Generating Extras: Procedural AI with Statecharts

Christopher Dragert, Jörg Kienzle, Hans Vangheluwe, and Clark Verbrugge

McGill School of Computer Science: SOCS-TR-2011.1

***Abstract*—Populated and immersive game contexts require large numbers of minor, background characters to fill out the virtual environment. To limit game AI development effort, however, such characters are typically represented by very simplistic AI with either little difference between characters or only highly formulaic variations. Here we describe a complete workflow and framework for easily designing, generating and incorporating multiple, interesting game AIs. Our approach uses high-level, visual *Statechart* models to represent behaviour in a modular form; this allows for not only simplistic, parameter-based variation in AI design, but also permits more complex structure-based approaches. We demonstrate our technique by applying it to the task of generating a large number of individual AIs for computer-controlled squirrels within the Mammoth [1] framework for game research. Rapid development and easy deployment of AIs allow us to create a wide variety of interesting AIs, greatly improving the sense of immersion in a virtual environment.**

***Index Terms*—Game AI, Modelling, Statecharts**

I. Introduction

A well-designed artificial intelligence (AI) in a computer game can take significant development effort. Complex AI design is typically focused on opponents or other *non-player characters* (NPCs) that are central to game play. Many games, however, take place in populated areas, and thus also contain a large number of background characters, or *extras.* Although extras have limited to no direct impact on game mechanics, reasonably intelligent extras are necessary to provide a realistic visual milieu and ensure a good sense of immersion. Individual, manual AI design for extras is not economically feasible; to limit development costs, these characters are often controlled using a limited set of template behaviours along with simple randomization of timings or probabilities.

In this work we address the problem of facilitating and procedurally generating interesting yet variable AI behaviour for minor game characters, as a means of improving player immersion. Our approach constructs game AIs based on a high-level *Statechart* representation, allowing a designer to model in a high-level form. Additionally, we provide a complete development path from a visual formalism into actual game code, enabling in-game integration and experimentation. Modularity inherent within the Statechart approach is preserved in our design, allowing for easy perturbation of AI designs at different levels. We generate multiple AIs using basic randomization of model parameters, as well as higher-level variation of the Statechart behaviour based on capturing and redirecting events or on rewriting Statechart structure. These higher-level techniques can provide more interesting, semantic changes in behaviour, and an automated approach allows for rapid prototyping and batch generation.

To illustrate the power of our approach, we show in the following sections how we modelled the AI of a squirrel NPC. A squirrel is a good example of a game character that is not essential to the game's storyline, but through its behaviour creates a feeling of immersion for the players that explore the game world. Animals such as squirrels abstractly follow a similar set of daily activities and potential reactions, but also exhibit unique and individual behaviours in practice.

Following the stratified design of Kienzle et al. [2], we develop first a basic AI composed of a set of modular, high-level behavioural components that implements a core set of squirrel behaviours. From this, using both high and low-level perturbation techniques we automatically generate a variety of AIs that produce similar, but interestingly different squirrel behaviours. Importantly, and despite the variation, our randomized squirrel AIs maintain an overall semantic consistency, always exhibiting core squirrel activities and reactions.

The design we explore applies to any game or simulation context in which rapid generation of varying, but interesting AI contexts may be useful. Specific contributions of our work include:

- We describe a Statechart-based visual modelling framework and workflow suitable for producing game AI for NPCs. This allows for flexible, high-level design and analysis of game AI.
- Our design can function as procedural content generation. We show how to apply automatic or manual variation at different levels of abstraction in order to quickly produce individual AIs with interesting differences, but still well-defined and bounded behaviours.
- To validate our approach we develop a non-trivial simulation of squirrel behaviour within a 3D virtual environment. Using our techniques we are able to easily generate multiple and complex squirrel behaviours that provide a rich sense of immersion with minimal design costs.

In the next section we give essential background on our modelling formalisms. Section III describes our AI modelling strategy in detail, and in Section IV we describe our resulting, multi-tiered approach to content generation. Implementation and experimental validation are presented in Sections V and VI, followed by a review of related work in Section VII, and future work and conclusions in Section VIII.

II. Background

A. Statecharts and UML

Since the main purpose of the AI models is to define reactions to game events, an event-based formalism seems to be the most natural choice. We extend State Chart XML (SCXML) [3], a hybrid of *Rhapsody Statecharts* [4] and

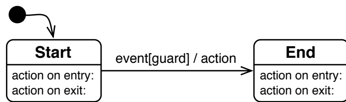

Fig. 1. Statecharts Basic Transition

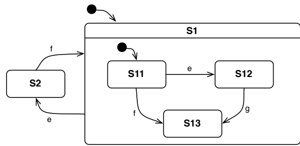

Fig. 2. Hierarchy in Statecharts

STATEMATE Statecharts [5] for our experiments, combining State Diagrams and Class Diagrams.

Statecharts were introduced by David Harel in 1987 [6] as a formalism for visual modelling of the behaviour of reactive systems. A full definition of the STATEMATE semantics of Statecharts was published in 1996 [5]. More recently, with the introduction of UML 2.0, the Rhapsody semantics as described in [4] is more tuned to the modelling of software systems.

At the heart of the Statecharts formalism is the notion of discrete *states* and the transitions between them. Statecharts are a discrete-event formalism, meaning it takes a timed sequence of discrete *events* as inputs and produces a timed sequence of discrete events as output. Internally, the system transitions between discrete states due to either external or internal events. In Figure 1, a simple model is shown with two states: `start` and `end`. The small arrow pointing to `start` denotes that state as the *default* initial state. If the system is in state `start` and it receives *event* while condition `guard` evaluates to `true`, the transition to state `end` is taken and the side-effect `action` is executed. Additionally, `entry/exit actions` are executed whenever a state is entered/exited. Each of the parts of the `event[guard]/action` transition trigger are optional.

Figure 2 shows a composite state `s1` with several nested states. Initially, the system will be in state `s11` as at the top level, `s1` is the default state and within `s1`, `s11` is the default state. To understand the nesting, when in a state such as `s11`, upon arrival of an event such as `f`, an outgoing transition is looked for which is triggered by event `f`. This lookup is performed traversing all nested states, from the inside outwards (this is the Rhapsody semantics). The first matching transition is taken. This approach keeps the semantics deterministic despite the seemingly conflicting `f` trigger on transitions to both `s13` and `s2`. When in state `s12`, there is no conflict and event `f` will take the system to state `s2`. When in state `s2`, event `e` will take the system to state `s1`. As the latter is a composite state, the system will transition (after executing `s1`'s entry action) to the `s11`, the default state of `s1`.

In addition to hierarchy, Statecharts add orthogonal components and broadcast communication to state automata. Figure 3 shows orthogonal components (or *and-states*) `ocA` and `ocB` as separated by a dashed line. This denotes that the system will simultaneously be in exactly one of the *or-states* of *each* of the orthogonal components. As such, this is a short-hand notation for the unordered cartesian product of state sets. All orthogonal

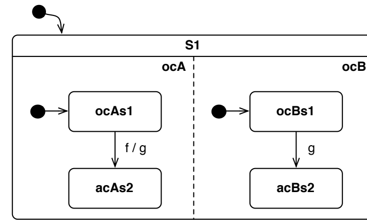

Fig. 3. Orthogonal Components in Statecharts

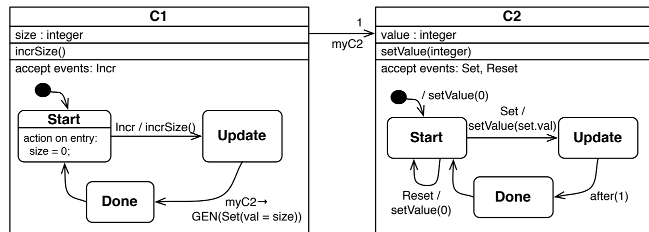

Fig. 4. Modelling Structure and Behaviour

components react to external events. Furthermore, events such as `g`, which is output when the external event `f` is received while `ocA` is in state `ocAs1`, are *broadcast* and are in particular sensed by other orthogonal components.

