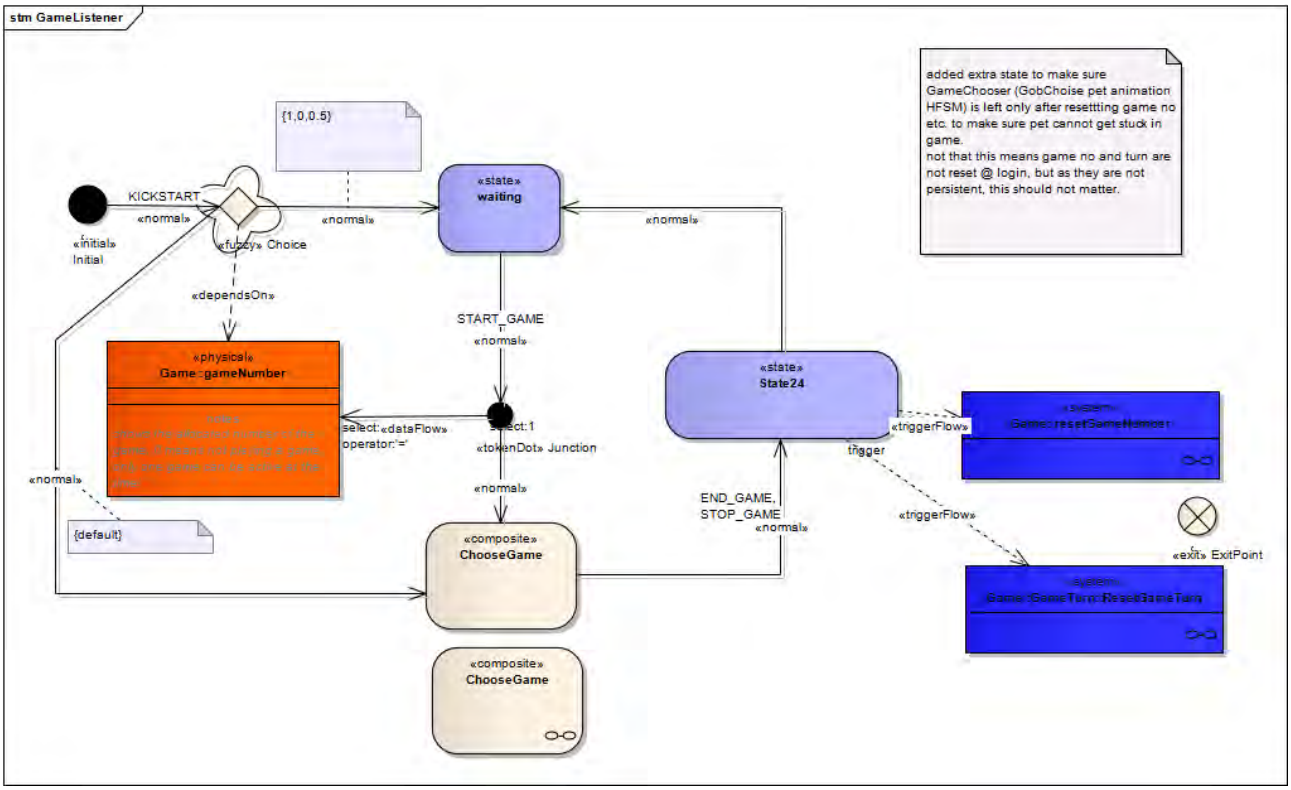

Illustration 3: Diagram GameListener

Not all transitions are automatic. Sometimes, execution pauses at a node and waits for a specific trigger to be received. These triggers can originate from the system or from state machines. In Illustration 3 the system remains in the state 'waiting' until a trigger named START_GAME is received. From this trigger, a number is extracted and stored into the variable 'gameNumber'. (This functionality was added by me, to increase the versatility of the messaging system) Execution continues in the composite state ChooseGame (which branches on the gameNumber and enters a game-specific flow, not shown here). Eventually the game ends, resulting in an END_GAME trigger. The state State24 resets the gameNumber and gameTurn variables and the FSM returns to the Waiting state, ready to be notified of a new game. If a START_GAME is received while it's not in the waiting state, the message gets discarded. It is the responsibility of the sender to determine whether this pet is already busy, before telling it to start a game.

The setting of gameNumber also causes the animation state machine of the pet to pick a game-specific flow; this will be referred to as 'being in the game-state'. A pet that is in a game-state will postpone fulfilling its needs such as hunger and thirst, but only exhibit the behavior for that game. We ultimately chose this solution because otherwise the unpredictability of a pet's behavior during a game would be unacceptably disruptive.

2.3 The future of Tailplanet

Tailplanet will have a large shared world with many locations that players and their pets can visit. The only location available when I joined Connected Dreams was the player's livingroom. Pets could be given food and water and would react to it. Some ideas for mini-games already existed, but most were in the conceptual phase, without having been implemented.

One existing mini-game involved the hatching of an egg (that your pet comes out of), where the player had to keep both sides of a rolling egg at the right temperature using a floorlight. That mini-game was eventually suspended, because it introduced too much of a delay between creating your account and obtaining your first pet to play with. However, the analysis of its implementation provided useful information about the nature of the problems and possible solutions.

2.4 Problem description: mechanisms for coordination

The research question can be stated as: "What are the best ways to achieve coordination for typical use cases?" This leads to follow-up questions “What are the typical use cases in Tailplanet?”, “What methods for coordination exist?” and “How can a method be evaluated?”

2.5 Use case-centric approach

My approach for this assignment has three steps. First,I will identify possible use cases for multi-pet (multi-object) interactions and abstract common elements from these interactions. Finally I will design and evaluate implementations of these elements.

3. Use cases

In this section I present a sample of the use-cases that we maywant to support in Tailplanet (section 3.1) and propose a theoretical framework that divides a typical activity into four phases (section 3.2).

3.1 Examples

First we need to determine what typical use cases we are expected to find. At the start of my internship, there were a few ideas for activities and those were in the concept phase. By establishing a broad range of use cases, it should be easier to determine common elements. Activities can be derived from the real-world games that children play and from activities for dogs. We should also take into account the future plans for Tailplanet.

In these use cases we will look at the arising interactions between agents, players and objects. The Trick Duel, Puzzle Gauntlet and Team Sports are stand-alone activities, the kind of mini games that might be added in the future. Outside these activities, there are also smaller interaction scenarios such as Say Hello, Coin Search and Pet as Assistant..

3.1.1 Trick Duel

In Tailplanet, the players can make gestures that correspond to certain tricks, which are decoded and then relayed to their pet. The Trick Duel is a turn-based game where players use their pets to perform tricks and combinations of tricks. One player (the 'leader') inputs a sequence of moves, which the pet then tries to perform. When the pet has attempted all the tricks (successfully or otherwise) or the timer runs out, the turn switches to the other player (the 'follower'). This player

has to copy the moves of the leader. Points are awarded depending on the quality of the performance. At each of the leader's turns the sequence becomes longer and the time limit is extended. The player with the most points wins the round, then a new round starts and the roles of leader and follower are reversed. When a player scores three wins, he wins the match.

This activity takes place in the arena, which is a Colosseum containing a mat for the pets to do their tricks on. This mat can only hold one pet at a time, so a pet has to make room for the next participant when its turn ends. A pet that doesn't have the turn should ignore the gestures of its owner.

3.1.2 Puzzle Gauntlet

This is an activity for two players. Their pets start out in the top two corners of the room, and a gauntlet is randomly generated. The goal is to lead the pet to the bottom of the screen as quickly as possible. The gauntlet moves by itself, but players can influence the gauntlet by moving (e.g. sliding or turning), adding or removing objects.

3.1.3 Team Sports

Team sports are coordination-intensive activities par excellence. Let's take football for example. Team-sizes would be limited to 2 or 3 on each side, but this already offers much potential for interesting behaviors. The players can be spectators or more involved as coaches.

3.1.4 Say Hello

An important feature of Tailplanet is visiting friends. When players visit a friend's house (i.e. move their camera to one of their rooms), they can send also their pet along. If the player tells its pet to go and visit, this pet moves to his owner's doormat (or the exit of wherever he happens to be) and is then teleported to his friend's livingroom. Once the pet arrives, it should look for another pet and both pets should wave to each other.

3.1.5 Coin Search

When a player discovers a new area, there will be coins hidden on the floor. The player's pets can search the area. It would be nice if the pets could 'intelligently' divide the work, instead of checking areas multiple times and blocking each other's paths.

3.1.6 Pet as assistant

In these cases, it is the player(s) who perform the activity and the pets have a supporting role. As an example, consider a game of Simon. This is a game where players have to reproduce an ever-increasing sequence of colors. When a player hesitates, its pet may try to help by hinting at an answer, provided it is happy and well-slept.

3.1.7 Use cases to increase immersion

Besides the full delimited activities there are also smaller behaviors that we would like to show on occasion. For example, pets acknowledging each others existence while in the same room. This raises the question on what these behaviors should be, and what triggers them. One potential moment for spontaneous action is when a pet is bored (the default state when its needs are fulfilled) and another pet happens to be in the room. Rather than trying to attract the owner's attention, a pet could play with the other pet instead.

Another case could be if there is one available resource (e.g. food bowl) and multiple pets that want to use it. The first pet to arrive at this resource acquires a lock on this resource, and other interested parties would be informed of this event. There are many different ways to handle this exception. The easiest way is to give up on obtaining the resource. An alternative is to try to obtain the resource by force (pushing the other out of the way). Yet another way is to negotiate with the other pet, who may yield if it likes or fears the other pet enough. A more complex negotiation may cause a commitment on the asking pet to reciprocate at a later stage.

3.2 Phases

A typical game activity has several phases. The first phase is the *invitation phase*. Here we determine which players and pets will be the participants of this activity, and whether or not it can take place at all.

The second phase is the *preparation phase*. Any objects that will be used for the activity are initialized or spawned, and involved pets are gathered at the appropriate location.

The third phase is the *execution phase*, which can be seen as the body of the activity. Pets, players and objects act according to the rules of a game.

The final phase is the *wrapup phase*. This is when the activity has ended and typically when results are shown if the activity calls for it, temporary objects get removed and pets are returned to their owner's home. Sometimes this stage can lead back to the preparation phase or execution phase, for example if there is a dialog for restarting the activity.

For some use cases these phases can be very simple or even absent. For example, the *Say Hello* case has a very small wrapup phase, consisting only of making the pets forget one just arrived (to avoid repeated greeting).

4. New Protocols

In this section I will provide protocols for common patterns of interaction. The first four subsections (4.1-4.4) cover common patterns from the four phases that were introduced in (3.2). The final subsection contains general patterns that are not specific to one phase.

A protocol is a set of rules for communication. The new protocols are subject to a number of constraints. Of course they are required to lead to the desired behavior, but this is often a subjective measure. All state machines must be free of deadlock, that is, they must never mutually wait for a response. If the machines are run on different machines, there is the possibility that one of the participants stops responding unexpectedly (e.g. when the browser is closed), this possibility should be handled gracefully. The robustness of these solutions will be evaluated in Section 5.

