Model-Driven Development of AI for Digital Games

Christopher W.A. Dragert

Doctor of Philosophy

School of Computing

McGill University

Montreal,Quebec

2015-02-16

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctorate of Philosophy

DEDICATION

This thesis is dedicated to my father, John Volker Dragert. He taught me how to think, how to reason, and how to be a man of science. Life interrupted his own Ph.D. studies, and he was never able to obtain the doctorate he so rightly deserved. Getting this degree is for both of us. We did it, Dad.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Jessica Ip. By helping to keep me sane when the days grew long, and gently pushing me to work when I needed it, your love and support meant the world. You are the love of my life, and I could not have done this without you.

Next, I would like to acknowledge the help and support of my family. Mom and Dad, Mike, John, and Jackie: all of you did so much to grant me confidence. Your unwavering belief in my ability was foundational to my own belief that I could in fact complete my Ph.D. Family is the most important thing in life, and mine is a great one.

For their mentorship and supervision, I would like to acknowledge Jörg Kienzle and Clark Verbrugge. My Master's supervisor Juergen Dingel once said that a Ph. D. is a unique period of your life where you have time and freedom to think deeply about a subject, and to truly do research. Jörg and Clark, you both gave me the utmost freedom to explore my own interests, follow my own path, and gave excellent supervision at every step of the journey. My studies here at McGill were fruitful because of your guidance, and you both get my profound thanks for all of the support you've given to me.

Strong friendships brought joy to my life during my degree. Tim, Ryan, Ian, and Greg were always there when I needed them, and offered nothing but true friendship. We, my friends, have a bond that shall endure.

Lastly, I would like to thank the Natural Sciences and Engineering Research Council of Canada, McGill University, and Lorne Trottier for the financial support. Funding is the life-blood of research, and strong support for young researchers enables them to do great work for the benefit of all Canadians.

Christopher William Arthur Dragert

McGill University

November, 2013

ABSTRACT

When developing AI for digital games, developers typically adopt a narrow, game-by-game focus, where new AIs are often developed from scratch for each game. Time is spent reimplementing behaviours, despite significant similarity between basic behavioural traits of non-player characters (NPCs). Software engineering offers many techniques that could streamline the game AI development process, but lack of a suitably generic AI representation that is both formalized and flexible inhibits their effective application.

In this thesis, we demonstrate how model-driven development techniques can be applied in the context of game AI development to effectively address these shortcomings. We propose a new formalism, *layered statechart-based AI*, to specify NPC behaviours. This representation allows designers to work at an appropriate level of abstraction while reducing accidental complexity and enabling model-driven development. Industrial-scale applicability of the approach is demonstrated by modelling a large-scale AI similar to that found in modern digital games.

Building from our clearly defined AI representation, we offer three specific improvements to the game AI development process that advance the state of the art while demonstrating the effectiveness of a model-driven approach. First, we present a comprehensive approach to modular reuse of game AI, made possible by the definition of an AI-module interface. Supporting this is *Scythe AI*, a tool to enable the modular reuse of statechart-based AI. Secondly, we show how correctness of a modular AI can be verified at both a syntactic and semantic

level through the use of model-checking. Finally, we illustrate how to generate a population of AIs with varying, individualized behaviours that preserve and enhance desired properties by means of automated parameter variation, AI module reconfiguration and model transformations.

ABRÉGÉ

Lors du développement d'intelligences artificielles (IA) pour jeux vidéo, les développeurs adoptent habituellement une approche "sur mesure" où les nouvelles IA sont développées à partir de rien. Un temps considérable est pris pour réimplémenter les comportements en dépit du fait qu'il existe de très grandes similarités entre les comportements de base des personnages non-joueurs (PNJ). L'ingéniérie logicielle permet de recourir à une multitude de techniques qui peuvent simplifier le dévelopement d'IA dans le contexte des jeux vidéo. Toutefois, le fait qu'il n'existe pas d'implémentation générique appropriée d'IA qui soit à la fois formalisée et flexible rend les techniques d'ingénierie logicielle beaucoup moins efficaces.

Dans cette thèse, nous démontrons comment les techniques de développement orientées-modèle peuvent être appliquées dans le contexte du développement d'IA pour jeux afin de palier aux lacunes précedemment exposées. Nous proposons un nouveau formalisme intitulé *layered statechart-based AI* (IA en couches basée sur des diagrammes d'état) afin de spécifier les comportements des PNJ. Cette modélisation permet aux concepteurs de travailler à un niveau d'abstraction approprié tout en réduisant la complexité accidentelle et en permettant de recourir à des techniques de développement orientées-modèle. L'applicabilité au niveau industriel de cette approche est démontrée par la modélisation d'une IA à large échelle comparable à ce que l'on peut retrouver dans les jeux vidéo modernes.