The most interesting feature of Rhapsody Statecharts is that it allows for a combined description of structure and behaviour of objects. This is achieved by adding Statechart behaviour descriptions to UML Class Diagrams as shown in Figure 4. The behaviour of individual objects (class instances) is described by the class' Statechart. For conceptual clarity we require that methods in a class have local effects, and can only change the object's attributes. All external effects must be modeled in the Statechart. This allows for a clean separation of externally-visible, reactive, timed behaviour from internal (computation) details. Objects communicate by means of a `GEN` action which sends an event to a target object as shown in Figure 4 (`myC2->GEN(Set(size=2))`). Events can be handeled asynchronously or synchronously (in which case they are similar to remote method calls). We will mostly use asynchronous message passing. To support concurrency between objects, our Statecharts compiler will give each object an event queue. All object queues will be processed fairly. Note that in the rest of this article, we will use the more common UML notation where structure and behaviour (Class and Statechart) are shown separately, but connected via an association with stereotype `<<behaviour>>`.

B. Mammoth

Mammoth [1] is a Massively Multiplayer Online Game (MMOG) research framework. It provides an implementation platform for academic research related to multiplayer and MMOGs in the fields of distributed systems, fault tolerance, databases, networking, concurrency, but also artificial intelligence, content generation, and software engineering in general.

In Mammoth, players take control of a game character called an *avatar*. A game session consists of moving around in a vir-

tual world and interacting with the environment by executing actions. Basic building blocks of such actions include moving the avatar, picking up or dropping items, or communicating with other players.

In order to allow researchers to easily conduct experiments, Mammoth has been designed as a collection of collaborating components that each provide a distinct set of services. The components interact with each other through two types of well-defined interfaces, engines and managers. The most important component in the context of this work is the NPC Manager, which takes care of associating AI behaviour with controllable entities (the NPCs). At run-time, the NPC Manager passes relevant game information to the AI, and provides an interface for the AI to trigger game actions within the virtual world. The current NPC manager of Mammoth supports game AI written in Java, and game AI modelled using the approach we present in this paper in section V.

III. Game AI Modelling Formalism

In games or simulations, a character receives information about the environment, decides on a course of action, and then tries to execute it. For instance, a character might observe an obstacle, decide to turn left, then execute that action. Our modelling formalism logically divides an AI into components based upon this control-inspired philosophy. All components are modelled as a Statechart with an associated class.

Components that learn about the game-state are called *sensors*, while components that alter the game-state are classified as *actuators*. Together, they comprise the interface between the game and the AI. Only the sensors have the ability to learn about the current game-state, and only actuators have the ability to change it. This creates a clear distinction; the other components can only learn about the game-state through events, and can only change the game-state by issuing events.

Between these are a hierarchy of components that transform sensing input to actuated output. The main form of communication between components is the asynchronous sending and receiving of events. This yields a loose coupling between components and hence makes reconfiguration, reuse, and creation of variations of behaviour easier.

We explicitly model the generation of significant events using Statecharts. The associated class has access to the state information needed to generate an event. This is done by inspecting the values of the attributes of the current class, or by looking at attributes of other classes associated by composition relationships.

The architecture of our formalism is described in Fig. 5. The first level contains *sensors*, which allow the character to observe the environment and its own state, filtering the input and sending events to higher levels. The second level contains components that *analyze* or correlate the events from individual sensors, which might lead to the generation of further events. *Memorizer* components keep track of the observed state. Conceptually at the highest level of abstraction, *strategic deciders* act to decide on a strategy for the character based on the current state and memory.

From here, the level of abstraction decreases. The *tactical deciders* plan how to best pursue strategies sent out by the strategic deciders. The *executors* then translate tactical decisions to low-level commands according to the constraints imposed by the game. *Coordinator* components understand the interplay of actuators and might refine the low-level commands further. Finally, the *actuators* perform the desired action.

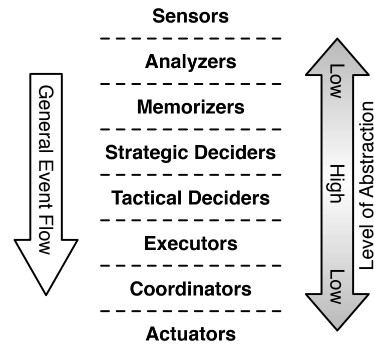

Fig. 5. Our AI Model Architecture

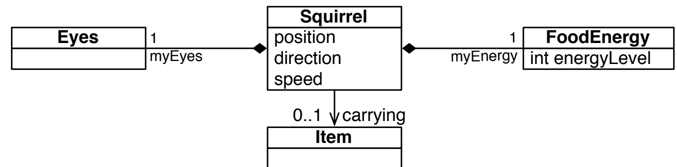

Fig. 6. Modelling the State of a Squirrel with Class Diagrams

A. Modelling the State of a Squirrel

As natural creatures, squirrels have a wide variety of possible behaviours and internal states, and may of course be modelled to arbitrary detail. For game purposes, a high level of abstraction is sufficient. We give each squirrel a physical size, approximated by a bounding rectangle, a position, and an energy level. Energy has an initial high value that gradually decreases as the squirrel moves about, but that can be restored by eating acorns. A squirrel can also carry a game item, such as an acorn.

The above mentioned state of a squirrel can naturally be modelled using class diagrams as shown in Fig. 6. Each sensor can be modelled as a stand-alone class. The composition association is then used to connect the different components together to form the complete state of a particular squirrel.

The advantage of using hierarchical composition is easy to see: depending on the particular game environment, different models of squirrels, for example ones with additional sensors such as *ears*, can easily be constructed by adding new components.

B. Sensors – Generating Important Game Events

Sensors receive basic game and squirrel state data, translating individual state observations to higher level events of interest. This reduces the complexity of constructing higher level, goal-directed behaviours, such as seeking food or fleeing from players.

A simple example is shown in Fig. 7. The *FoodEnergy* class encapsulates an attribute that stores the current energy level

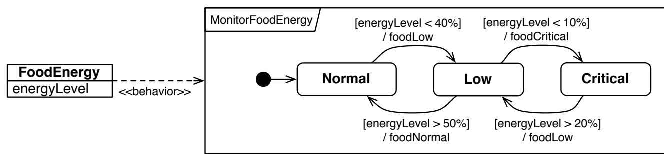

Fig. 7. Generating *FoodNormal*, *FoodLow* and *FoodCritical* Events

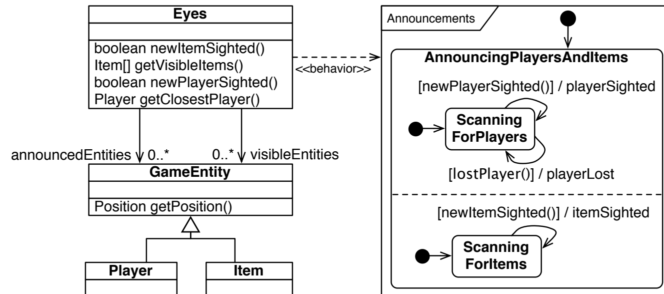

Fig. 8. Generating Events based on Visible Game Entities

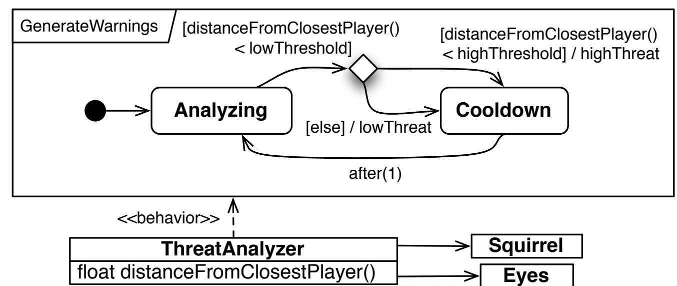

Fig. 9. Generating Events based on the State of Several Components

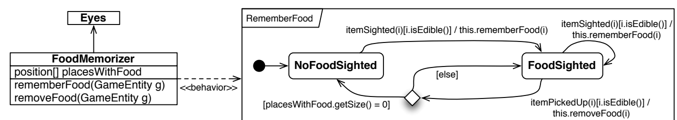

Fig. 10. Remembering the Position of Sighted Food

of the squirrel. While energy is essential for the squirrel to function, the exact level is not of great importance. Hence we abstract from the continuous level to three discrete states, *FoodNormal*, *FoodLow* and *FoodCritical*. Only when energy is low should the squirrel take appropriate measures. We model the generation of a *foodLow* event by attaching a Statechart to the *FoodEnergy* class, triggered when the energy level drops below the low energy threshold.