From the list of use-cases, the Trick Duel and Say Hello have been implemented as proofs-of-concept. The Trick Duel is taken as the main example use case, following the four phases as given in section 3.2 (see section 4.1-4.4, respectively). This should cover many of the recurring elements in game activities. After that, some general useful constructs will be covered in section 4.5 and considerations for the not implemented Coin Search case are given in 4.6.

An overview of the participating objects in the Trick Duel process is given in Table 1. *Local* refers to the side that started the invitation and *Remote* refers to the challenged side.

Name	Function	Comments
Inviter	Track mutual agreement between players, spawn coordinator	Local only
Listener	Inform remote player of Trick Duel invitation and get consent.	Remote only
Mailhandler	Spawn a Listener when the Inviter calls for it.	
Coordinator	Enforce game rules, keep scores, coordinate pets, end game.	Local only
Helper	Update UI elements. Stop game. Monitor disconnection by A.	
UI elements	Give visual feedback to players about the game state.	Large number.
Players	Draw gestures for the pets.	
Player object	Keep track of whether a player is in a game.	
Pets	Perform the tricks that are entered.	

Table 1:Objects that are involved in the Trick Duel process.

4.1 Invitation phase

In this phase, we determine the participants for the activity and decide whether it can take place. Suppose the activity is between players A (the host) and B (the visitor). When player A invites player B for an activity, a flow is started to determine if player B wants to join. If player B is online when the invitation is made, we should display a dialogue window on his side with the request. If he is not online, we should store the invitation for a later moment. In this case, the original challenger will need to confirm his challenge. The bottom-line is that both players must have given permission within a certain time-interval for the activity to start.

4.1.1 Spawning Inviter and Listener

To implement this behavior, I used two objects. An *inviter* on A's side, and a *listener* on B's side.

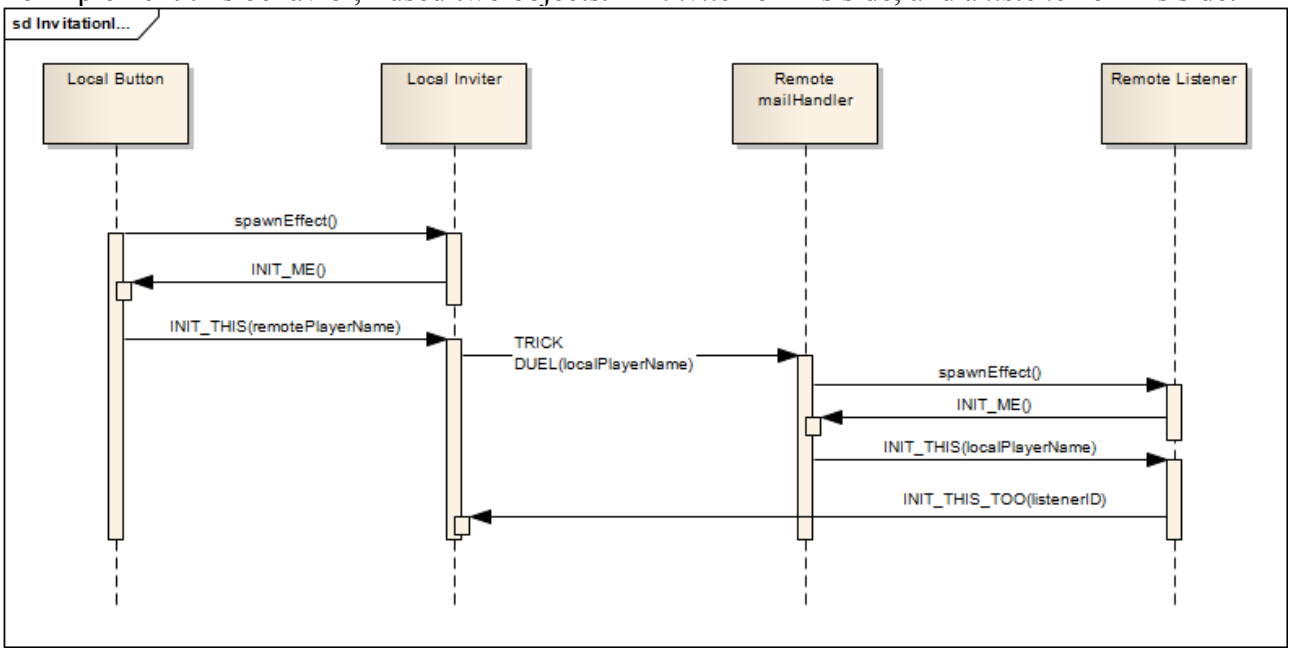

Illustration 4

The sequence diagram in Illustration 4 shows the process of spawning and initializing the inviter

and listener. The Trick Duel button under the contact card[1] spawns a local Inviter object. This inviter is initialized with the name of the remote player. It uses this name to send (in this case) a Trick Duel request to the remote mailHandler of the intended player. This mailHandler spawns a Listener and initializes it with the name of the local player (i.e. player A) and the ID of the inviter. Finally the listener informs the inviter of its ID. At this point the listener and inviter know both players' names and each other's ID.

The next step is obtaining permission from both participants to start the activity. Neither the pets nor their owners should already be in an activity. The inviter and listener check if their pets are already in a game. If so, the Trick Duel cannot take place and is aborted.

4.1.2 Semaphore on being in game

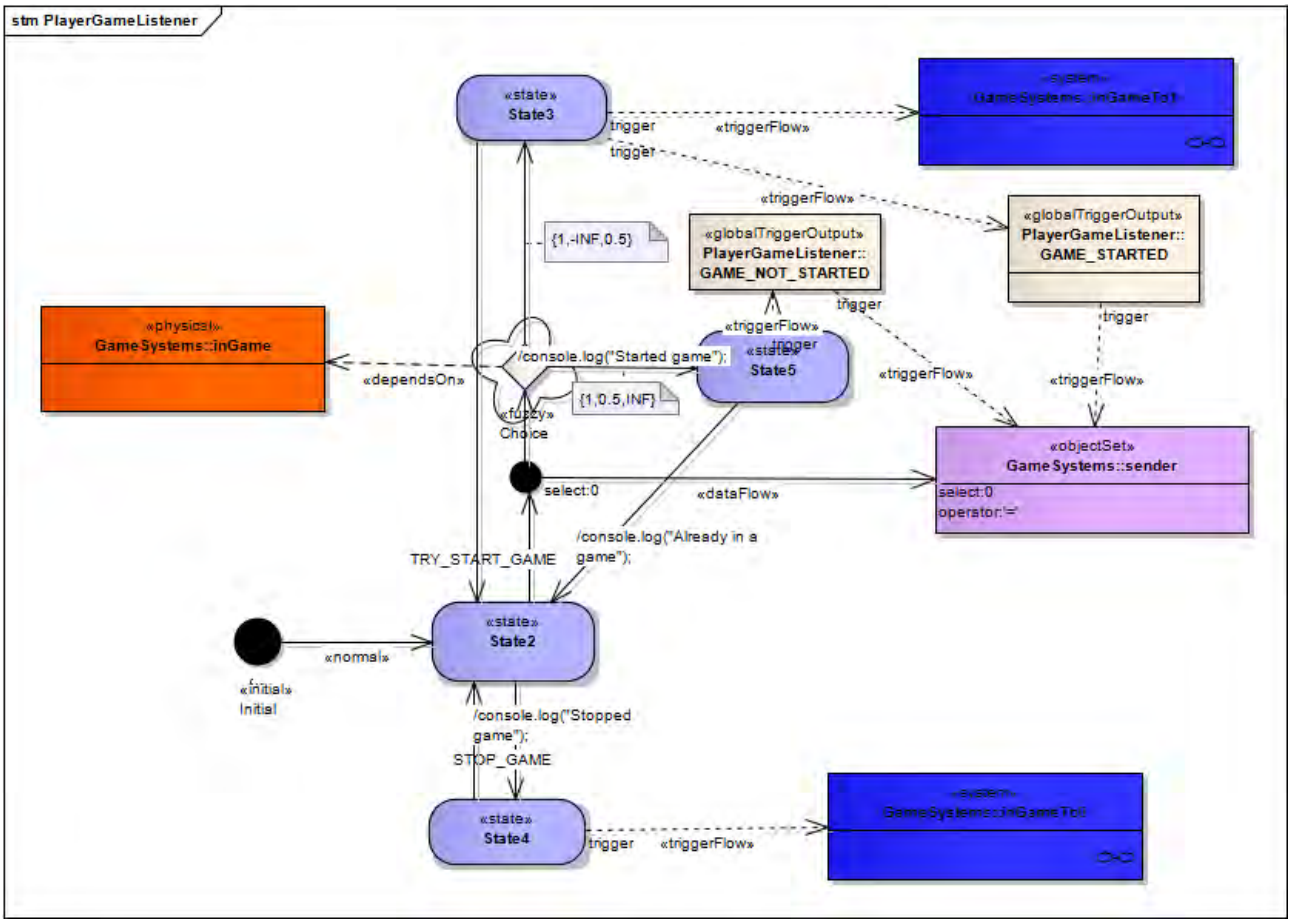

Illustration 5

To ensure that the players are available, we try to obtain locks on their *inGame* variable (Illustration 5). If we cannot obtain these locks for both players, the activity is canceled. A <<globalTriggerOutput>> models that a message should be sent. The target of that message is determined by the target of the outgoing triggerflow.

1. A contact card is a UI element below the main screen that refers to a specific player.

The request for the lock is made by sending a TRY_START_GAME trigger to the player object. When handling the request it stores a reference to the sender, then checks if the *inGame* variable is true. If it is, the reply to the sender is GAME_NOT_STARTED, otherwise *inGame* gets set and GAME_STARTED is returned. The lock is released again with a STOP_GAME trigger. This trigger should only be sent by the owner of the lock (though a check against the sender could be made, 4.5.2 Testing for a specific sender)