En construisant à partir de notre représentation d'IA énoncée précedemment, nous offrons trois améliorations spécifiques au processus de développement d'IA qui font avancer l'état de l'art tout en démontrant l'efficacité d'une approche orientée-modèle. Premièrement, nous présentons une approche globale permettant la réutilisation modulaire d'IA de jeux, rendue possible en définissant une interface commune aux modules d'IA. cette fin, nous proposons un outil permettant la réutilisation modulaire d'IA basées sur des diagrammes d'état: *Scythe AI*. Deuxièmement, nous démontrons dans quelle mesure l'exactitude (*correctness*) d'une IA modulaire peut être vérifiée aux niveaux syntaxique et sémantique en recourant à des techniques de validation de modèle (*model-checking*). Finalement, nous indiquons comment il est possible de générer une population d'IA possédant des comportements individualisés et variés qui préservent et bonifient les propriétés désirées en recourant à des techniques de variation automatisée de paramètres, de reconfiguration de modules d'IA et de transformation des modèles.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

The video game industry has exploded in size as game sales now easily eclipse Hollywood box office ticket sales. The recent hit game 'Grand Theft Auto 5' (GTA5) grossed over US$800-million in just its first day of sales [18]. Unfortunately, the cost of developing a major state of the art game (a 'AAA' title) has grown accordingly. In 2009, M2 Research put the cost of developing a modern multi-platform game between US$18 and $28-million [47], while the aforementioned GTA5 had a development cost upwards of US$130 million [6]. While these high totals include art and content creation, a sizeable portion is the cost of software development.

Over the years, many short cuts have emerged to reduce the magnitude of the programming task. Primarily, this is through the use of *game engines* that provide the core components of the game-to-be. A popular industrial choice is the Unreal Engine [70], while the Unity 3D game engine [69] is rapidly growing in popularity. Open source examples include jMonkey [64] for Java and OGRE [65] for C++. Usage of these engines allows developers to skip basic development tasks such as memory management and entity representation, and instead focus on customizing the provided environment for their game.

A primary enabler of the success of game engines is the format of game assets, chiefly graphics assets. Models for in-game objects are built from texture mapped

polygons using programs such as Maya or 3D Studio Max, then stored in one of several standard formats. Game engines can then implement a graphics engine that takes these assets and displays them using OpenGL or DirectX. This allows for reuse and portability of graphics assets, reuse of graphic engines, as well as practical approaches such as unit tests relating to the correct display of models in standard formats.

Reuse is thus a powerful tool in limiting costs. The game publisher Electronic Arts has built up a set of internal libraries that manage resource loading, controller input, sound management, and so on [13]. However, AI packages are conspicuously absent when considering reuse approaches. The lack of standardization means that the approach to AI for a game is often custom developed specifically for that game. Usually this results in developers designing simple AIs by reinventing prior art. For much of the history of the gaming industry, this resulted in non-player characters being driven by a *simple reflexive agent* using scripted behaviour customized to operate within the game context.

This type of narrow game-by-game focus is a source of consternation for game developers. At the Game Developer's Conference in 2011 (the largest conference of its type), Kevin Dill argued this point, saying that the lack of behavioural modularity was stymying the development of high quality AI [62]. With no standard, there is no generally agreed-upon representation of AI logic, and so there is no clear path towards the packaging and reuse of AI assets. While graphics development has a texture mapped triangle, there is no clear analogue for AI logic. If a developer could simply reuse basic behavioural modules (such as fleeing, taking

cover, following, and so on), then they would be free to spend development efforts on perfecting logic for intricate behaviours specific to their game context.

1.1 Game AI Development

We believe the fundamental cause for the absence of modularity and reuse in game AI is the lack of a formalism suitable for the application of software engineering techniques. A model-driven development approach would allow for modularity through component reuse. The predominant approach of scripted AI is too context specific for reuse, lacking a high-level model that would form the framework for higher-level reasoning. Games that do employ a formalism tend to use *finite state machines* (FSMs) or *behaviour trees*, both of which present problems for reuse that will be discussed in Chapter 10.

The lack of a high-level description of game AI inhibits more than just reuse. For instance, establishing the correctness of an AI is usually done by testing the AI in given scenarios. This can involve considerable and repetitive manual effort by hired testers and community beta testers. Each change to the underlying game logic may necessitate additional rounds of feature and regression testing. While software engineering offers many verification approaches such as model-checking, the value of heavy-weight approaches is limited by the lack of standardization.

An appealing alternative is offered by Kienzle et al. [35], who introduced an AI based on an abstract layering of statecharts. Here, each statechart acts as a modular component implementing a single behavioural concern, such as sensing the game state, memorizing data, deciding upon high-level goals, and so on. While appealing, little work has been done to show the benefits of this approach.

1.2 Contributions

The central aim of this work is to use model-driven development to improve the development process of game AI. Starting from prior work on layered statechart approaches, we develop and define *layered statechart-based AI*, a formalism to represent game AI. In this thesis, we will demonstrate both the utility and capability of the formalism, and show how this model-driven approach enables a number of beneficial techniques.