A more complicated example is shown in Fig. 8. In this case, the *Eyes* component wants to signal *playerSighted* and *playerLost* events when players enter or leave the squirrel's line of sight. This behaviour is described in the first orthogonal component of the Statechart *Announcements*. Analyzing game data to properly detect players is an operation requiring a small computation. It is modelled as a simple operation attached to the *Eyes* class. The attached Statechart attached can use these operations to trigger the transition that sends the desired events. The orthogonal *AnnounceItems* component performs similar event generation for detected game items.

C. Analyzers – Correlating Sensor Events

Some significant events can only be generated based on the state of several sensors. For instance, to determine if it makes sense to eat, information about current energy (which must be low enough for food to not get wasted) and information about what the squirrel is carrying (which must be something edible) is needed. The *EatAnalyzer* component observes the states of the energy sensor and the squirrel and generates *readyToEat* events when appropriate.

Another example of an analyzer component is the *ThreatAnalyzer*, which is represented in Fig. 9. While in the *Analyzing* state, if the distance to the closest player is smaller than a given distance, then the *highThreat* or *lowThreat* events are generated and the component moves to a *Cooldown* state where no new events are generated. After one second, the component moves back to the *Analyzing* state. As a result, threat events are generated every second until there are no players in the vicinity of the squirrel anymore.

D. Memorizers – Modelling Memory

Aside from reacting to current events, squirrels must also make decisions based on previous observations. In order to remember interesting states or events for future strategical decisions, we need to add state to the models that act as the squirrel's "memory".

Occurrences of events can be remembered using boolean or enumeration fields, or states in a Statechart. An example of the latter is shown in Fig. 10, which depicts a *FoodMemorizer* component. When an edible item is sighted, the Statechart transitions into a *FoodSighted* state and stores the position of the discovered food in the *placesWithFood* array. When a food item is picked up (signalled by the *itemPickedUp* event), the position is removed from the array.

Remembering complex state, for instance geographical information, is less trivial, and usually requires the construction of an elaborate data structure that stores the state to be remembered in an easy-to-query form. This could be done, for instance, by an *ObstacleMap* component, which listens to *ObstacleDetected* events sent by the eyes and then updates the map data structure accordingly.

E. Strategic Deciders – Deciding on a High-Level Goal

Now provided with knowledge of the game-state, it is possible to model the high level strategy of the squirrel. An example *Brain* component is depicted in Fig. 11. At the highest level of abstraction, the squirrel brain switches between different operating *modes* based on events. It starts in the *Safe* state, which is a concurrent state with two compartments. In the left compartment, the squirrel starts in the *Wandering* mode, and switches to *LookingForFood* mode if energy becomes low. In the right compartment the model states that whenever the squirrel is ready to eat, it will eat. If at any point in time the squirrel is exposed to a high threat, then wandering, looking for food, and eating activities are interrupted as the brain switches to fleeing mode. A low threat event on the other hand only interrupts wandering, e.g., a squirrel currently looking for food is willing to take a small risk to reach its goal.

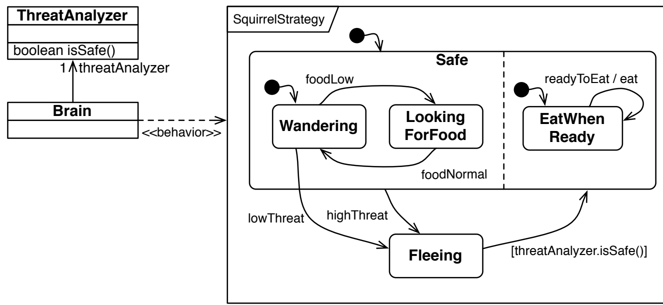

Fig. 11. The Squirrel Brain Strategy

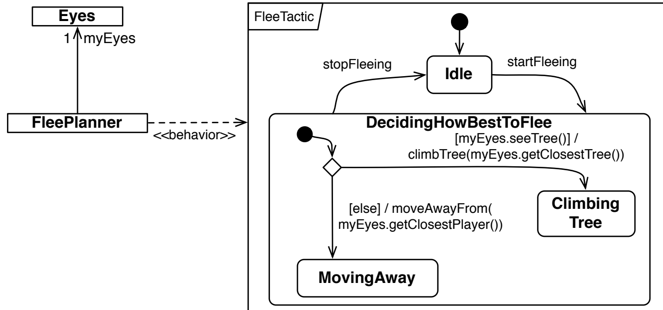

Fig. 12. Planning how to Flee

Mode changes are announced by sending corresponding events to the tactical deciders (or planners). To make this happen, each of the states of the brain has enter and exit actions defined: when entering the *Wandering* state, the *startWander* event is sent; when exiting the *Wandering* state, the *stopWander* event is sent; and so on for all other brain modes.

F. Tactical Deciders – Planning how to Achieve the Goal

High-level goals sent by the strategic decider must be translated into lower-level commands understandable by the different actuators of the squirrel. Translation is not trivial, since it can require complex tactical planning decisions to be made. This can include consideration of the game history, learned by consulting the memorizers.

Each mode in the squirrel brain has a corresponding *planner* component. Fig. 12 illustrates how the *FleePlanner* decides on a flee tactic: when the squirrel wants to flee, it checks if there is a tree nearby. If yes, the *climbTree* event is sent. Otherwise, the best tactic is to move in the opposite direction of the threat. This is achieved by sending a *moveAwayFrom* event.

Another tactical decider for our squirrel is the *LookingForFoodPlanner* component. It first looks for food using information provided by the *Eyes*. If food is visible, it is picked up. Otherwise, the *FoodMemorizer* is consulted to obtain positions of previously seen food. The squirrel then visits these positions hoping that the food is still there. If no food has been sighted in the past, the squirrel wanders around randomly, hoping to eventually find some.

G. Executors – Mapping the Decisions to Actuator Commands

Executors map the decisions of the tactical deciders to events that the actuators can understand. This mapping is constrained by game rules; if, for instance, an AI controls a car, a *Steering* executor must translate waypoint events generated by a planner component into events that the *MotorControl* and *SteeringWheelControl* actuator components understand.

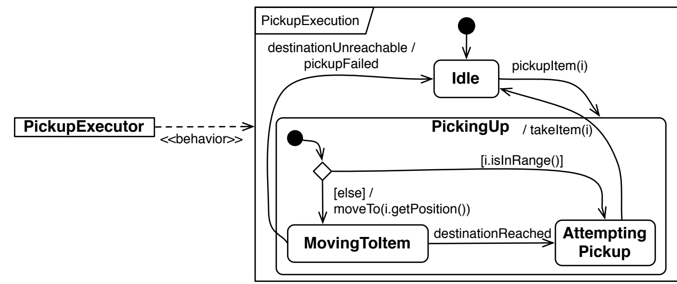

Fig. 13. Executing an Item Pickup

In the case of our squirrel, we have a *PickupExecutor* component that knows how to interpret the *pickupItem* events generated by planners such as the *LookingForFoodPlanner*. The model is shown in Fig. 13. Another example executor is the *moveAwayFrom* executor, which, given a position to move away from, generates *moveTo* events for the leg actuators.

H. Coordinators – Resolving Undesired Actuator Interactions

For modularity and composability reasons, executors individually map tactical decisions to actuator events. This mapping can result in inefficient and maybe even incorrect behaviour when the effects of actuator actions are correlated. In such a case it is important to add an additional *coordinator* component that deals with this issue.

For example, our squirrel might want to continuously look at a threatening player in order to always know about the exact position of the latter. Coupled with a simultaneous decision to move, this requires coordinating head and body rotations to ensure natural results. Our simple game environment does not model squirrels to this level of detail, so we do not include any coordinators of this type in our current design.

I. Actuators – Signalling the Action to the Game

At our level of abstraction, the squirrel actuators are very simple. A squirrel's legs can walk to a certain destination. The hands of the squirrel can take items and put them in the squirrel's mouth. The mouth can drop items and eat edible items. We suggest to model each actuator as a separate *Control* component.