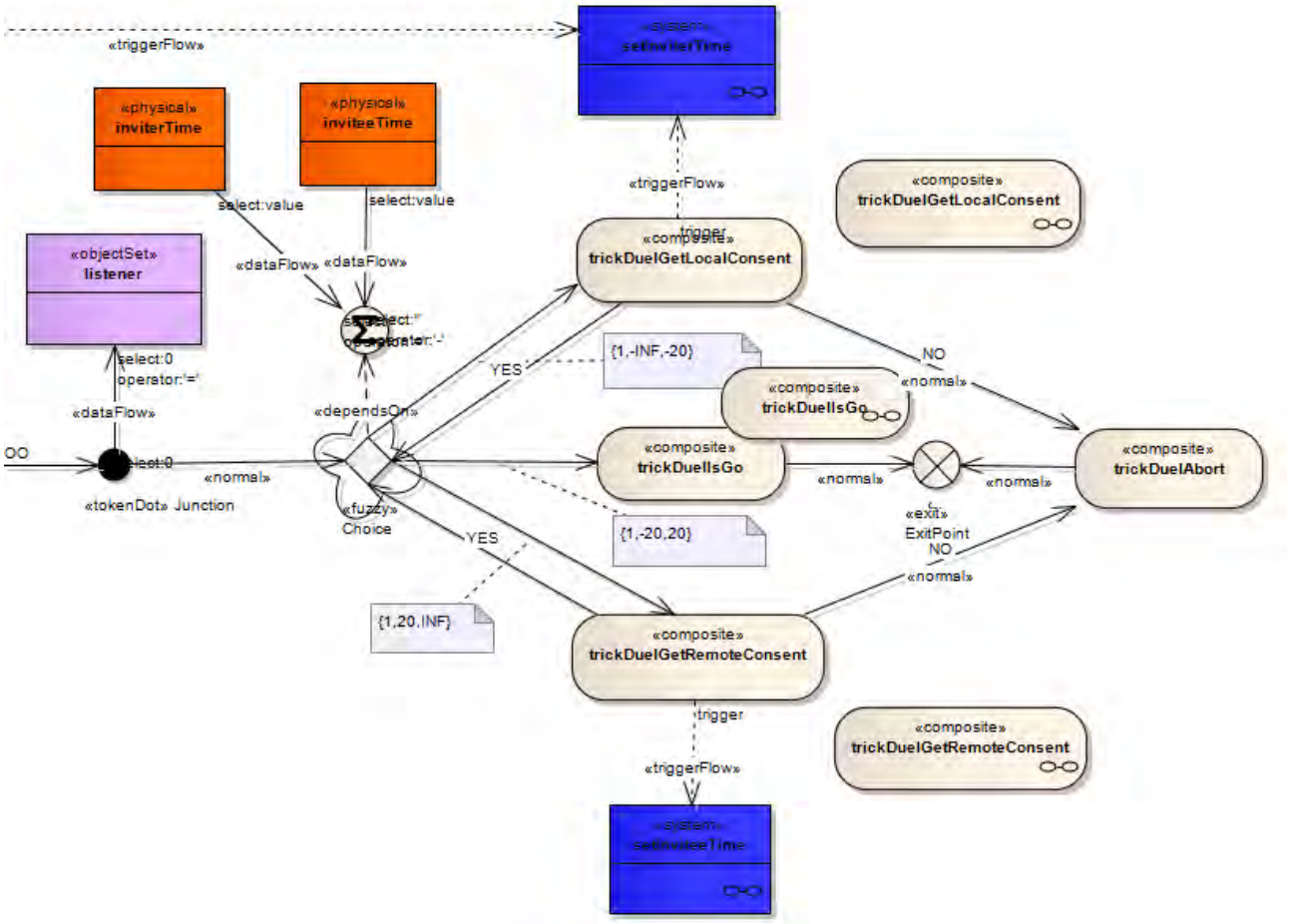

Illustration 6

4.1.3 Getting mutual permission

The invitation variant where both players must accept within 20 seconds of each other. The diagram (Illustration 6) can be interpreted as “while the difference between the times of acceptance is larger than 20 seconds, ask the oldest responder for confirmation". If that responder confirms, update the time of the response. If the responder rejects, the invitation process gets aborted and the locks are released. When both participants agree within 20 seconds, the inviter completes the process by spawning and initializing a coordinator object for player A and then closing itself and the listener.

As a simpler alternative, the inviter's permission could be implict and the invitee gets 20 seconds to confirm. If he doesn't, the invitation is canceled.

4.1.4 Autonomous invitation

In the Say Hello case, there is no actual player involvement in the invitation phase, but the pets have to coordinate the greeting by themselves. The newly arrived pet knows that it's a visitor through a

flag that's set. This flag causes it to choose the visitorFlow when it reaches the GOB[2]. When this flow gets selected, the flag gets unset (so it only enters this behavior once per visit). The visiting pet must obtain the ID of another pet to do this activity with. To this end, it broadcasts a message WHO_IS_HOST to every object in the room and waits for a reply while looking around the room. Pets that hear the message may choose to respond to it with I_AM_HOST. If the visitor receives an I_AM_HOST trigger, it stores the sender ID and acknowledges. This acknowledgment sets a flag on the host's side. After the handshake, the participating pets know that the activity will happen and who the other pet is. If, however, there is no reply, the pet will make a sad face and go back to its regular behavior.

4.2 Preparation phase

Now that the participants are confirmed, it is time to prepare the activity. For the Trick Duel this means gathering the pets at the challenger's arena. After obtaining the names of the involved players, the coordinator tells the pets to move to the arena using the *SET_GO_VISIT* {room owner, room name} trigger. This trigger is received by the visitation state machine on each pet, which then sets a flag to remember to visit once the current flow is completed. Upon arriving, the pet acknowledges this, so that the coordinator knows whether both pets have arrived. Whenever an object changes room, all its state machines are reset, and restart at the initials. Its variables are not reset, so these can be used to resume execution where he left off.

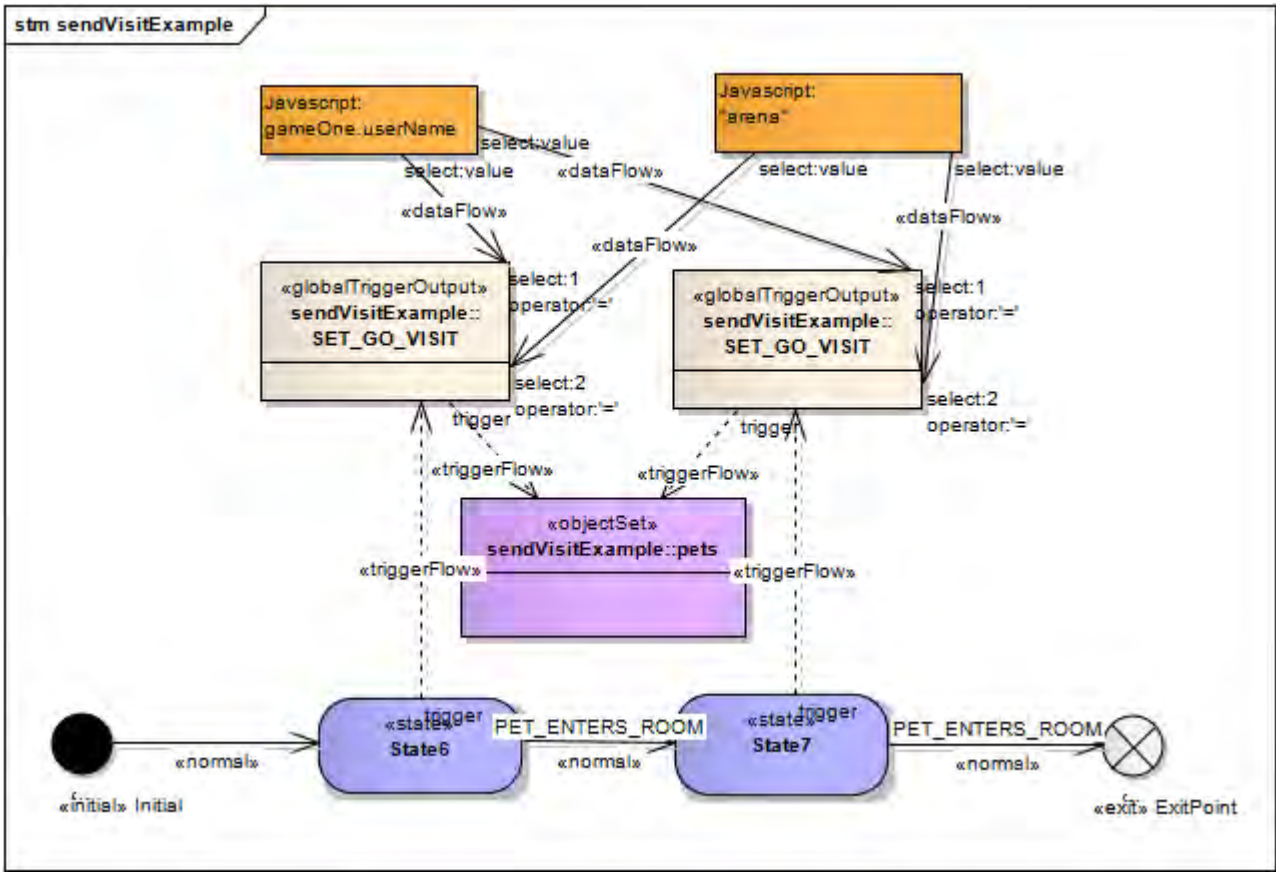

Illustration 7

2 GOB (Goal Oriented Behavior), here refers to the pet's top level decision state machine.

The diagram from Illustration 7 comes from the main state machine of the coordinator. The explanation window is shown for both players and the host's pet is told to visit his owner's arena. Once that pet arrives it sends a PET_ENTERS_ROOM trigger, which prompts the coordinator to call the other pet to the arena. The pets were sent in this order to work around an issue with dynamically loading rooms. Finally, the pets are informed that they are now in a game through a START_GAME trigger (which is handled in the FSM in Illustration 3). Not shown in the diagram is the guard on the PET_ENTERS_ROOM transitions, which checks if the signal originates from the pet that was sent (Section 4.5.2).

4.3 Execution phase

4.3.1 Turn based game with coordinator

For the Trick Duel, this is the phase where pets take alternating turns, performing the tricks that players enter. The bulk of the coordination efforts are performed by the coordinator, which tells the pets when their turns start and end. The coordinator has three concurrent state machines.

The main FSM (Illustration 8) listens to incoming gestures and relays them to the pet if they are correct. State idle has a private queue. Incoming messages are processed sequentially, using a composite state per event.