As a fundamentally modular approach, layered statechart-based AI is well-suited for modular reuse. We will define a complete approach to performing reuse. The approach will be supported in the form of *Scythe AI*, a tool designed to manage modular reuse of game AI. Since statecharts have a clear operational semantics, we are able to develop a technique to formally verify behaviour correctness of AI logic. This makes it possible for game developers to more formally test and verify their creations at design time. This additionally benefits reuse, since verification can ensure that new modules are correctly placed in the new context. Furthermore, we will explore transformations at the modelling level that will permit procedural generation of varying AIs.

Together, this work creates a impressive toolkit for the game AI developer, and introduces beneficial software engineering approaches to the field. Specifically, this thesis presents the following contributions:

- A complete and detailed description of layered statechart-based AI, expanding and building on the original theory. This includes a novel definition of

inter-statechart communication, and presents a model for communication that respects the original statechart definition.

- A complete description of a fully implemented AI for a squirrel using layered statechart-based AI. In addition to providing a simple test case, it additionally provides a complete guide to implementation, making it straightforward for others to develop their own AIs.
- A set of statechart-patterns that reflect common logic patterns for an AI. Similar to design patterns in software engineering, each statechart pattern represents a solution to a commonly encountered problem when designing AIs.
- The creation of the first large scale AI using layered statechart-based AI. This proves that the formalism can handle AIs at the scale of industrial games, while providing a valuable test-bed for future research.
- A complete approach to modular reuse of layered statechart-based AI. This includes definition of the reusable AI module, a thorough investigation of how modules interact, culminating in the creation of an AI module interface. Furthermore, we create a description of behavioural modularity through the composition of functional groups. A detailed example is presented, demonstrating the usefulness of the approach, and how reuse works in practice.
- Development of the tool 'Scythe AI', which provides a complete workflow for modular reuse. By supporting the creation and manipulation of AI module interfaces, managing involved files, outputting new AIs, and ensuring

correctness through an error and warning system, Scythe AI makes modular reuse practical for game developers and researchers.

- We present two approaches to verifying the correctness of a layered statechart-based AI. First, we address syntactic correctness, showing how a reachability analysis can be used to ensure that modules are connected properly. More powerfully, we address behavioural correctness by presenting an approach to model-checking behavioural specifications. Included is an exploration of various methods to create useful specifications that accurately represent game scenarios.
- We provide a complete demonstration of the verification approach by fully verifying the behavioural correctness of the squirrel AI. In addition, we investigate the verification tractability of our large scale AI.
- We develop a complete method to generate varied populations from a seed layered statechart-based AI. This subsumes existing techniques such as parameter modification, while formalizing compositional alterations using the new AI module interfaces. We also employ rule-based transformations to alter statecharts directly. Through a series of experiments, we show that alterations of this type can in fact be employed in a directed manner for generation of specific populations.

1.2.1 Validation

This work is validated largely through case studies. This is a fairly common approach in the modelling and software engineering communities. Using this approach, we present *prima facie* evidence that our techniques can indeed be

used to generate game AI. By giving a detailed explanation as we work through the case studies, the ability of our approach to overcome specific development challenges will be highlighted. The next subsection will detail the approaches used in validating this work, followed by a discussion of other validation approaches that were not employed.

In working through case studies, we will provide subjective evidence of our claims of improvements to the development process by describing our own experiences. Where our claims specifically require user validation, we will note where a user-study could be effectively employed. Again, this type of approach is common for evaluation of modelling problems. At Models 2013, the premier conference on modelling in software engineering, fully 50% of papers verified their work through a case study or example problem, while only 12% did so through user studies. Indeed, of those 12%, half of those papers focused entirely on the user-study itself.

Validation Specifics

To validate our work on the layered-statechart approach, we first seek to show feasibility. As a first step, we will work through the creation of a small example AI for a squirrel NPC. Such a proof of concept proves baseline feasibility, while demonstrating what an actual AI in the layered-statechart approach looks like and how it works. To prove scalability, we then reimplement a complex, full scale AI deriving from the Halo series of video games. Success in modelling demonstrates that our approach can scale up to the size of larger AIs found in industry, while providing some insight as to the size and difficulty of creating complex AIs.

Importantly, we attempt to maintain the core benefits of the original Halo AI, showing that our approach can also provide the same key features.

The work on reuse will be validated through a case study, and the creation of the tool Scythe AI. Here, we aim to demonstrate that the developed modular approach is well-designed with respect to enabling a reuse process to generate a new NPC AI from existing modules, and enabling tool support. A reuse case study clearly illustrates the process and gives a good impression of how and when reuse can be applied. A primary concern when formulating the layered-statechart approach was to provide enough modularity to allow for tool support; creation of a tool shows that this goal was met. While the effectiveness of Scythe AI itself could only truly be established through a user study, we show the design intention behind the tool and describe how features of the software are designed with the reuse process in mind.

The section of the thesis covering verification of layered-statechart based AI allows for a more direct validation. Here, we apply the verification techniques enabled by our modelling approach to exhaustively verify the correctness of the squirrel AI with respect to the design goals. In addition, the depth of this verification yields firm descriptions of the time and expense involved in verifying layered statechart-based AIs. We extend this analysis by applying the strategy to the Halo AI case study, addressing the question of scalability. Similar to Scythe AI, the exact benefit of this with respect to the development process would require a case study.