Fig. 14 shows the *Legs* component, the actuator controlling the movement of the squirrel. The Statechart shows that, upon receipt of a *moveTo* event, the actuator moves into a *Moving* state. The associated class of the actuator then calls the appropriate internal method to cause the NPC to move within the game.

The *Legs* component in Fig. 14 is actually a special kind of actuator, a *feedback actuator*. Feedback actuators not only translate events from the Statechart world to game-environment-specific commands, they also provide feedback to the Statechart world on the outcome of a command. In other words, they are both actuators and sensors. In our case,

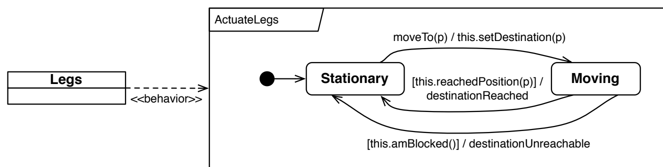

Fig. 14. The *Legs* Feedback Actuator

the *Legs* send *destinationReached* and *destinationUnreachable* events to provide feedback on the outcome of a move.

J. Squirrel AI Model Example Component Interaction

At run-time, communication among our components is done using events, and hence the individual components are only loosely coupled. Some communication must be synchronous, such as consulting a memorizer. This is accomplished through asynchronous events that are paired to have a call/callback.

Fig. 15 shows a possible sequencing of events triggered by the *Eyes* detecting a player. The *playerSighted* event causes the *ThreatAnalyzer* to calculate the distance between the squirrel and the sighted player and finally output a *lowThreat* event. This in turn causes the *Brain* to switch to *Fleeing* mode and generate a *startFleeing* event. The *FleePlanner* checks for trees nearby, but since there are none decides to flee by moving away from the player by generating a *moveAwayFrom* event. The *MoveAwayFrom* executor finally calculates an appropriate destination position based on the player position and the squirrel position, and generates the *moveTo* event that triggers the *Legs* to start moving in the desired direction.

IV. Content Generation

Content generation is the process of creating new elements to put into the game world. Rather than create truly novel AIs, our methodology seeks to enrich the gameplay experience by giving variation and personality to existing AIs. We present several methods that take AIs designed using the above described hierarchical Statechart model and generate variations. We show that our modifications are safe, in that they ensure semantic correctness of the resulting AI. The end goal is to create many different versions of an AI, each able to fill the original role, but with variation that add flavour and life.

Three approaches are employed. First, we describe how to employ parameter modification to create new AIs. Next, the benefits of a modular approach are discussed. Specifically, we look at how the addition, removal, or swapping of Statecharts can lead to new AI behaviour. Finally, rule-based variations of state charts are introduced, showing how a Statechart can be transformed to create a new behaviour while maintaining semantic correctness. In each section, we comment on how these changes must be managed to ensure validity of the generated AI.

A. Varying Parameter Values

Non-player characters typically have many properties that are defined when the character is instantiated. These can include properties such as maximum energy, movement speed, inventory size, and so on. Varying these parameters is a obvious method to create differentiated instances of an NPC. However modifying a finely tuned AI could easily lead to incorrect behaviour, and thus several guidelines need to be enforced to ensure semantic correctness of resulting NPCs.

First, all parameters must have appropriate ranges. Some parameters can support dramatic modifications, while others have a very small range. Some parameters can change from negative to positive (and vice versa), while others cannot. For example, if the speed of an average squirrel is 5 units/second, increasing that up to 50 units/second would clearly be an error, as would negating the speed since it would reverse the direction of all moves. Thus, each parameter to be modified should have a defined range. Speed for a squirrel could have the range $[4, 6]$.

Since the discussion is based upon creating semantically correct variations, thought should be given as to the selection of values within the range. Some properties are artificial and can have a wide range. An example of this is the wait time for a squirrel performing random exploration. High or low values are equally valid, and there is no ‘typical’ behaviour that players will expect. A random value within the defined range is acceptable for parameters of this class.

Other parameters tend to the mean. These tend to be physical properties of the natural being that is being represented. For example, adult squirrels would tend to be roughly the same speed, though some exceptional squirrels may move noticeably faster and injured squirrels may move much slower. To account for this, a probability distribution should be employed, such as a Gaussian. One approach would be to set the standard deviation from the mean to half of the distance between the mean and the range limits for that parameter. Anything that falls more than two standard deviations away could be rerolled.

Parameters can have interdependencies. Some are critical dependencies, where violations will cause erratic or incorrect behaviour. As an example, we refer back to Fig. 9. Perhaps the lowThreshold parameter has a range of $[3, 10]$, while highThreshold has a range of $[1, 5]$. A possible outcome is for the highThreshold to be greater than the lowThreshold. The result would be the removal of any behaviour associated exclusively with a low threat state, since the Statechart would transition immediately to the high threat state. If the lost behaviour is essential, then the resulting AI would be semantically incorrect. In cases like this, parameters should be generated with dynamic ranges. To solve the problem with threat ranges, a generator could use a range of $[1, 5]$ for highThreshold, then use a range of $[1 + highThreshold, 10]$ to generate the value for lowThreshold.

Some dependencies are non-critical. These relationships will not cause errors in the behaviour if ignored, but can reduce the realism of the resulting NPCs. Respecting these relationships will give better results. For example, a larger squirrel tends to move slower than its smaller counterpart. If there is a parameter that defines the size of the squirrel, a more realistic generation strategy will increase or decrease the generated movement speed of the squirrel.

These steps are summarized in our parameter modification strategy, presented in Fig. 16

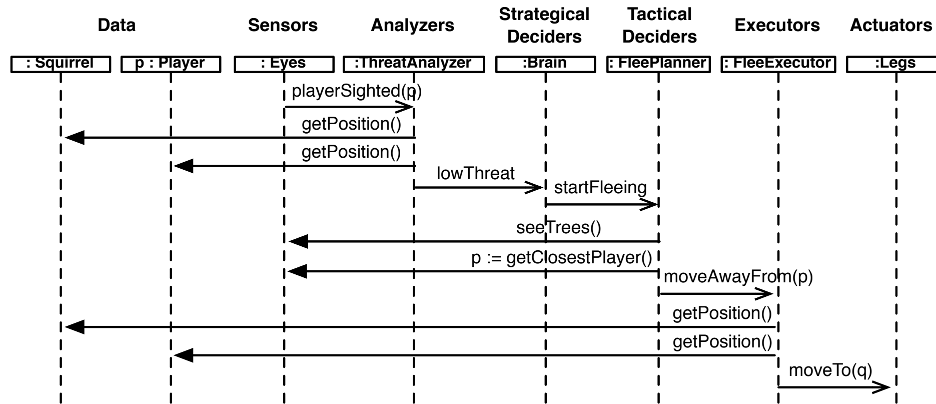

Fig. 15. Possible Event Sequence when a Player is Seen by a Squirrel

```
1. For each parameter:
   2. Create a range for each parameter.
   3. Decide on the probability distribution best suited
      for that parameter.
4. Examine parameters for dependencies.  For each
   dependency:
   5. Determine if the dependency is critical.
   6. If critical, assign dynamic ranges to resolve
      problem.
   7. If non-critical, resolve if desired, perhaps using
      dynamic ranges.
```

Fig. 16. Strategy to Perform Parameter Modification

B. Varying Component Configurations

Whereas the previous subsection presented how different behaviours of an *individual component* of an AI can be generated by changing parameter values, this subsection presents how different *overall behaviours* of an AI can be generated with little effort by assembling existing components in different configurations.

1) Removing Components: The easiest way to generate a new configuration from an existing one is to simply omit some of the components. In our modelling approach this is easy to do, since the communication among components is done using events. As a result, the components are only loosely coupled, since the component that generates an event does not know and therefore does not depend on which components react to the event, and vice versa. Theoretically, by selectively omitting components of an AI with n components, there are 2^n possible configurations. Practically, however, only a very small number of AIs generated by component omission exhibit realistic behaviour.

The strategic deciders are usually the most essential components. In general, most AIs will only have one "brain" that performs management at the highest level. Taking that out would sever the connection between the input and the output of the AI. In our squirrel example, removing the *Brain* component would result in a squirrel that performs no actions whatsoever.