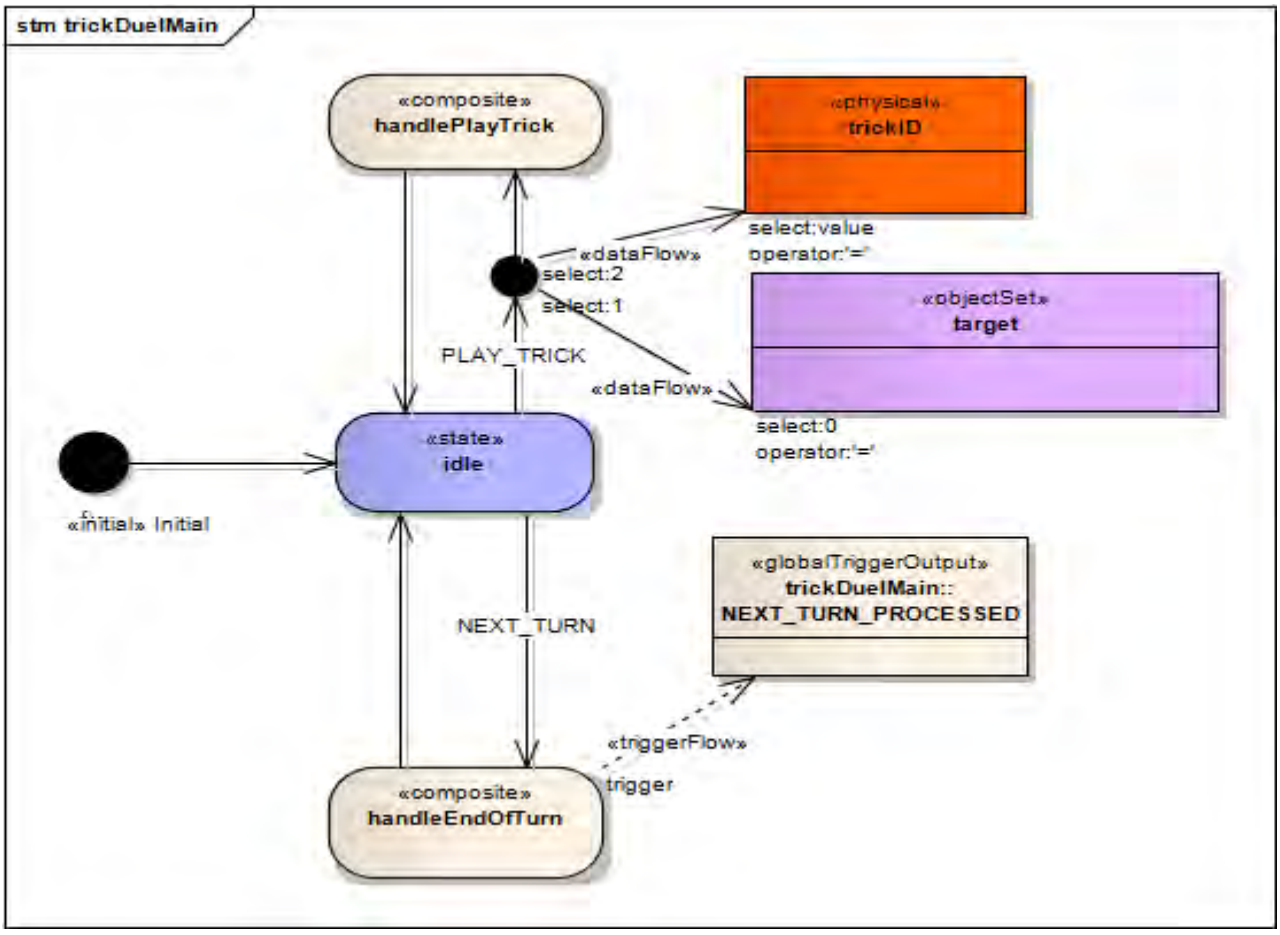

Illustration 8: Diagram trickDuelMain

In this state machine, we listen to two events; PLAY_TRICK, which is the event of a player entering a gesture, and NEXT_TURN, which is sent when the current player is done with his turn. These events are then processed by composite states. *HandlePlayTrick* checks if the input comes from the active player and if the entered gesture is valid. If it passes these tests, the gesture is relayed to the active pet. If the trick was the last of the sequence, the trigger START_DOING_MOVES is sent to another state machine within the coordinator. *HandleEndOfTurn* makes the current pet leave the mat, then starts a new turn (if there are turns remaining), or a new round (if there are rounds remaining) or it presents the score screen with the option to rematch.

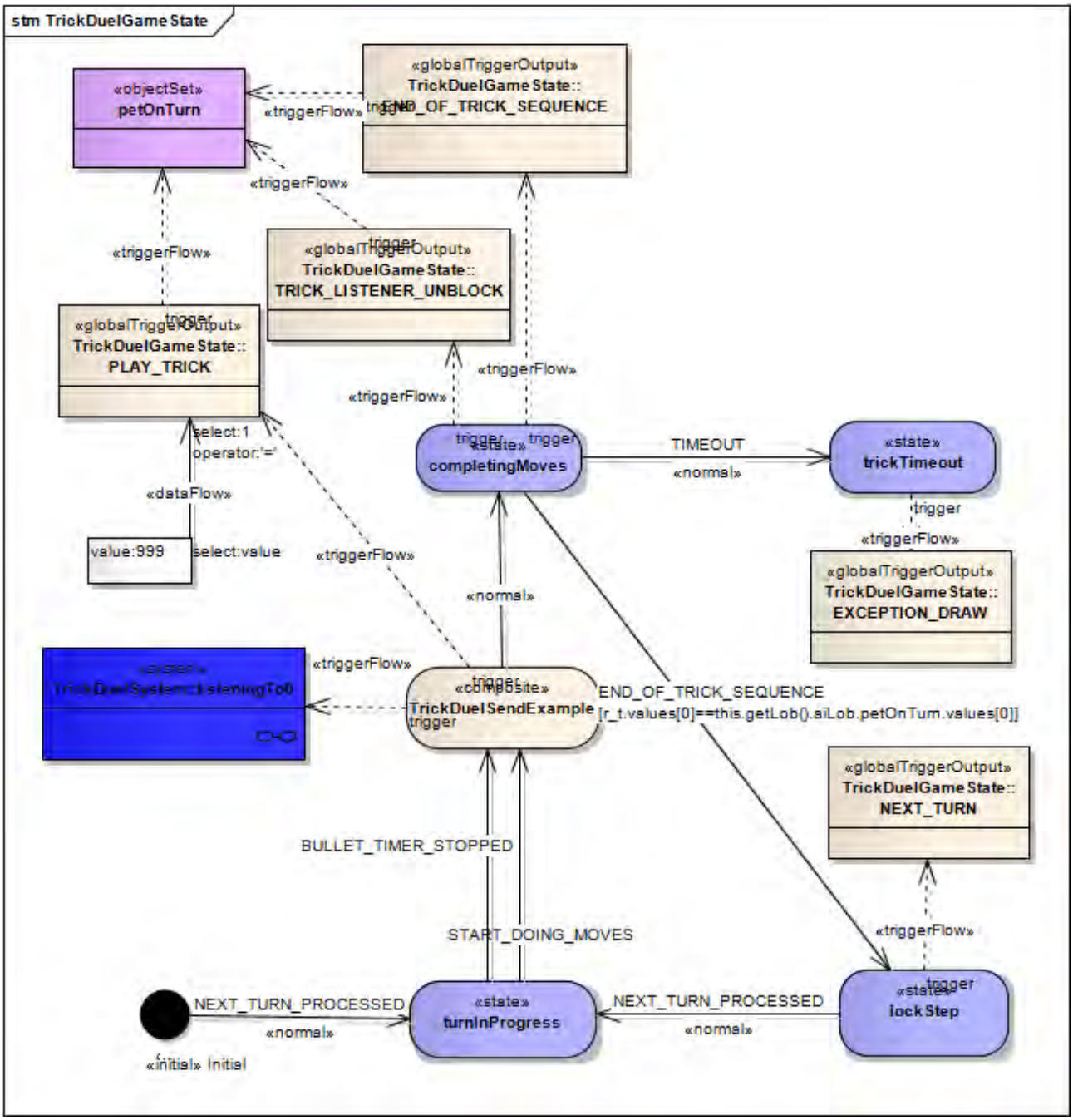

Illustration 9: Diagram TrickDuelGameState

While in state turnInProgress, the GameState FSM (Illustration 9) listens to events that cause the current input-phase to end. When the bullet timer expires, or when input is complete, the pet that's on turn is told to execute the moves that were entered by a TRICK_LISTENER_UNBLOCK message. This tells the pet to start processing its trick queue. END_OF_TRICK_SEQUENCE is pushed to the end of the pet's tricklist and will be echoed once those tricks have been completed. If the completion of the tricks takes too long, it times out and sends an EXCEPTION_DRAW to the exceptionListener FSM. Upon timely execution of the tricks, the next turn is scheduled with a NEXT_TURN message to the trickDuelMain FSM.

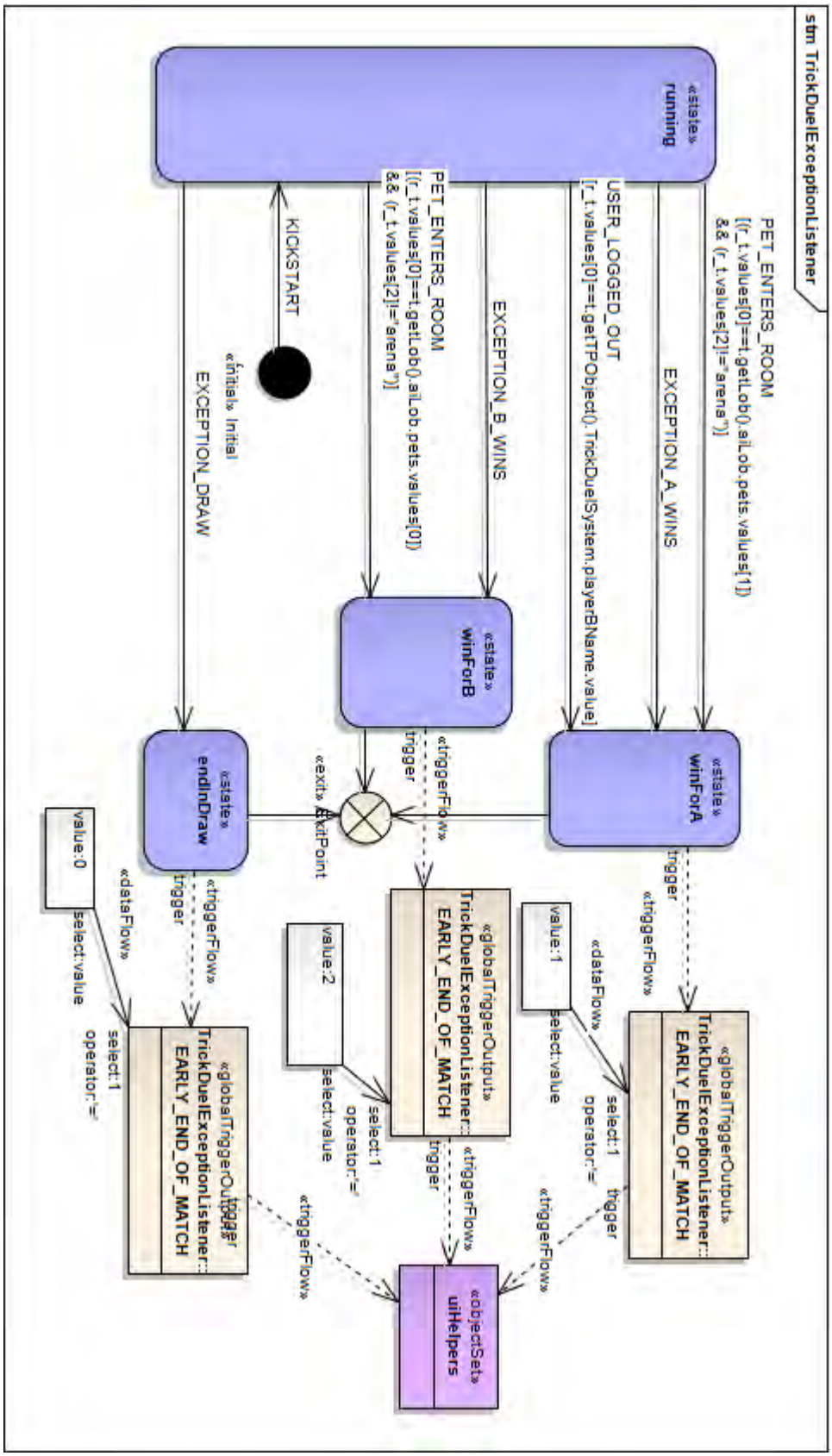

Illustration 10: Diagram TrickDuelExceptionListener

ExceptionListener (Illustration 10) listens to events that cause an assumption of the activity to be violated. These events can originate from outside (player logging out), or from the coordinator's

other state machines (e.g. The trick execution times out). Upon detecting such an event, it notifies the helpers that the game activity is over due to an exception. The helpers then send the pets home and display the final score screen, disabling the option to rematch.