Finally, the work on generating varied AI will be validated by experimentation. The goal of this work is to inject changes that cause generated NPCs to express new behaviours, or to show significant variability in their extant behaviours. By generating populations of NPCs and testing them, we can directly evaluate if we have met our goals of increasing variation and variability.

Development Process versus Process Output

While we focus on the development process itself, the application of the process results in NPCs that meet game design goals, and provide or enhance the enjoyableness with respect to the player. To be clear, our primary concern is not the generation of a 'good' NPC, but rather to create processes and a tool that, in the hands of a talented game designer, could be employed to more quickly and easily create NPCs (when compared to the current state of the art). Testing the quality or enjoyability of our own generated NPCs speaks more to our talents as game designers, but has little bearing on the effectiveness of the presented approach.

User Studies

The ideal way in which to validate our claims of improvements to the development process would be through user-studies. In this case, the act of evaluating improvements would require a large amount of domain specific knowledge. Evaluating non-experts would conflate the question of *how* to make an AI with the *process* of constructing an AI. Avoiding this necessitates expert users, most likely professional AI programmers working in the video game industry. Unfortunately,

recruiting a pool of volunteers meeting this stringent requirement proved impossible. As well, effectively comparing their current workflow including the use of professional tools against our new workflow is complicated due their high level of expertise with their existing tools.

1.3 Thesis Layout

This document tells the full story of layered statechart-based AI, addressing the basic theory, and proceeding to cover the advanced topics of reuse, verification, variation, and tool support. It begins in Chapter 2 by giving appropriate background on the field of game development and AI in particular. This includes descriptions of the popular formalisms used to represent AI and their supporting background theory.

Chapters 3, 4, and 5 together give a detailed description of layered statechart-based AI. Chapter 3 gives full background on the original description of the approach, while updating and improving aspects of the theory. Chapter 4 shows the approach in practice, giving a complete description of an AI for an NPC squirrel, including an implementation in an actual large-scale game. To show that this approach meets the needs of industry, Chapter 5 presents a implementation of a complex AI similar in complexity to AIs in AAA games. This is a significant study, and demonstrates the validity of the layered statechart-based formalism. Along the way, it reveals several important features of developing a layered statechart-based AI in practice.

Chapter 6 examines in detail what it means to reuse a module of statechart-based AI, and how reuse can be performed. This includes definition of a module

interface and functional groups to make reuse a practical and straightforward process. This pairs with Chapter 7, which introduces the Scythe AI tool. At the time of writing, Scythe AI was developed to the point of enabling reuse, and so this chapter discusses the nature of the tool and its functionality.

Chapter 8 addresses the important topic of correctness and validation, detailing how layered statechart-based AI can be verified at the model level. To determine the value of the approach, we apply the process to the squirrel NPC by performing a complete verification with respect to design goals. The effectiveness of our validation is further demonstrated by applying it to the complex AI developed in Chapter 5. Chapter 9 addresses generation of variations. This topic shows the power of manipulations at the modelling level, further justifying the model-driven approach. The work in this chapter is directly validated through experimentation.

Chapter 10 presents a survey of related academic work. This chapter gives a broad summary of how these results fit into the academic field at large, and in some places, serves to illustrate that much of this work is exploring novel ground. Finally, Chapter 11, offers concluding thoughts and explores avenues for future research on the topic.

CHAPTER 2
Background Theory

Understanding how to apply a software engineering approach to computer gaming requires a wide range of background theory to properly describe and illustrate the process. This chapter starts with a thorough description of the application domain. Computer games are described along with the role of AI in gaming, followed by descriptions of the AI formalisms appearing in this document. Much of the work will be based on the layered statechart formalism and so special attention is given to this topic. The remainder of the chapter will introduce some of the software engineering approaches employed, including modular design, reuse, and model checking. Other necessary background will appear as needed throughout the document.

2.1 Artificial Intelligence in Computer Games

Computer games, colloquially known as video games or digital games, can be described as interactive computer programs with the primary goal of entertaining the user. The user, now called a *player*, has control over some aspect of the game, influencing or controlling progression through the game. Challenges are presented to the player that must be overcome through a combination of strategy, reflexes, coordination, or timing. If the player has a specific character within the game that they control, that character is either called an *avatar* or a *player character* (PC). Many games include other entities within the game world that act and

move without player input, such as enemies, allies, and others. These are called *non-player characters* (NPCs).

A *game context* provides the description of the game world, detailing the game world, the rules, and how the game progresses. The game context for the classic game Chess would be comprised of information about the board layout, starting position and abilities of the pieces, rules regarding movement and taking pieces, along with turn taking. Some variations would include extra information such as the length of turns. Once a game context is established, the current state of the game is tracked in what is called a *game state*. Looking at Chess once again, the game state would be comprised of the current position of all pieces on the board, along with whose turn it is. Thus, any computer game can be fully described using a combination of game context and game state.