Removing sensors or acutators is also dangerous. Since these components form the interface with the game at large, their funtionality cannot be replaced by other components. For instance, a squirrel without legs to actuate move commands cannot move, nor would any other class have the methods required to issue move commands to the game. The squirrel would be unable to perform core squirrel activities and thus would be invalid.

Interface components can be safely removed when there is an alternate. For instance, our squirrel model heavily depends on its *Eyes* to navigate in the world and find food. If our model had other sensors, such as a *Nose* that can also be used to find food, removing any one of them might generate interesting behaviour. Also, some components may be extra, such as *VocalChords*, and can be safely removed without invalidating the squirrel.

In general, the best component candidates to remove are analyzers and coordinators, followed by planners and executors. Without a given analyzer, the AI is not able to correlate sensor events, i.e., it is only capable of reacting to "raw" sensor data. Without a given coordinator, the AI's actions might become less efficient. As a result, the AI behaviour appears a little clumsy. Removing tactical deciders takes away the ability for an AI to perform one of its high-level goals. For our squirrel model, removing the *FleePlanner* would result in a squirrel that freezes when exposed to a threat. Removing an executor results in an AI that has difficulty in carrying out a tactical plan. However, the squirrel could still perform orthogonal tasks without issue.

This can be automated through the use of a flagging system. Given the full set of components, a subset should be designated as removable. The generation tool can then optionally remove any flagged components, knowing that the result will still be valid. Conditionals should again be used to validate the generated result. If a squirrel has *Eyes* and *Nose* components, both flagged as removable, then a realistic condition would be $Eyes \vee Nose$, ensuring that at least one of the two sensing components will always be included. Squirrels that violate this would not be valid and should be discarded immediately.

2) Replacing Components: A more advanced and effective way of generating new configurations from an existing AI is to replace components by other equivalent components. For example, it would make sense to replace our squirrel *FleePlanner* with a component that, instead of preferring to climb trees, would first try to run away from the threat, and only when trapped decide to climb a tree. To make things work, the replacement component must have a compatible component interface, meaning it should expect the same events

that the replaced component expected.

Our components actually have a loose coupling between the Statechart and the associated class. To replace a component, one could either associate an existing class with a new Statechart, or create a new class and associate it with an existing Statechart. The new elements must respect the existing interface, i.e., a new Statechart must use the same calls to the associated class that the old state diagram did, and similar for the reverse case.

Often there is a semantic equivalence between components, but not a syntactical one, i.e., the event that the first component produces is not the one that the reacting component expects. The meaning of the event, however, is the same. For example, the squirrel *Brain* generates a *startWander* event, but the *ExplorationPlanner* expects a *startExploration* event. To solve this problem, our approach allows the developer to specify that specific events of specific components are renamed, which makes it possible to use the *ExplorationPlanner* instead of the *WanderPlanner* within our squirrel model.

In order to automate the generation of new AI using component replacement, a library of semantically equivalent components needs to be created together with mappings that specify the correspondence between events.

3) Adding Components: Finally, it is possible to create new AI behaviours by adding components to an existing configuration. Components belonging to any category between sensors and strategic deciders are easy to add. New sensors augment the ways the AI can perceive the environment. For example, *ears* could allow a squirrel to detect approaching game entities even in the dark. New analyzers can help the AI to detect high-level events that are based on correlated occurrences of low-level events. New memorizers expand the capability of the AI to react based on historical information.

Components belonging to any category between the strategic deciders and the actuators are more tricky to add. Adding a new actuator component, for example, is easy, but it typically also requires the addition of a new executor that generates events that the actuator can react to. Following the same reasoning, a new planner is needed, and the strategic decider needs to be replaced with an updated model that activates the new planner when the situation is appropriate.

In the special case where the new actuator triggers behaviour that is completely orthogonal to any already existing behaviour, it is possible to add a second strategic decider that runs concurrently with the original one, controlling the newly added components independently. For example, a squirrel could have a second *Brain* that determines when the squirrel should wag its *Tail*, a new actuator. However, the general case requires also the addition of new coordinators to take care of interactions between the existing actuators and the one that is to be added.

The ultimate power of varying configurations is achieved when event renaming is combined with component addition. This makes it possible for a component to intercept events generated by another component and to transform them or delay them. The *StutterExecutor* component shown in Fig. 17 is an example of an executor component that, when asked to *stutterMove* to a given position, moves towards the destination

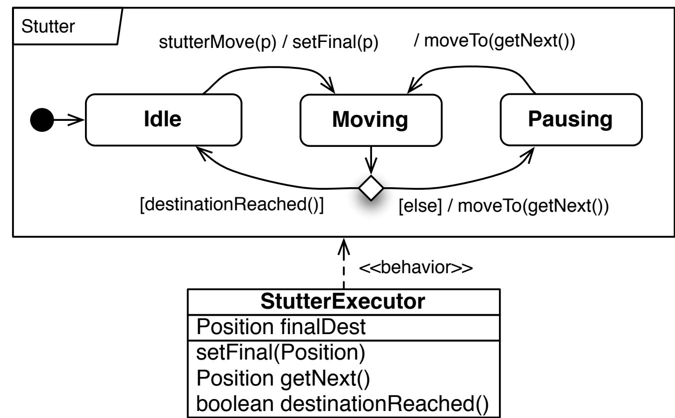

Fig. 17. Adding Stutter Behaviour to the Squirrel Movement

for a moment, then waits some time, then continues moving, then waits some time again, etc.

In normal situations, squirrels tend to move in this stuttering pattern. When under threat or when picking up food, however, squirrels run directly to their destination position without stopping on the way. In our current squirrel model there are three executors that produce *moveTo* events: the *WanderExecutor*, the *PickupExecutor* and the *MoveAwayFromExecutor*. Using our event renaming approach, it is possible to transform the *moveTo* events generated by the *WanderExectutor* to *stutterMove* events. As a result, our squirrel moves intermittently while exploring, but still runs straight for the target when picking up an acorn or when fleeing from a threat.

C. Varying Component Models

The most drastic variations consist of arbitrary structural modifications of the AI behaviour models. As with the previous variations, these modifications still need to preserve the semantics.

We have chosen to to explicitly model these variations in the form of transformation rules as these allow one to represent changes in the same modelling notation as the transformed models themselves. In rule-based model transformation [7] such as graph transformation, the transformation unit is a rule. A transformation rule uses model patterns as pre-conditions and post-conditions. The pre-condition pattern determines the applicability of a rule: it is usually described with a Left-Hand Side (LHS) and optional Negative Application Conditions (NACs). The LHS defines the pattern that must be found in the input model to apply the rule. The NAC defines a pattern that shall not be present, inhibiting the application of the rule. The Right-Hand Side (RHS) imposes the post-condition pattern to be found after the rule was applied. An advantage of using rule-based transformation is that it allows us to specify the transformation as a set of operational rewriting rules instead of using imperative programming languages. Model transformation can thus be specified at a higher level of abstraction (hiding the implementation of the matching algorithms), closer to the domain of the models it is applied on. When a model transformation is executed, rule scheduling describes in what order the rules will be applied (inter-rule

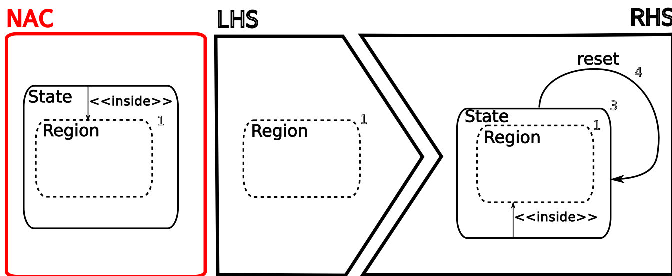

Fig. 18. The re-setting rule.

management). The priority-based rule scheduling we use tries every rule in decreasing order of priority. When a rule matches, it is applied and the process starts again from the highest-priority rule, until no more rules match.