4.4 Wrapup phase

When the game is finished, it should be left in a proper state. For the Trick Duel, a screen with the results is shown at the end of a game. If the game ended gracefully – that is without exceptions occurring – this screen also offers a rematch button. If both players press this button within 30 seconds, the game restarts. Otherwise, we clean up. This means releasing the participating pets from the game and sending them back to their owners' homes, and releasing the locks on the player starting another activity.

The coordinator tells the helpers that the match is over and passes the ID of winner along. The helpers initialize the score screen, show it and wait for a response. This response is passed back to the coordinator. This is an instance of *waiting for multiple unordered events (4.5.1).*

4.5 General

4.5.1 Wait for multiple unordered events

In general, events may happen in any order. Before the actual Trick Duel starts, both players must close their explanation windows. This can be done in any order. The diagram of Illustration 11 shows another FSM in the coordinator. This statemachine listens to closing events from either player. It is done in a seperate FSM because explanations can be closed at any time, and the main FSM is already in use for directing pets towards the arena.

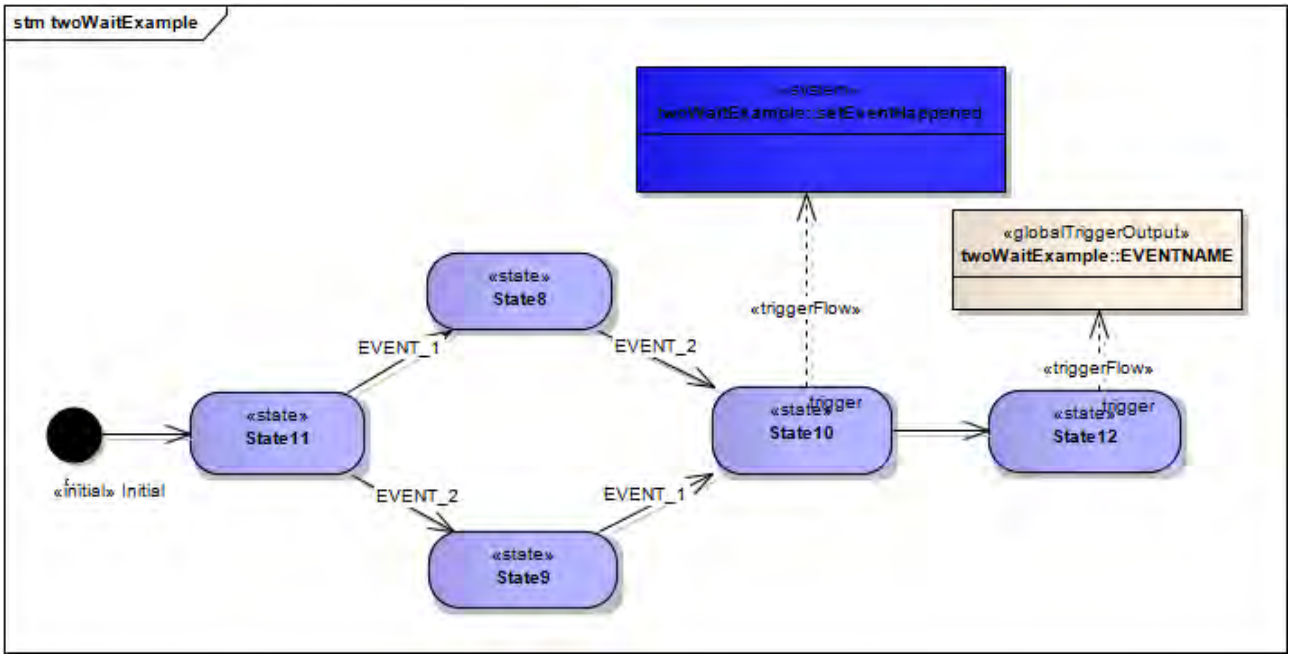

Illustration 11: Diagram TwoWaitExample

The possible traces for the events are included in a fairly straight-forward manner. There are two unordered events, and thus two possible traces; either player one closes before player two, or the other way around. When *State10* is reached, this means both events have been received.

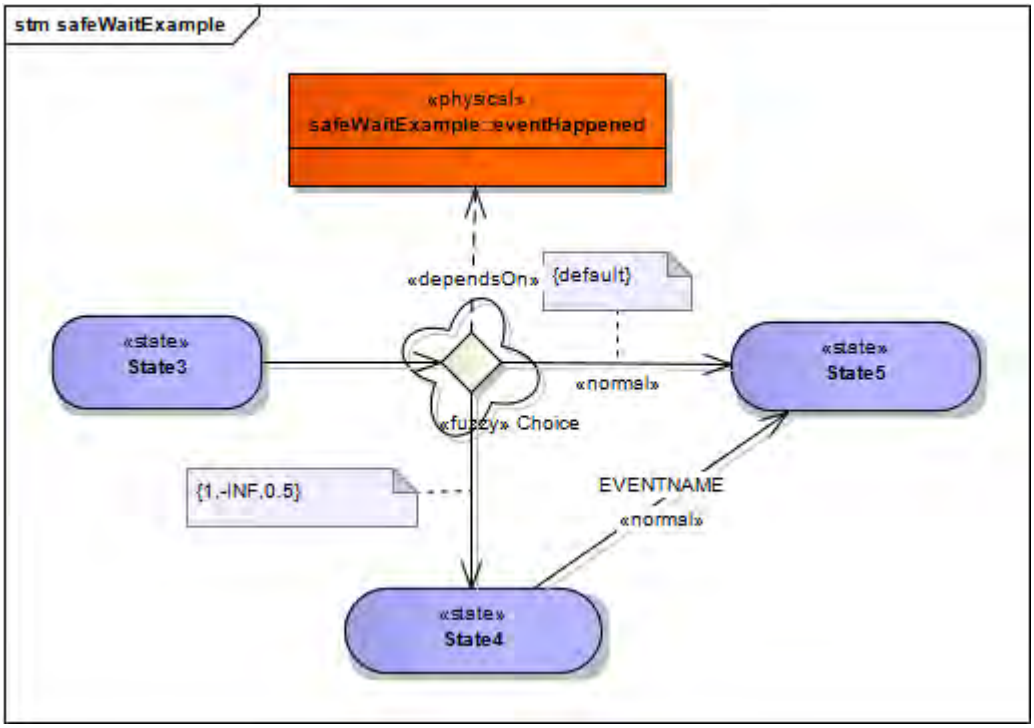

Illustration 12: Diagram SafeWaitExample

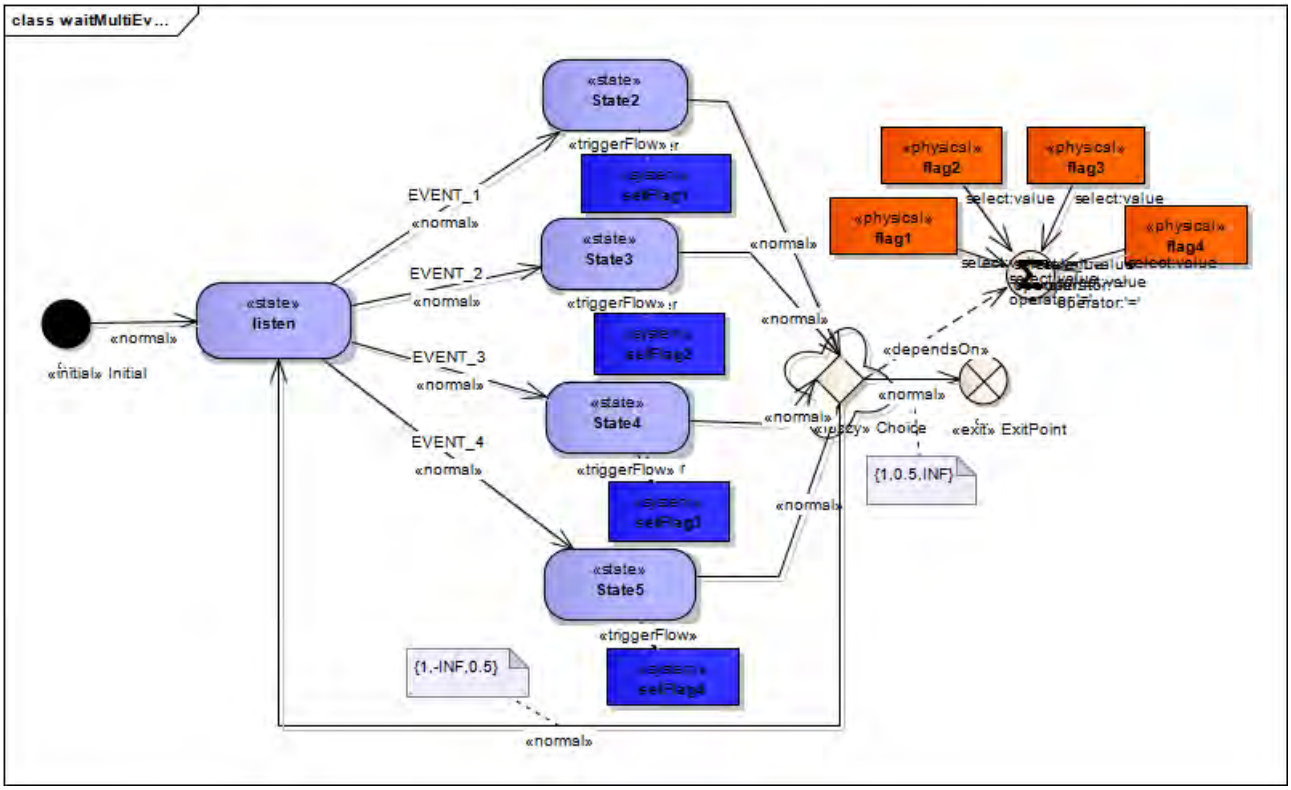

Illustration 13: Diagram waitMultiEvents

For demonstration purposes, I assume that twoWaitExample runs in parallel with the state machine that is interested in the completion of both events, and within the same object. To communicate, after both events have been received a flag is set and a trigger (here *EVENTNAME*) is sent. We set the flag to ensure that this event is not missed if the receiver state machine hasn't reached the waiting state yet. Illustration 12 shows the FSM used on the receiving side.

Since the number of possible traces is the factorial of the number of events, the naive approach does not scale well. So for more than 2 events, an alternative approach is required. A straight-forward approach (Illustration 13) keeps one flag per event. We assume that the state machine is in a listening state before any of the events occur. Once an event occurs, its corresponding flag is set, and we check whether all flags are now set. If they are all set, the flow continues, otherwise it returns to the listening state. In this case, the condition is (flag1 AND flag2 AND flag3 AND flag4).