Artificial intelligence plays an important role in modern computer games. Most frequently, it is used to control NPCs such that they exhibit behaviours relevant to the character's role within the game context. This type of AI is referred to as *computational behaviour*. In cases where the governing AI is inspired by natural intelligence, the term *computational intelligence* is sometimes used. When considering game development, efficiency, rapid development, and testability are paramount, strongly constraining design approaches. This distinguishes game AI from more complex classical AI approaches such as inference engines, neural networks, and learning.

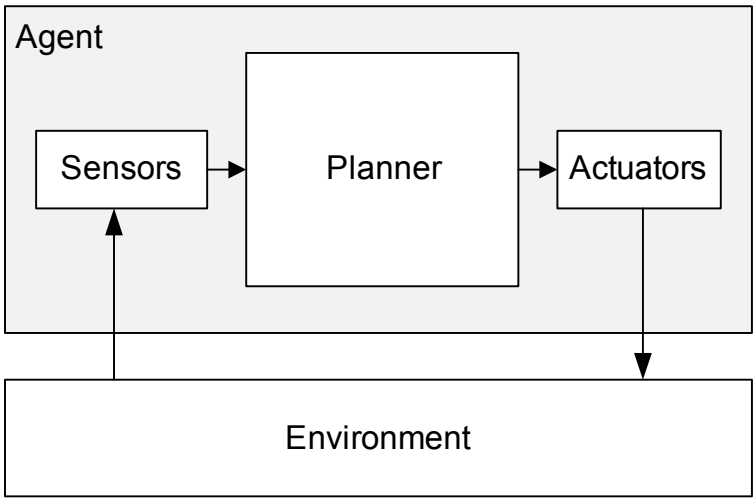


Figure 2–1: The Sense-Plan-Act architecture.

2.1.1 Agent-Based AI

Theory governing the behaviour of autonomous agents is well established in robotics literature such as in Arkin's book [3], which summarizes behaviour-based control of robots. In the *sense-plan-act* architecture, a robot senses the current state of the world, uses that information to decide upon a goal, then converts that goal into a command that drives the actuators. This is shown in Fig. 2–1.

One thing common to all of the aforementioned AI architectures is that they make behavioural decisions as independent entities. This ability to self-determine goals is the central concept of *agent-based* control, where each AI is an independent entity that controls only a single entity. A robot operating in the real world is analogous to an AI operating in a game world, since each makes observations about its environment, chooses a goal, and executes it using the actions and actuators available to it. Indeed, most NPCs in games use agent-based

AI. Simple *reactive* agents are common, where NPCs react identically to identical game states. These are like sense-plan-act architectures that omit the planning portion. *Deliberative* agents employ memory, allowing them to select behaviours using past observations and decisions. *Hybrid* agents combine the two agent types by using reacting deterministically to some events, but using a deliberative approach for other inputs. For many types of games, enemies only require highly simplistic tactics such as running at the player and attacking, and thus a reactive agent is sufficient. Modern first-person shooters use military tactics such as squad-based maneuvering, searching behaviours, and taking cover, necessitating the use of deliberative or hybrid agents.

In both games and robotics, sense-plan-act architectures grow increasingly complex as the desired behaviour becomes more sophisticated. The decision making process becomes more challenging as it must continually choose between an increasing variety of less differentiated behaviours. An alternative approach is the *subsumption* architecture described by Brooks [5], shown in Fig. 2–2. Here, complexity is controlled by separating behaviours into different layers. Lower layers handle simple tasks, such as collision avoidance, while higher levels handle more complex behaviours, such as moving towards a goal, exploring, and so on. Each layer receives sensor data and sends data to actuators, meaning that lower levels can act independently of higher levels. Consider a moving robot that detects an imminent collision with an object. In a standard sense-plan-act architecture, this information would be fed into a complex general decision process along with all other sensory data, whereas a subsumption approach would feed sensor

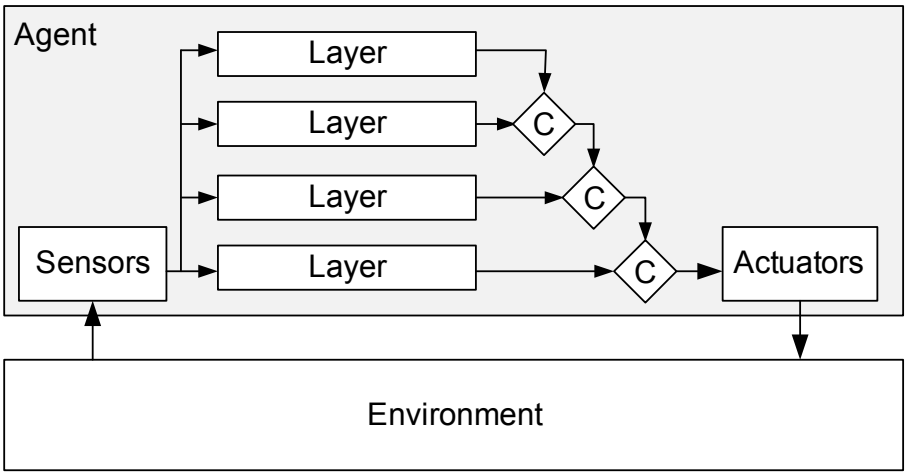


Figure 2–2: The subsumption architecture. C's represent coordinators.

data into a layer concerned only with collision avoidance, which in turn would immediately issue a command to actuators that prevents the collisions. This allows all notions of collision avoidance to be removed from higher levels, reducing the complexity of the decision making process. However, the subsumption approach is not without its drawbacks. Since multiple layers could issue contradictory commands (e.g., collision avoidance says turn left while an exploration layer says turn right), coordinators process all actuator commands to resolve any conflicts.