A typical modification is encoded in the "re-setting a component" rule which wraps a Statechart in a single super-state with a transition looping from that state to itself. This transition is triggered by a "reset" event and effectively brings the Statechart back to its original state. This rule is shown in Figure 18. The LHS identifies a Statechart region. The transformation label "1" allows one to refer to that particular matched entity in other parts of the rule. The NAC specifies that this region should not be inside a Statechart state (orthogonal nor composite). This allows us to select only the top-level region of a Statechart. The RHS subsequently wraps the top-level region inside a new state (evidenced by the new label "3") with a new transition with trigger "reset" looping on it.

Another modification is the moving of a global state such as *Fleeing* in Figure 11 into a single orthogonal component. This has as an effect that activities in the other orthogonal components (*EatWhenReady* in our example) are not interrupted.

V. Mammoth Implementation

Mammoth provides the game framework for validating the results of our method. To enable Statecharts within Mammoth, we employed SCXML [3]. Generation of large numbers of AIs required the ability to quickly and easily import these into the Mammoth game world; the workflow issues are addressed here and a solution is presented.

A. NPCs in Mammoth

All non-player characters in Mammoth are controlled by an *NPC Manager*. On each iteration of the game loop, the NPC Manager executes the AIs of each NPC. This will sometimes result in a change to the game-state by causing the controlled NPC to take action. Actions could be moving, picking up objects, sending messages, or others.

AIs are individually housed in *roles*, which act as the high level containers for the various behaviours of each AI. Examples of roles in a medieval setting would be a shopkeeper, a city guard, a knight, or even a riding horse. In Mammoth, the squirrel AI described herein is implemented as a role. Specific behaviours are implemented as *tasks*; examples include moving, wandering, and eating. This arrangement is deliberately modular, in that any role can have any subset of the available tasks, with each role creating its own instances of tasks employed.

Adding Statecharts to Mammoth was done at the level of the task. Roles facilitate the operation of these Statecharts by acting as an event clearinghouse. All events, generated by the tasks, are passed directly to the containing role, where they are propagated in order of creation to all contained tasks. Tasks with Statecharts forward events to the Statechart's execution environment, where the event is processed and the Statechart reacts accordingly.

B. SCXML Description

The Mammoth implementation represented Statecharts using State Chart XML (SCXML). SCXML defines a representation of Statecharts in a human-readable XML format. Proposed as a W3C standard [3], the working draft gives the authoritative definition of the language.

States are defined using `<state>` tags. Attributes allow the specification of name and final status, amongst others. Transitions use the `<transition>` tag, and each transition can define the event and/or condition that triggers the transition, as well as the target. States can contain inner states, and those doing so must have an `<initial>` block that contains a conditionless transition to the default sub-state. Orthongonal components (called parallel states in SCXML) and history states are also supported.

The `action` portion of a transition, as well as the `onentry` and `onexit` blocks of a state, are represented as *executable content* in SCXML. Upon a transition, these locations are checked and the context executed. Some tags defining executable content include the `<log/>` tag, which logs to the SCXML log; the `<assign/>` tag, which can assign variable data; conditionals, such as `<if/>` and `<else/>` ; and others.

All executable content contains either the `expr` or `cond` attribute. Upon execution, contents of these attributes are passed to the implementation-specific expression-evaluator. Commons SCXML [8], developed through the Apache Commons project, is an open source Java implementation of SCXML. It provides Java libraries that create a complete SCXML execution environment, including the ability to parse SCXML files and execute the resulting state machine. Commons SCXML supports Commons Java Expression Language (JEXL) [9], also provided through Apache Commons. An executing state machine has a *context*, which is a simple hash table that can be populated with references external to the Statechart. Evaluation first resolves identifiers by looking them up in the context. Next, any method calls are evaluated through the use of reflection. Finally, the expression is evaluated and the appropriate action taken (based on the tag containing the expression).

C. SCXML Integration

Tasks in Mammoth were enhanced by adding an SCXML Engine. When an task is instantiated, it loads the indicated SCXML file and passes it to the SCXML Commons parser.

```
<?xml version="1.0" encoding="ASCII"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml"
             version="1.0" initialstate="safe">
 <parallel id="parallel">
  <state id="safe">
    <initial> <transition target="Wandering"/> </initial>
    <state id="Wandering">
      <onentry>
        <log expr="this.createEvent('start_wander',null)"/>
      </onentry>
      <transition event="foodLow" target="LookingForFood"/>
      <transition event="lowThreat" target="fleeing">
        <log expr="this.createEvent('fleeLow',_eventdata)"/>
      </transition>
      <onexit>
        <log expr="this.createEvent('stopWander', null)"/>
      </onexit>
    </state>
    <state id="LookingForFood"/>
      <transition event="foodNormal" target="Wandering"/>
    </state>
  </state>
  <state id="EatWhenReady">
    <transition event="readyToEat" target="EatWhenReady">
      <log expr="this.createEvent('eat')"/>
    </transition>
  </state>
  <transition event="high_threat" target="fleeing">
    <log expr="this.createEvent('fleeHigh',_eventdata)"/>
  </transition>
 </parallel/>
 <state id="fleeing">
  <transition cond="ThreatAnalyzer.isSafe()" target="safe">
    <log expr="this.createEvent('stop_flee',_eventdata)"/>
  </transition>
 </state>

</scxml>
```

Fig. 19. SCXML version of the Squirrel Brain Statechart presented in Fig. 11

The parser creates the execution environment in the form of an SCXML engine, along with a JEXL evaluator.

The execution environment and Mammoth are connected through the *context*, which is simply a hash map. Anything not added to the context is unaccessible from the Statechart, including the class to which the Statechart is associated. We mitigate this via a simple solution: before execution, the context is loaded with the key 'this', mapping to a reference to the associated class. Thus, expressions in the SCXML that refer to the associated class appear as `this.foo()` . Other fields in the associated object's class can be accessed by the Statechart either by adding them to the context, or by using get/set methods with a `this.getFoo()` call. Other convenient references may also be added. A memorizer may put a reference to its memory data structure, for instance. While it is possible to add anything to the context, we consider adding references outside of the associated object to be a degenerate approach and avoid this to maintain proper semantics.

Fig. 19 presents the Statechart from Fig. 11 in SCXML format. The structure is straightforward. Unfortunately, SCXML does not provide side-effect free executable content tags, so we use the `<log/>` tag as a workaround. Since we do not employ a logger, `<log/>` has no effect, but still triggers the JEXL Evaluator, allowing methods to be called without side-effects.

D. Loading NPCs

To facilitate the quick introduction of newly generated roles and behaviours into Mammoth, a run-time loading system

```
<?xml version="1.0" encoding="UTF-8"?>
<ai xmlns="http://mammoth.cs.mcgill.ca">
    <roleAssignments>

        <assignment npcName="Squirrel1">
            <role location="external"
                  roleName="squirrel.role.xml"/>
        </assignment>

        [...]

        <assignment npcName="Squirrel2">
            <role location="internal"
                  roleName="squirrel"/>
        </assignment>
    </roleAssignments>

</ai>
```

Fig. 20. External XML file showing an AI mapping.

for Mammoth was developed. An external XML file, corresponding to the players in the game world, defines a mapping between NPCs and the AIs that are to control them. The role for each NPC is instantiated as described above, then assigned to the player.

While this allowed roles to be quickly and easily changed, it was still limited to the roles that were hard-coded within Mammoth. This was insufficient for the purposes of AI generation, where a more dynamic system was required. Forcing a developer to directly modify the source code to accommodate each change is in clear opposition to the goal of easily generating new AIs.

The solution came in the form of external XML-based role definitions. To match the existing structure within Mammoth, a special type of role, called an *external role*, was created. External roles are classes in Mammoth that act as an role, but contain no tasks by default. Instead, they are empty vessels, ready to be told by an external source what tasks should be loaded, and what their parameters should be.

Figure 20 shows the XML file employed to map roles to NPCs. A role with an internal location, such for "Squirrel2" in this example, uses a built-in lookup table within Mammoth to instantiate the role with the given name. In this case, the role already contains the code to instantiate the contained tasks. A role with an external location, such as "Squirrel1," creates an External Role, and uses the given XML file to create tasks.