4.5.2 Testing for a specific sender

When an object is communicating with several others, it will need to discriminate between senders. Since the name of the sender is passed along with the trigger, it can be compared to the expected name. This assumes that the expected name has been initialized at an earlier stage, which can be done with a construct as in Illustration 5.

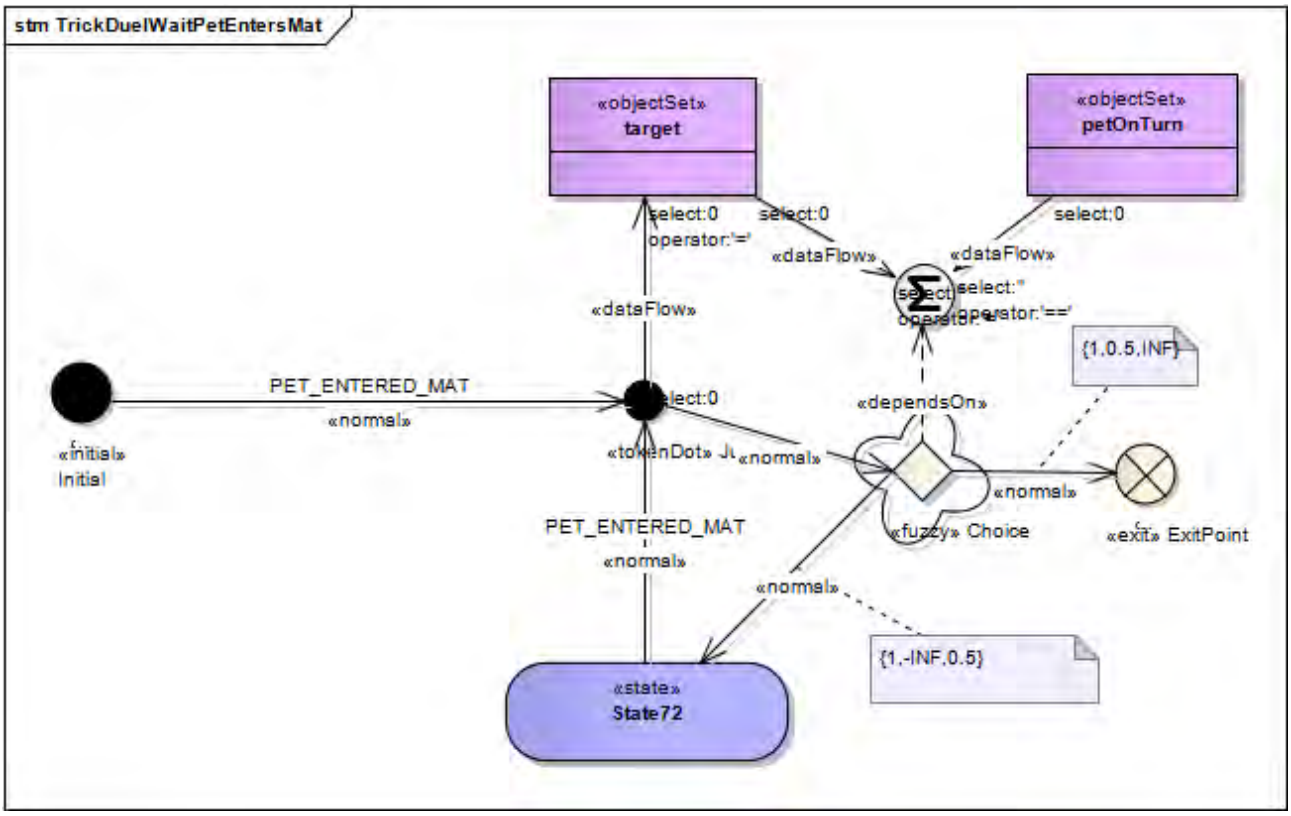

Illustration 14: Diagram TrickDuelWaitPetEntersMat

Alternatively, a JavaScript guard can be defined on the transition, but this requires knowledge of the internal structure of the generated JavaScript. The guard corresponding to the situation in Illustration 14 would look like

```
r_t.values[0]==t.getLob().aiLob.petOnTurn.values[0]
```

This guard reads the first value of the trigger's message (which always contains the ID of the sender) and compares it to the first value of the local array petOnTurn, which holds the expected ID.

Using the guard is more concise, but it breaks when the location or name of the variable is changed (for example due to restructuring of the variable structure or a change in the compiler.)

4.5.3 Managing remote UI elements.

In a multiplayer activity, the control of individual UI elements is delegated to helper objects for each participant. These helper objects receive information about high-level events (such as the start of a new turn), and then display or hide UI elements accordingly. This reduces communication between players at the cost of more local communication (but this is desirable, because it reduces the serverload), and avoids the need to acquire the IDs of all the UI elements on the remote side. Where appropriate, the helper relays communications by the UI elements back to the coordinator, for example the event that the rematch button was pressed on the remote computer. The basic principle is shown in Illustration 15. Local UI elements are also updated through the helper.

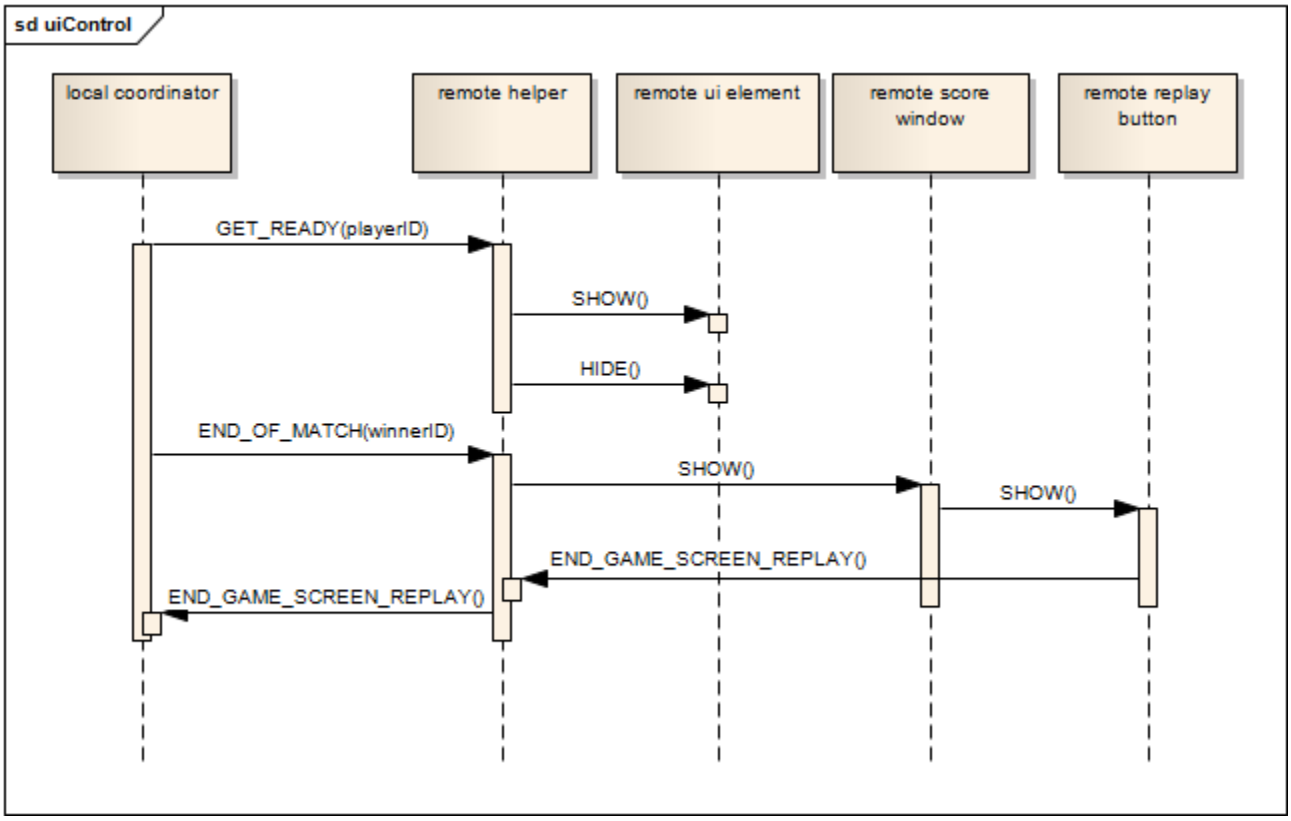

Illustration 15

4.5.4 Synchronized actions

Having multiple objects perform synchronized actions is easily accomplished by sending some signal when all objects are in a wait state, and having all objects respond to it. Thus the problem reduces to determining when the signal can be sent, and where it should originate.

Two-object synchronization occurs in the Say Hello case. The host pet signals when it has reached the waiting state and the visiting pet sends the synchronization signal when it's also there and it knows that the host is.

In the more general case of *N* participating objects, let each object broadcast I_AM_HERE when it arrives in the waiting state. Each of these messages can be counted (4.5.1), also adding the ID of the sender to an object set (A.4). When the set size equals *N*, we are ready to send a trigger (e.g. SYNC_COMPLETE) to all objects in the list. Keeping track of arrivals can be done either by a coordinator object, or decentralized by the pets themselves. In the latter case, only respond to the sync-trigger once.

4.6 Non-implemented case: Coin Search

One of the example cases was the Coin Search, where coins are hidden in (under) the floor and can be discovered by the player's pets. When a player has multiple pets, this gives an opportunity to intelligently divide the amount of labor. This use case was not implemented, but thought was given on the possible design.

A similar problem was covered in [5], set in the context of floor cleaning. In that situation, the dimensions of the rectangular floor were initially unknown, so the first phase determined the dimensions. Then, the floor is divided into quadrants; the cleaning of a quadrant is a task. Robots broadcast their intentions to perform a certain task, this helps in avoiding duplication of work. They also communicate when a task is finished (though they may be wrong). Thus, robots have awareness of what tasks are already taken and who are performing them. They have a degree of impatience and may decide to start doing a task that is already being performed by a different robot. Conversely, a robot that's currently performing a task may choose to abandon it when it finds it's taking too long.

Translating this to Tailplanet, the following processes can be distinguished:

- Determining the dimensions of the floor is not required. Floors have known dimensions. This also means that the floor can be divided into regions (quadrants or otherwise) at the very start.
- Keeping track of tasks that are already being done. This is difficult to model in our system, probably requiring multiple arrays and a communication FSM to keep track of all the information.
- Selecting a task (i.e. quadrant). This be fairly straight-forward; if the pet is not currently performing a task, iterate over existing tasks until one is found that's not complete and select that one. Prioritize tasks that no pet is currently doing.
- Performing the task. This involves implementing some behavior that systematically visits every cell of the quadrant, playing some animations at each of the cells and logging internally that you have visited that cell. Tailplanet currently has no support (method of modeling) movement to arbitrary locations, but if that is added, this part should be straightforward.

It might be worthwhile to keep track of all the information centrally in a coordinator object. The coordinator acts as a shared memory that can be queried by the pets using triggers. Typical queries would be: “Is this region already being processed?”, “Who is using it?” “Give me a random task”.

This seems to be a special case of shared use of resource. Each region would be used by at most one pet. However, this is a soft constraint, since agents may select a task that is already taken.