While the subsumption architecture was popular, the interdependence between layers made it difficult to modify any layer without being forced to modify all layers. By surveying a number of successful architectures, Gat *et al.* [21] classified robotics algorithms as falling into three layers. The exact layers represent strong commonality between independent approaches. At the control layer, sensors are closely looped to actuators such that the robot can perform primitive behaviours such as wall-following. The next layer, the sequencer, seeks to

stitch together primitive behaviours to accomplish larger tasks. The final layer, the deliberator, performs long term calculations such as planning. This description has proved highly popular and many subsequent efforts have adopted this classification.

While AI for robots and AI for game NPCs have large similarities, there are some specific differences that are worth highlighting. Indeed, some of the primary challenges in robotics (such as sensing, world modelling, and planning) are simplified in game AI. This is because a game is essentially a simulation that both defines and controls the complete model of the game. With respect to sensing, the game logic itself can cause an AI to unambiguously sense whatever it needs to sense at any given time[1] . World modelling is unnecessary, since the game is already represented in a format that is directly accessible. Planning is simplified, since the controlled nature of a game simulation makes the expected outcome of a plan much easier to predict, if not completely knowable in advance. Still, an NPC must know what to do with the model and the sensed data, and so it must have higher level abstractions that allow it to decide how to use this information and act within the game world. In this regard, the modelling and layering techniques developed in robotics provide excellent reference points when considering formalisms and layering problems in game development.

[1] Indeed, the difficulty in games more often lies in making NPCs worse at sensing, rather than better. This is because players dislike NPCs that appear omniscient, especially in genres such as strategy gaming.

2.1.2 Game AI Formalisms

While game AI has the ultimate goal of choosing behaviours for an NPC, the decision making structure employed by an agent can vary quite broadly. One common approach is to employ arbitrary code expressed through a custom scripting context [70, 53]. A scripted NPC is usually a reactive agent, reacting to inputs by acting in accordance to the script. Relatively simple tree and graphs structures are also used, such as decision trees. Millington's introductory text summarizes a variety of these approaches [52].

Goal-based planning approaches, including hierarchical task networks, are occasionally employed, such as that in the commercial game F.E.A.R. [54], or the academic language ABL [48]. In this approach, a library of actions with preconditions is created, defining the capabilities of the AI. Behavioural goals establish preconditions for a task, then a search finds and executes an action that satisfies preconditions and accomplishes the goal. Dynamic awareness and reactivity are compromised by this model. In practice, the planner must be repeatedly run to ensure that the selected action is still the appropriate one. Reuse models are based upon exporting actions and goal sets.

Behaviour Trees

Behaviour trees are quickly becoming a popular formalism in industrial game AI. Highly successful industrial games such as *Halo 2* [32] and *Spore* [28] have brought this approach to the forefront. Behaviour trees (BTs) are strictly hierarchical decision trees, where leaf nodes represent actions and non-leaf nodes choose how the tree should be navigated. Each node has an evaluation function

that when called by the parent, returns true or false. Action nodes answer this based upon the result of the action undertaken. If the action is to open a door, the node might return true if the door was opened, and false if the door remains closed.

Non-leaf nodes evaluate in a much different manner—by asking their children to evaluate. This child-evaluation is what gives BTs their hierarchical nature, and allows a designer to place behaviours in a logical manner. The two primary non-leaf nodes are *sequencers*, drawn as circles containing horizontal arrows, and *selectors*, drawn as circles containing question marks. A sequencer begins by asking its first child to evaluate. If it returns true, the sequencer asks its next child to evaluate, moving through all of its children in a sequence. If they all evaluate successfully, then the sequencer returns true. However, if any of the children return false, the sequence immediately halts and the sequencer returns false. Selectors seek to successfully execute a single behaviour. A selector asks its first child to evaluate, and if it returns true, then the selector halts and returns true. If the first child returns false, the selector evaluates the second child, repeating this until a child eventually returns true. If all children evaluate to false, then the selector returns false. Since both of these nodes evaluate from the first child, the traversal of the overall tree is depth first.