Task information is the entirety of the content in an XML role file. In Fig. 21, an external definition of a squirrel role is given. Each `<task>` block gives the information required to instantiate a task. First, the class attribute tells us what task class will be used. Next, the scxmlFile attribute points to the SCXML file that is to be associated with the class. A special constructor is used to instantiate an externally specified task. It accepts a set of parameters, populated from the XML role, and uses reflection to set the fields in the task. By accepting an arbitrary SCXML file, the user is free to give each role different behaviours for the same task in different roles.

VI. Validation

The techniques we describe ensure model validity, but some experimentation is still required to detect subtle component interactions, and correct any misconceptions as to parameter

```xml
<?xml version="1.0" encoding="UTF-8"?>
<role xmlns="http://mammoth.cs.mcgill.ca" name="squirrel">
  <tasks>

    <task class="Mammoth.AI.NPC.SCXML.SCXMLWanderPlanner"
                                                    type="scxml">
      <scxmlFile value="SCXMLWanderPlanner.scxml" />
      <xRadius value="2.5" />
      <yRadius value="2.5" />
      <restTimeMin value="2000" />
      <restTimeRange value="5000" />
    </task>

    <task class="Mammoth.AI.NPC.SCXML.SCXMLFleePlanner"
                                                  type="scxml">
      <scxmlFile value="SCXMLFleePlanner.scxml" />
    </task>

    <task class="Mammoth.AI.NPC.SCXML.
                  SCXMLProximityMemorizer" type="scxml">
      <scxmlFile value="SCXMLProximityMemorizer.scxml"/>
      <lowThreat value="1.0" />
      <highThreat value="0.5" />
    </task>

  [...]

</role>
```

Fig. 21. Squirrel.xml, defining a squirrel NPC for use in an external role.

dependencies. In our design this is facilitated by providing a direct path to game integration, allowing us to easily test different constraints and observe the impact on generated AI quality within the actual virtual context.

To quantitatively assess results we need a metric for determining whether squirrels meet minimal levels of "squirrel-like" behaviour. Certainly, all squirrel NPCs should sometimes gather acons and eat, and should move around occasionally. This gives us a fitness function for a squirrel AI:

$$(acorns_gathered > 0) \wedge (distance_moved > min) \wedge (energy_gained > min)$$

This type of fitness function allows for hard limits on certain behaviours. If we wanted squirrels that never ran out of energy and never approached a player, then we could easily add:

$$\neg(energy = 0) \wedge \neg(player_distance > min)$$

Experimentally, we developed squirrel AIs through the automatic creation of external squirrel roles, adjusting parameters to improve the resulting behaviour. Each test involved 50 new squirrel AIs. Squirrels were tested over a five minute period, gathering information needed by the fitness function. After the test, failing squirrels were examined and used to improve generated behaviour.

Modularity was valididated by adding and removing Statecharts. Each new AI was given a 50% chance of including a *StutterExecutor* as per Fig. 17. Brave squirrels were also generated by giving generated AIs a 25% chance to have no ThreatAnalyzer. Without this, the components interested in fleeing would never be activated, meaning such squirrels could fearlessly move past the player.

In our tests, maximum energy was set to a low value of 30 000 for all squirrels. This gives the squirrels exactly 30 seconds of movement, during which time they must find and eat an acorn, or else they run out of energy and become stuck. The short window forces the squirrels to act.

A. *Results*

Initially, generation was done using minimal constraints. However, this caused an infinite loop whenever AIs were generated with a higher threshold for critical energy than that of low energy. If such an NPC has an energy level between the critical and low threshold, the energy sensor class will incorrectly report that it has both high and critically low levels of energy. Transitions in the associated Statechart are guarded by conditions based upon the current energy level, and as a result the Statechart will cycle endlessly. This reinforces the need to properly identify and manage critical dependencies.

Using dynamic ranges for all dependencies (both critical and non-critical) allowed validation to proceed. The first test was highly unsuccessful, with about 50% failing. Examination showed that most failures were due to not eating a single acorn. To give the squirrels more chance to eat, the lower bound for the low energy threshold was increased. The result was that squirrels hit low energy sooner, meaning they spent less time on undirected wandering, and more time on trying to pick up food. This immediately increased the number of valid squirrels up to about 75%.

The failing squirrels tended to one of two problems. First, they gained very little energy despite eating. This tells us that squirrels were eating acorns while their energy was too high, and getting only a very small energy gain. The fix for this was to push the critical energy threshold lower. Secondly, a number of squirrels moved only a small distance and did not find any acorns. This implies that their wander radius used in searching was too small. The fix here was simply increasing the minimum radius of the wander range.

These next squirrels performed with 90% success. A visual observation showed us that squirrels fearful of the player (i.e., with a ThreatAnalyzer) could become caught in a cycle where they tried to pick up an acorn, encountered the player, fled, then tried to pick up the same acorn. Figure 22 shows a squirrel caught in this cycle. This is desired behaviour, in that the player could be thought of as 'protecting' the acorn. To capture this specialized interaction, the fitness function was adjusted to

$$(((acorns_gathered > 0) \wedge (energy_gained > min)) \vee (fled)) \wedge (distance_moved > min)$$

These settings repeatedly generated groups of squirrels that were 95% fit. The few remaining invalid squirrels all had a *StutterExecutor*, and simply did not expend enough energy to eat their gathered acorns. The wait time on wandering was thus reduced to increase activity. This final change led to several groups of 100% valid squirrels.

Using only these listed prototype iterations, we were able to quickly converge to a generation of only valid AIs. Figure 23 presents a screenshot that shows several of these valid squirrels, happily gathering under a tree and collecting acorns near the player.

In spite of the tighter constraints, there was still more than sufficient variation to consider the method worthwhile. Some squirrels moved almost constantly, industriously collecting dozens of acorns. Others remained indolent, only moving when necessary to eat and collecting only a few acorns. This

Fig. 22. A squirrel hungrily eyes the unreachable acorn near the player.

Fig. 23. Several squirrels gather near a concentration of acorns.

difference was noticeable when observing squirrels, and evident in the fitness numbers. Future work is required, however, to develop or apply further quantitative metrics for assessing player-observed variability [10].

VII. Literature Review

A. AI modelling

The use of visual modelling environments is not new to the gaming industry. Also known under the name of *Visual Scripting Languages*, finite state machines and other formalisms have been used to model various features of games [11], including cinematics and story narratives [12]. The main objective of developing such systems is to offload work from the programmers to the game designers and the animators, allowing them to participate in the development of the game without requiring any programming or scripting knowledge [13].

More interesting is the use of a modelling environment to define the behaviour of agents, as proposed by *Simbionic* and its toolset which allow a developer to describe the behaviour of intelligent agents using finite state machines [14]. We base our framework on statecharts, as they generalize FSMs with improved encapsulation, while still allowing for efficient implementations and visual development environments [15]. Our work directly builds on the Statechart AI-design described by Kienzle *et al.* [2].

Visual modelling environments can also be found in commercial engines. The Unreal Engine 3 [16] includes *UnrealKismet*, a visual scripting system, which provides artists and level designers the freedom to design stories and action sequences for non player characters within a game without the need for programming. One key feature of *UnrealKismet* is the support for hierarchy of components, which makes it possible to structure complicated behaviour descriptions nicely. The difference with our approach is that the models in *UnrealKismet* essentially describe the decision making steps of an AI algorithm graphically. Our approach does not model the control flow explicitly. The behaviour emerges based on the components that listen for and react to events.

Also worth mentioning is *ScriptEase* [17], a textual tool for scripting sequences of game events and reactions of non player characters. Although it does not use a visual formalism, *ScriptEase* introduces a pattern template system – a library of frequently used sequences of events – that allows designers to put together complex sequences with little programming.

Interest has recently grown in using *behaviour trees* for building complex game AI [18]. Like Statecharts, behaviour trees are inherently hierarchical, with encapsulated substates allowing for improved component reuse as well as more scalable implementations. Recent work on behaviour trees has shown that AI generation is possible using evolutionary techniques [19]. We believe the event-driven approach of Statecharts has more flexibility for purposes of rapid, automatic and varied generation, although it would be interesting to explore how to adapt our techniques to a behaviour tree context as well.

B. AI generation

Procedural content generation has been most successfully applied to visual game assets, and especially virtual terrains [20], [21], including urban [22] and residential architecture [23], interiors [24], and extending up to entire cities [25]. In structured game contexts, higher level content, such as game levels, can also be effectively generated [26].