5. Evaluation of implementation

In this section I will describe the actual behavior of the implementation, and determine how well it performs according to technical (5.1) and subjective (5.2) criteria.

5.1 Technical criteria

5.1.1 Robustness

Robustness is reliability in the face of exceptional events. In the case of the Trick Duel, exceptional events include a player logging out, a pet leaving the arena while the game is ongoing, and any participating object not responding to messages. The typical way to handle this is to abort the activity. Players can then decide to try again. Crashes are unacceptable, as is getting into a game state from which a player cannot continue. Below we formulate a number of test cases; each line states the steps taken to reproduce an exceptional case, followed by the resulting behavior.
As before, player A is the inviting player and player B the invitee.

Invitation phase

- A invites, A logs out, B does nothing: *Player B times out and returns to a proper state.*
- A invites, A logs out, B accepts: *After a while it shows a dialog that the game will not start.*
- A invites, B logs out: *Invitation is canceled.*

Preparation phase

- Both accept, A logs while his pet is moving to the arena: *Player B is declared winner as expected.*
- Both accept, A logs out while B's pet is moving to the arena: *Player B is declared the winner, but pet ended up staying in the arena. This is not what was intended, but it is recoverable on the player side.*
- Both accept, A sends his pet back home before B arrives: *Player B is declared the winner, but his pet stays in the arena. The pet only moves home if it hears the game end while it's in a game-state. However, it doesn't enter the game-state until both pets have arrived, so the trigger is discarded.*
- Both accept, B calls back his pet once it starts moving to the arena: *This causes the preparation phase to time out, and player A is declared the winner.*

Execution phase

- Both pets in the arena, A's pet's turn, A logs out: *Player B is declared the winner, his pet teleports to his livingroom's doormat.*
- Both pets in the arena, B's pet's turn, B logs out: *Player A is declared the winner, his pet moves back home normally.*
- Either pet is called away during the execution phase: *the other player wins and the remaining pet moves back home.*

Wrapup phase

- Either player logs out during the final score screen: *the other player wins. At the moment, this is not deemed to be a large problem. It does mean that a player will need to explicitly stop playing the game to 'commit' his victory. When it turns out players routinely log out at this point, a solution will need to be found.*

Robustness of Say Hello

- Visitor logs out directly after arriving: *host pet is not affected.*
- Visitor logs out when the pets are turned toward each other: *host pet stares for a while, then continues with normal behavior.*

5.2 Subjective criteria

5.2.1 Ease of use

Are the diagrams simple?
Some of the presented diagrams were modeled to be the simplest possible illustration of the concept. As for the diagrams that were taken literally from the implementation, they should indicate the typical amount of complexity one can expect in practice.

How well do they apply to different situations?
The phases that were presented are a reasonable framework to build an activity around. To create a new activity, the designer needs to determine which information to gather in the invitation phase. For the Trick Duel example, the information gathered was the names of the players and the ID's of their pets. For a different activity, it might be that the invitation phase is more elaborate, allowing the challenger to put additional settings for the activity (for example, enable or disable certain rules or set a difficulty level). This functionality can be easily added by passing the additional information when asking for the remote player's consent.

The preparation phase is fairly straight-forward in a coordinator. Sending pets to a new location was already covered (section 4.2). Preparation can also involve destroying, creating and initializing objects, all of which are easy to model. The implementation can be a linear state machine, potentially using some handshakes for spawning and initializing.

The main differences between the activities will be expressed in the execution phase. My starting point with the Trick Duel was to identify substates within the game, which turned out to be *reading-input*, *performing-moves* and *switching-turns*. Then I determined the events that cause one substate to transition into another, such as “input complete” or “execution complete”.

Suppose we wanted to implement a real-time ball game instead. In this case, substates might be *game-ongoing*, *goal-scored*, *game-complete,* with transitions based on time elapsed and the location of the ball. Each of these states can have associated entry actions to inform objects that their behavior should change. For example, upon entry of goal-scored, this fact may be broadcast to all pets, which move away from the goal. At the same time, the pet that was designated keeper can be told to retrieve the ball and bring it back into play.

The wrapup phase is similar to the preparation phase and not too difficult.

In conclusion: while different activities may have different needs, the given diagrams and approaches can be easily adapted to fit the new requirements.

5.2.2 Quality of behavior

What is the resulting behavior?
For the most part, the Trick Duel works as intended. After the players coordinate the invitation, both pets gather at the designated location. The pets get on the mat when it's their turn, and vacate it on

the other's turn. One issue with the implementation is that pets leave the mat at the end of their turn, but they may in fact have two turns in a row when a new round starts. This behavior may be improved by keeping track of which pet is currently occupying the mat, and replacing the logic of “At the end of the turn, leave the mat” by “At the start of the turn , leave the mat if you're not the active pet.”. Another possible improvement is in the preparation phase; it could be sped up a little bit by sending both pets to the arena at the same time, though the challenged pet would have to wait on his doormat until the host arrives.

In Say Hello, the goal was to have pets be greeted when they come to visit. The basic scenario involves two pets. The host pet is already present in the room, when a visitor arrives. What happens now is that the host pet will complete what it is doing. Then the pets turn towards each other and wave simultaneously. If the visitor has timed out by the time the host is ready (to improve robustness it will not wait forever), there will be no waving.

When there are multiple pets already present, the visitor selects one at random (the first to respond to its host-request). However, this pet might not be the first to be done with his current behavior, causing the visitor to wait unnecessarily long. A possible improvement would be to postpone the host-selection until one of the pets returns to the GOB, and choose that one to be the host. This should improve the speed and success rate of greeting.

The implementation was not made to support multiple pets arriving at the same time. If this occurs, one of the arrivals might be ignored. This might be improved by keeping a list of visitors, instead of just the single pet. When a host pet is ready, it could select a random visitor, verify that it still needs greeting and then greet it. Alternatively, one host could greet all visitors at the same time (4.5.4).

6. Comparison with literature

In [1], a framework for the design of state machines for concurrent activities is described. It is suggested that one starts by creating an event timeline. From this, attempt to identify event threads. Each of these threads is handled by a task.

In my solutions simultaneous events are either handled by separate state machines or handled sequentially by the same state machine. I deem either approach to be acceptable, but note that sequential handling has certain risks. For one, the developer must guarantee that the central node will continue to be reached for the entire lifetime of the object. The rate at which new triggers arrive must not systematically exceed the rate at which they are consumed. If these conditions are not met, the queue may grow without bounds.

[2] first posits a weak notion of agency as meaning hardware or software-based computer systems that are *autonomous*, *social*, *reactive* and *proactive*. By that definition, the pets are also agents. It then discusses possible internal workings of agents. In deliberative architectures, agents are typically said to have beliefs and intentions [5], using some form of theorem proving on a symbolic representation of the world and planning to generate appropriate actions.

The approach in Tailplanet is more akin to reactive architecture. Pets have no planning capabilities by default, but they use greedy objective-optimization; they only look one step ahead by default and each behavior is hard-coded into its decision state machines (though behavior may be stochastic). A notion of *commitments* and *plans* can in principle be implemented through flags that are set (causing the pet to 'remember' these commitments and taking them into account when deciding what to do next), with a hard-coded action corresponding to that commitment. This method is currently used for moving pets to different locations, but it is not the dominant mechanism for behavior control.

[3] discusses state machine patterns for use in telecommunication systems, providing patterns for lower-level behaviors as examples for translating CEFSM's into SDL. These patterns have good correspondence to the way we use HFSMs, ignoring syntactic differences. The paper also explains how state machines may be merged, but these methods seem only applicable to the most basic of state machines.

In [8], a few reusable patterns for the coordination of (software) agents are established. A recurring principle is to have agents register themselves at a third party object. The blackboard pattern can help with information sharing between agents (useful for the Coin Search case). The Trick Duel coordinator and the pets have a master-slave relation, as the coordinator has a global view of the task and tells the pets when to move on and off the mat, and what tricks to perform.

Finite state machines are a popular technique in AI, because they are easy to code and intuitive to understand. They form the basis for the behaviour of the monsters and weapons in Quake [12], the ghosts in Pacman and units in Warcraft and other RTS's, among many others.

Alex J. Champandard predicts that the limitations of Finite State Machines [9] will cause developers to move on to other techniques (scripting, behavior trees [11] and planning) and notes that this trend existed in his retrospective for 2010 [10]. For Tailplanet, the decision to use finite state machines has already been made and a complete switch of technology is not feasible. It is not uncommon for developers to enhance the 'standard' finite state machine framework with extensions to meet their own requirements. For example, Tailplanet's own framework implements a mechanism for one-step look-ahead goal oriented behavior. It also supports the execution of arbitrary JavaScript code (section A.3), which is useful in cases where the statemachine representation would be impossible (within the current system) or needlessly verbose. It would be instructive to see which improvements other developers have come up with, but this information does not seem to be readily available.

7. Conclusions and Recommendations

In this section I provide answers to the research questions (7.1) and guidelines for the implementation of new game activities (7.2). Finally, I reflect on the broader applicability of the developed theory and techniques in games beyond Tailplanet (7.3).

7.1 Revisiting the research questions

Original problem description was:

“What are the typical use cases in Tailplanet?”
We have given a small sample of the use cases for coordination that Tailplanet has or may get. Stand alone activities consist of invitation, preparation, execution and wrapup phases.

“What methods for coordination exist?”
Coordination can be centralized or decentralized. In the case of centralized coordination, we have demonstrated the use of a coordinator between participating pets. The Trick Duel uses the master-slave pattern. There are a few common coordination problems, such as simultaneous or ordered actions, resource sharing and work division.

For decentralized coordination, a mechanism for pets to find each other was introduced. However, the focus of my findings is on centralized coordination. Originally, we wanted to find a balance between autonomous behavior and centralized control, but this proved to be too difficult at this time, so it was decided that pets should enter a specialized game-state while participating in an activity. The game-state may be broken out of if one of the needs exceeds critical level, but this has to be modeled manually for each new game.

“How can a method be evaluated?”
To evaluate the methods, the Trick Duel and Say Hello use case were implemented and tested for robustness. Test cases were generated to determine the effect of exceptional events at various points in the game. An important thing to test for is the effect of a player logging out, because this exception cannot be prevented on the model side. It will cause objects that were running on that player's side to stop responding. Any communication between objects for different players should have a timeout behavior or listen directly to the event of a player logging out.

The use-case itself may present a natural measure. For example, the coin search can be evaluated by the amount of time it takes to uncover the entire area versus a baseline.

Another measure can be the amount of complexity of the diagram. Given identical behavior, the simpler solution should get precedence. The given diagrams and protocols are not complex for complexity's sake.

7.2 Guidelines for implementing a new game

In this section I will present considerations and guidelines for modeling a new game for TailPlanet, following the phases from section 3.2. I will use same framework as the Trick Duel, which used inviter and listener objects for the invitation phase and a coordinator object that handles the preparation, execution and wrapup phases. The workings of these objects were covered in section 4.

I recommend that models are planned on paper first and that implementation is started only when the design is complete and (apparently) bug-free.

Invitation phase

How many players are involved?