With only selectors and sequencers, it is possible to construct a working behaviour tree. In Fig. 2–3, a behaviour tree is given that guides an NPC through a door. Node evaluation takes us all the way down to selector 4. If the door is unlocked, or if the NPC can unlock the door, selector 4 will return true. In that

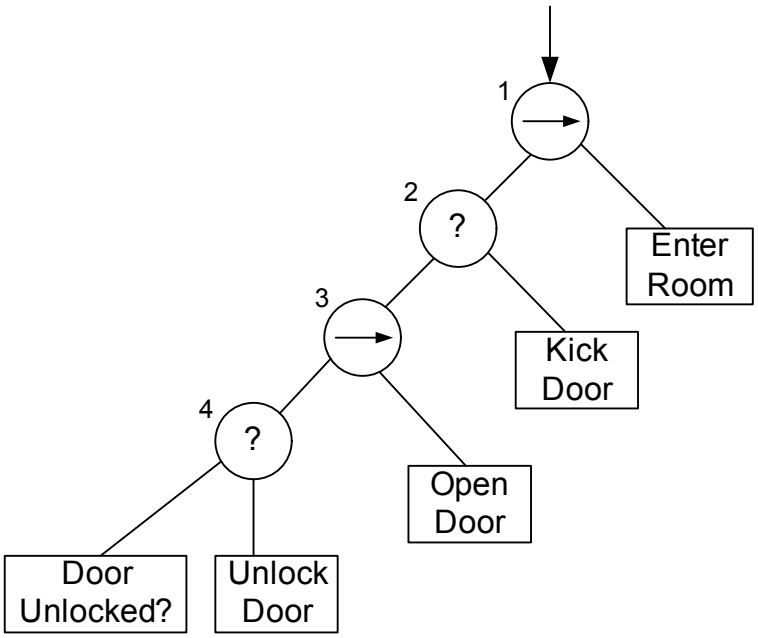


Figure 2–3: An example Behaviour Tree.

case, sequencer 3 will proceed to open the door and return true. However, if the door is locked and cannot be unlocked, sequencer 3 will halt and return false. If the door has been opened, selector 2 will return true; if the door is still closed, the AI will kick it in. Now that the door has been dealt with, sequencer 1 will complete the last action of entering the room.

The simple structure afforded by these two nodes can be unsatisfying, and so additional nodes are sometimes used. These can include repeaters, that repeat behaviours, random selectors, decorators that attempt different actions based upon a property of the NPC, and so on. However, all BTs suffer from a common issue. Once evaluation has gone down a certain path, there is no practical manner to react to changing game states that would alter a previous decision. Thus, typical

execution of a BT starts from the root on each pass, and actions must carefully consider a variety of variables, often unrelated to the specific behaviour, before returning true or false.

Finite-State Machines

The most prevalent formalism in game AI is that of *finite-state machines* (FSMs). An FSM typically consists of a finite state automata, where states represent the current status of the NPC. The transitions between states are directed, and labelled with inputs from the game. An example is shown in Fig. 2–4, where the small arrow pointing to the *Waiting* state indicates that as the initial state. When the game generates the event `ev_enemySpotted`, the FSM transitions into the state *Fighting*. To issue commands to the NPC being controlled, most FSMs include the notion of *actions*. An action can occur as part of a transition or when a state is entered or exited, and usually results in a function call that effects a behaviour. In the example, entering the *Fighting* state would cause the coded fight behaviour to begin executing, with the result that the NPC would fight the player.

While FSMs are reasonably intuitive for designers, increases in AI complexity can result in unmanageably complex state machines. Still from the previous example, a fighting behaviour might break down into states for moving, aiming, firing, reloading, and taking cover. These in turn may break down into further groups of states. Connecting these states back to *Waiting* and *Fleeing* requires a dramatic increase in the number of transitions, and quickly makes the FSM too large and complex to work with.

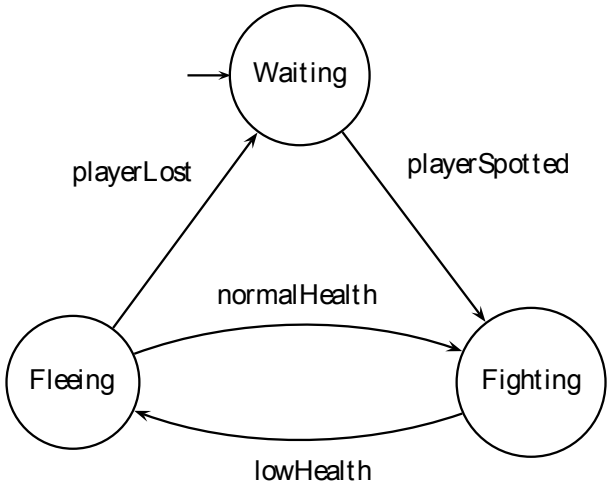


Figure 2–4: An example Finite State Machine.

In Harel's work introducing statecharts [24], the complexity problem was addressed through the introduction of hierarchical states. This notion was extracted and added to simple FSMs, creating *Hierarchical Finite State Machines* (HFSMs). These allow states to contain substates, with internal transition structures between substates. A state containing other states is called an *outer* state, while a contained state is called an *inner* states. For example, specific fighting behaviours such as aiming and taking cover could be more easily represented using substates within the `Fighting` state. Transitions may originate in inner or outer states, meaning the transition structure from Fig. 2–4 would remain the same, with the addition of internal states and transitions within the `Fighting` state. It is possible for a state to have an inner and outer transition with the same label, introducing non-determinism that must be resolved in any implementation.