Interesting variability in character behaviour is quite difficult to produce procedurally, and many games make use of simple scripts or rule-systems for minor characters [27]. For improved realism, group behaviour, such as found in crowd simulation [28] can be an efficient way of increasing player immersion. Individual AIs can also be modified at runtime, evolving strategy and better adapting to the current game context [18], [29], and increasing diversity [30]. For casually encountered characters such as extras, there is less opportunity for adaptation and so we have focused on the problem in terms of static generation. It would, of course be possible to apply our techniques at runtime to dynamically select or even construct behaviours.

VIII. Conclusions and Future Work

Constructing multiple, but varied AIs for minor characters is a challenging task. While game AI construction has traditionally been focused on increasing the depth of character

complexity, AI for extras can be seen as presenting problems in terms of breadth: simplified generation and game integration, as well as the ability to automatically create a large variety of behaviours is important. Our approach includes a graphical design context and workflow that allows us to easily incorporate AIs into our game framework, aiding initial high-level design and prototyping. Our quest to "minimize accidental complexity" led to the adoption of a Statecharts-based modelling formalism. This, as the essential features of the AI such as reactive, timed, event-based behaviour is most elegantly, concisely and re-usably expressed using the Statecharts formalism. Our modelling strategy allows us to introduce non-trivial variation into AIs, and we describe techniques based on multiple levels of abstraction. Importantly, these techniques also focus on ensuring semantic stability, limiting the extent to which behaviour can become unrealistic, and thus greatly mitigating the need for detailed runtime verification of individual AIs.

Significant future work is possible based on our framework and approach. We have focused here on extras and less important characters, but of course more important characters could also be modelled and generated using these same techniques. This would involve special considerations for their role in the game in terms of their interactions with the player.

We plan to extend our work on rule-based transformations. We explored only a few, simple rule designs; an extensive library of transformation schemas would further increase high level variability, and could be easily incorporated into our design. Furthermore, whereas our current exploration of AI variants is supervised by a modeller, a fully automatic exploration becomes feasible when transformation rules are applied in random order. As this may lead to AI models which exhibit non-realistic behaviour, automatic "performance analysis" based on simulated behaviour traces is needed to cull undesired variants.

ACKNOWLEDGMENT

The authors would like to thank the Natural Sciences and Engineering Research Council of Canada for its support.

REFERENCES

[1] J. Kienzle, C. Verbrugge, B. Kemme, A. Denault, and M. Hawker, "Mammoth: A Massively Multiplayer Game Research Framework," in *4th International Conference on the Foundations of Digital Games (ICFDG)*. New York, NY, USA: ACM, April 2009, pp. 308 – 315.

[2] J. Kienzle, A. Denault, and H. Vangheluwe, "Model-based design of computer-controlled game character behavior," in *Model Driven Engineering Languages and Systems*, ser. LNCS. Springer, 2007, no. 4735, pp. 650–665.

[3] J. Barnett, R. Akolkar, R. Auburn, M. Bodell, D. C. Burnett, J. Carter, S. McGlashan, T. Lager, M. Helbing, R. Hosn, T. Raman, K. Reifenrath, and N. Rosenthal, "State chart XML (SCXML): State machine notation for control abstraction," W3C, W3C Working Draft, May 2010.

[4] D. Harel and H. Kugler, "The Rhapsody semantics of Statecharts (or, on the executable core of the UML)," *LNCS*, vol. 3147, pp. 325 – 354, 2004.

[5] D. Harel and A. Naamad, "The STATEMATE semantics of Statecharts," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 4, pp. 293–333, October 1996.

[6] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[7] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, "Systematic Transformation Development," *Electronic Communications of the EASST*, vol. 21, 2010.

[8] Apache Commons, "Commons SCXML," http://commons.apache.org/scxml/, November 2010.

[9] ——, "Commons JEXL," http://commons.apache.org/jexl/, November 2010.

[10] R. McDonnell, M. Larkin, S. Dobbyn, S. Collins, and C. O'Sullivan, "Clone attack! perception of crowd variety," *ACM Trans. Graph.*, vol. 27, pp. 26:1–26:8, August 2008.

[11] S. Jacobs, "Visual design of state machines," in *Game Programming Gems 5*, K. Pallister, Ed. Charles River Media, 2005, pp. 169–176.

[12] C. J. Pickett, C. Verbrugge, and F. Martineau, "(P)NFG: A Language and Runtime System for Structured Computer Narratives," in *Game-On-NA 2005 - 1st International North American Conference on Intelligent Games and Simulation*. Eurosis, August 2005, pp. 23 – 32.

[13] S. Gill, "Visual Finite State Machine AI Systems," Gamasutra: http://www.gamasutra.com/features/20041118/gill-01.shtml, November 2004.

[14] D. Fu and R. T. Houlette, "Putting AI in entertainment: An AI authoring tool for simulation and games," *IEEE Intelligent Systems*, vol. 17, no. 4, pp. 81–84, 2002.

[15] P. Kolhoff, "Level up for finite state machines: An interpreter for statecharts," in *AI Game Programming Wisdom 4*, S. Rabin, Ed. Charles River Media, 2008, pp. 317–332.

[16] Unreal Technology, "The Unreal Engine 3," http://www.unrealtechnology.com/html/technology/ue30.shtml, 2007.

[17] C. Onuczko, M. Cutumisu, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, K. Waugh, M. Carbonaro, and J. Siegel, "A Pattern Catalog For Computer Role Playing Games," in *Game-On-NA 2005 - 1st International North American Conference on Intelligent Games and Simulation*. Eurosis, August 2005, pp. 33 – 38.

[18] M. Dyckhoff, "Evolving Halo's behaviour tree AI," Presentation at GDC, 2007, http://www.bungie.net/images/Inside/publications/presentations/publicationsdes/engineering/gdc07.pdf.

[19] C.-U. Lim, R. Baumgarten, and S. Colton, "Evolving behaviour trees for the commercial game DEFCON," in *Applications of Evolutionary Computation*, ser. LNCS. Springer, 2010, vol. 6024, pp. 100–110.

[20] R. M. Smelik, K. J. de Kraker, S. A. Groenewegen, T. Tutenel, and R. Bidarra, "A survey of procedural methods for terrain modelling," in *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, Jun. 2009, pp. 25–34.

[21] J. Doran and I. Parberry, "Controlled procedural terrain generation using software agents," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 2, pp. 111–119, Jun. 2010.

[22] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," in *ACM SIGGRAPH 2003 Papers*. New York, NY, USA: ACM Press, 2003, pp. 669–677.

[23] P. Merrell, E. Schkufza, and V. Koltun, "Computer-generated residential building layouts," in *The 3rd ACM SIGGRAPH Conference and Exhibition on Computer Graphics and Interactive Techniques in Asia*, December 2010, to appear.

[24] E. Hahn, P. Bose, and A. Whitehead, "Persistent realtime building interior generation," in *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, 2006, pp. 179–186.

[25] N. Rudzicz and C. Verbrugge, "An iterated subdivision algorithm for procedural road plan generation," in *4th Annual North American Conference on intelligent games and simulation (GameOn'NA 2008)*. Montréal, Canada: Eurosis, Aug. 2008, pp. 40–47.

[26] G. Smith, M. Treanor, J. Whitehead, and M. Mateas, "Rhythm-based level generation for 2D platformers," in *FDG '09: Proceedings of the 4th International Conference on Foundations of Digital Games*. New York, NY, USA: ACM, 2009, pp. 175–182.

[27] I. Millington, *Artificial Intelligence for Games*. Morgan Kaufmann, 2006.

[28] P. Kruszewski, "Real-time crowd simulation using AI.implant," in *AI Game Programming Wisdom 3*, S. Rabin, Ed. Charles River Media, 2006, pp. 233–248.

[29] C. J. Darken, "Individualized NPC attitudes with social networks," in *AI Game Programming Wisdom 4*, S. Rabin, Ed. Charles River Media, 2008, pp. 571–578.

[30] I. Szita, M. Ponsen, and P. Spronck, "Effective and diverse adaptive game AI," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 1, pp. 16–27, Mar. 2009.