For two or more players, let one player be the host and the other player(s) the clients. Spawn a Inviter for the host and a Listener for each of the clients. To establish consent, clients need only communicate with the host, not with each other. The host will notify the clients about success or failure.

A game can also be autonomous, being spontaneously organized by one of the pets. The same principles apply, but the pets are queried about their willingness to play instead. This query can be handled by a communication state machine in the pets. The inviter functionality can either be implemented in the pet or the Inviter object can be modified to communicate with pets instead.

What kind of information will be needed during the game? (for example the names of the players, the ID's of their pets and settings for the game).

Gather this information and use it to initialize the coordinator, which can be spawned at the end of this phase. When gathering the information, it is preferable not to read it from objects directly, but to use triggers to ask for and send the information (Section A.1)

Preparation phase

What objects are used in this activity? (e.g. a ball, a scoreboard, UI elements). Let the coordinator spawn any temporary objects and set the UI elements to their initial values. Use a simple serial state machine for this.

Bring any involved pets to the correct location (Section 4.2) and ensure they are in their game-state. If there are asynchronous preparations, wait for these to complete and then continue to the execution phase.

Execution phase

The state machine for the execution phase will probably have to be designed for each game, as there are a wide variety of possible activities with their own rules and interactions.

If possible, identify states and events within the game and build these into a state machine in the coordinator that reflects the state of the game at while it's running. It may be possible to distinguish independent event threads. Each event thread should have its own state machine that handles it.

If the processing of some events is not near instantaneous, consider doing the handling of these events in a state machine that uses a private queue (see A.2 and Illustration 8). The private queue will guarantee that every event is remembered and processed sequentially.

Wrapup phase

Consider using a room scoped trigger which kills the temporary objects, but is ignored by permanent objects. Send pets back to their homes (they can do this as the last step in their game flow). As the final step, remove the coordinator. This should happen automatically as it reaches the final state of its state machine.

Optimization

Are the statemachines becoming very complex? If so, consider splitting them up into composite states. Also do this when there are blocks that repeat themselves.

Are there a lot of cross-network triggers generated? Perhaps the overhead can be reduced by sending multiple pieces of information in the same trigger. It may also be possible that one trigger implies others (for example START_ROUND implies SHOW and HIDE for a UI element), so the dependent triggers can be generated locally. (see section 4.5.3, this also applies in general)

Apply simplifications where this is possible.

Validation

Finally, the models should be checked for correctness.

What assumptions are made in the execution of these statemachines? *Are these assumptions justified, and* w*hat would happen if they were violated*?

Correct any issues that you spot when answering these questions. A typical thing to test for is the sudden unresponsiveness of a target object. This can happen when its owner logs out or it moves outside the scope of the communication method that is used. At every point where the state machine waits for a remote response, there should be exception handling code, for example using a TIMEOUT transition. Decide the correct behavior for each exception and define test cases that deliberately cause exceptional events.

At this point, the models can be implemented. Run all the test cases and check if the expectations match the results. Also test the normal flow of execution, of course.

7.3 Application beyond TailPlanet

The implementations themselves were illustrated using Tailplanet's variant of HFSMs. Other companies will be using a wide variety of systems for their games, so the diagrams themselves may not be directly applicable. However, the theory of the phases is generic enough to be used in other games. I would assume that many game engines have support for message sending between objects in the game-world. If this is the case, then the objects and their interactions should be just as easily implementable in the other system.

Bibliography

[1] Bo I. Sandén – A Design Pattern for State Machines and Concurrent Activities

[2] Michael Wooldridge and Nicholas R. Jennings – Intelligent Agents Theory and Practice (1995)

[3] YoungJoon Byun, Beverly A. Sanders and Chang-Sup Keum – Design Patterns of Communicating Extended Finite State Machines in SDL

[4] Nwana, H.S., L. Lee, and N.R. Jennings, 1996. Co-ordination in software agent systems. BT Technology Journal, Vol. 14, No. 4, pp. 79-88.

[5] Anand S. Rao and Michael P. Georgeff , BDI Agents: From Theory to Practice

[6] Company website: http://www.connecteddreams.com

[7] Game website: http://www.tailplanet.com

[8] Dwight Deugo, Michael Weiss, and Elizabeth Kendall, Reusable patterns for Agent coordination

[9] Champandard, A.J., 10 Reasons the age of Finite State Machines is over. http://aigamedev.com/open/article/fsm-age-is-over/

[10] Champandard, A.J, This Year in Game AI: Analysis, Trends from 2010 and Predictions for 2011, http://aigamedev.com/open/editorial/2010-retrospective/

[11] Björn Knafla, 2011, Introduction to behavior trees http://www.altdevblogaday.com/2011/02/24/introduction-to-behavior-trees/

[12] Jason Brownlee, 2002, Finite state machines,

http://ai-depot.com/FiniteStateMachines/FSM.html

Appendix A. Documentation of new constructs

A.1 Message passing and reading

The functionality of triggers was extended with the possibility to add parameters to the message.

To set parameters, connect a variable to a triggerOutputNode with a <<dataflow arrow>> and pick a numerical index >0 as *targetSelect*. (Illustration 16)

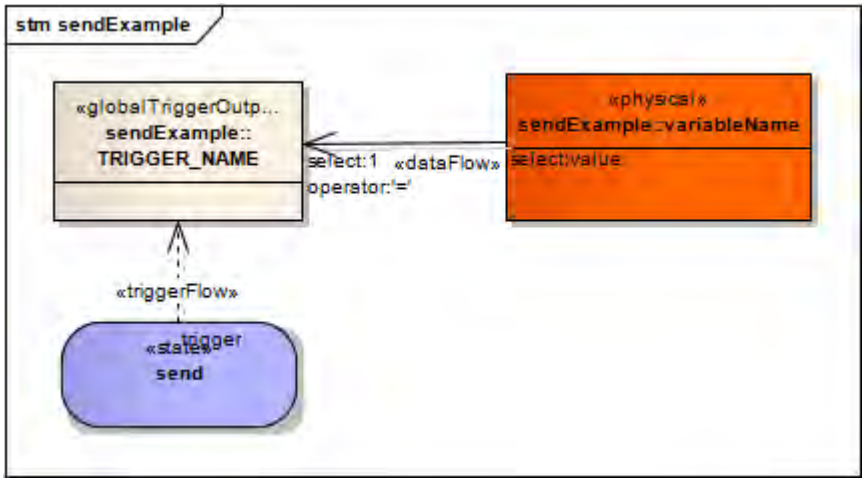

Illustration 16: Sending variables with triggers

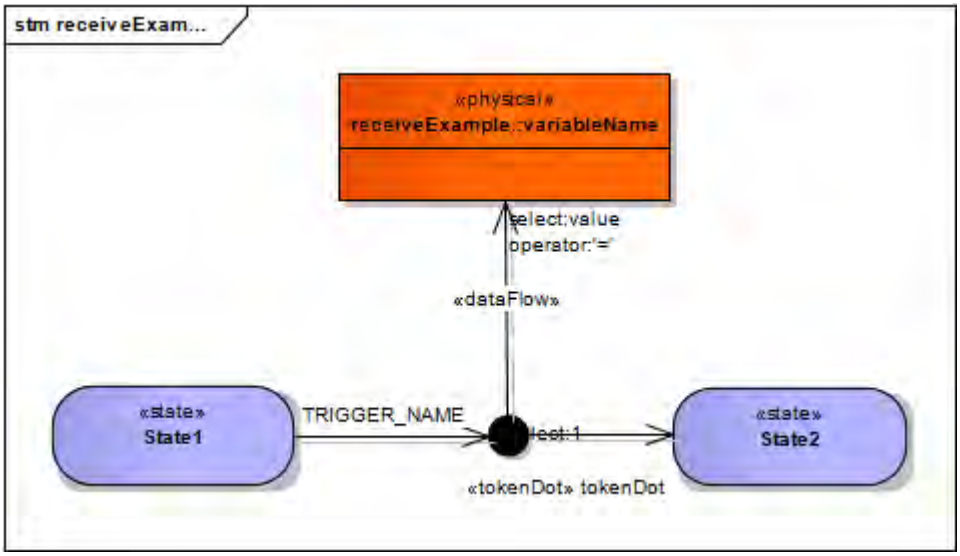

Illustration 17: Receiving variables from triggers

A tokendot provides access to the values of an incoming trigger. To read parameters from a trigger, put a tokendot after the transition, connect this tokendot with the next state with another transition. Draw a dataflow from the tokendot to the variable that you want to store the value in, and put the numerical index as *sourceSelect* (Illustration 17). Index 0 holds the ID of the sender.

The speed of the transition is taken from the transition with the trigger's name. A tokendot may have several incoming transitions, but only one outgoing transition. They may also be serially linked. Tokendots in self-transitions work (the state is not reentered, so entry actions are not executed again).

A.2 Private Queue

A private queue can be added to a leaf node. To do this, add the tag *ownQueue* with value 1 to this state.

A state with a private queue will read its triggers from that queue instead of the global queue. When the state is left, the remaining triggers are retained on the private queue (to be processed on the next arrival) A trigger that is consumed on the private queue is also consumed on the global queue, *this is not guaranteed the other way around*.

To avoid that a private queue grow without bounds, care must be taken to ensure that the state is visited periodically, or that the owning object is destroyed.

A.3 JavaScript Node

JavaScript nodes are intended to provide a simple means of inserting JavaScript into the generated code when there are is no simple alternative available. The content of the JavaScript node is copied verbatim into the appropriate place of the generated code. Depending on context, it can be an expression, or a block of statements.

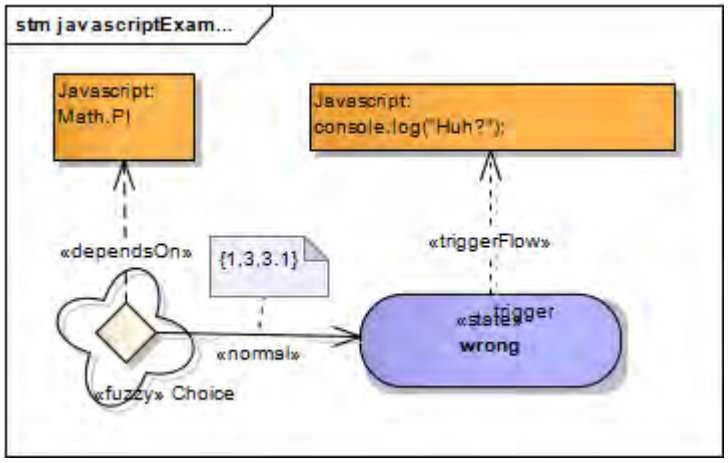

Illustration 18:Two contexts for JavaScript nodes.

A.4 Object Set

An Object Set in Enterprise Architect represents an array of values. These values can be accessed with the sourceSelect and targetSelect tags on dataflow arrows. A numerical value denotes a specific index as with tokendots. To refer to the entire array, use the keyword '*all*'. This replaces the standard behavior of operators by set operations, where '+' is union, '-' is difference and '=' is assignment. Triggers can have an object set as target. The default behavior converts the array values to object IDs and passes the trigger to each of these objects.