Definitions of FSMs and HFSMs are deliberately left informal. In the game AI field, the labels are used as a blanket term for any state machine-based approach.

For example, an FSM approach could include guarded transitions, random or dynamic target states for transitions, or other variations, yet would still be called an FSM or an HFSM if it allowed substates. The vagueness on this point is reflective of the reality of the community.

2.1.3 Statecharts for Game AI

Statecharts were introduced by David Harel in 1987 [24] as a formalism for visual modelling of the behaviour of reactive systems. These generalize FSM and HFSM (which explicitly derive from Statecharts) approaches. A full definition of the Statemate semantics of statecharts was published in 1996 [26]. Statecharts are built from states, representing states of the modelled system, and connected with transitions between them. Events are discrete, meaning that the transition between states is considered to be instantaneous. Later, Harel published the Rhapsody semantics [25], which clarifies the object-oriented behaviour of statecharts and describes the executable semantics. All statecharts in this document will use the Rhapsody semantics.

Transitions are of the form $e[g]/a$, where e is the triggering event, g is a guard condition, and a is an action executed when the transition is taken. Each of these three parts is optional. Additionally, states can have actions that occur on entry or on exit. When a transition triggers multiple actions, they are executed in the order [on exit→transition→on entry]. Figure 2–5 gives a simple statechart with two states: *Start* and *End*. The small arrow pointing to *Start* denotes that state as the initial state. If the system is in state *Start* and it receives `event` while condition

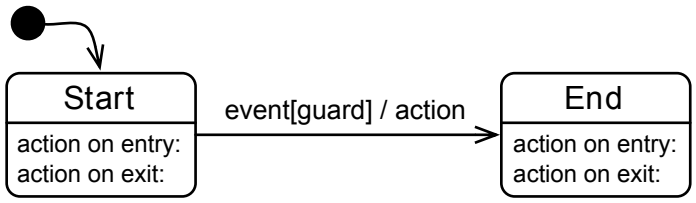


Figure 2–5: Statecharts Basic Transition

`guard` evaluates to true, the transition to state *End* is taken and the side-effect `action` is executed.

Statecharts allow states to be nested, as shown in Fig. 2–6 where the composite state *S1* has several nested states. Initially, the system is 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 `e`, an outgoing transition is looked for which is triggered by event `e`. This lookup is performed traversing all nested states, from the inside outwards as described in the Rhapsody semantics [25], in this case going to *S12*. This keeps the semantics deterministic despite the appearance of conflict between the inner and outer states. When in state *S12*, there is no conflict and event `e` will take the system to state *S2*. When in state *S2*, event `f` will take the system to state *S1*. As the latter is a composite state, the system will transition to *S11*, the default state of `S1`.

After leaving a nested state, *history* states denoted as a circled H allow for a return to the previous substate. When the outer state containing the history state is left, the current inner state is stored. A transition going to a history state instead terminates at the stored previous state. If there is no stored state,

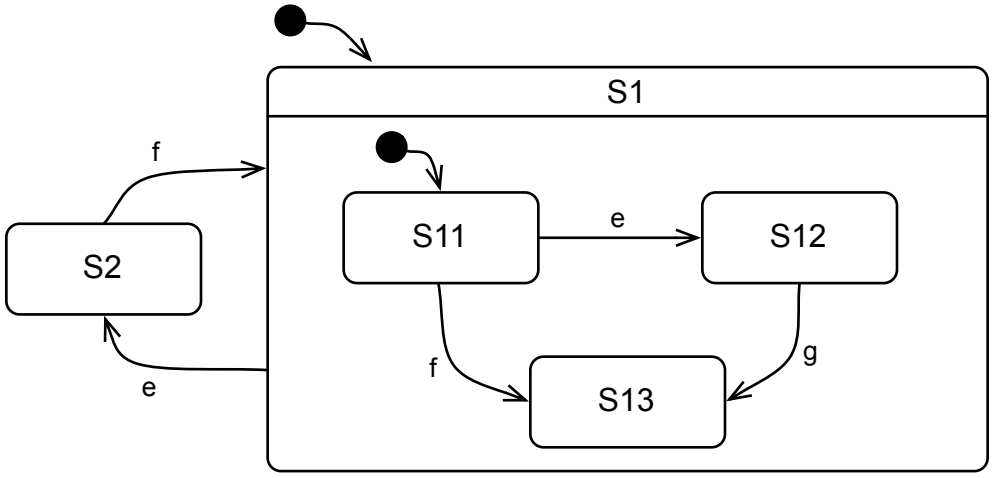


Figure 2–6: Hierarchy in Statecharts

occurring when the nested states have never been visited, the default state will instead be used.

In addition to hierarchy, statecharts add orthogonal components and broadcast communication. Fig. 2–7 shows a statechart with orthogonal components *ocA* and *ocB* as separated by a dashed line. This denotes that the system will simultaneously be in exactly one state in each of the orthogonal components. As such, this is a short-hand notation for the unordered Cartesian product of state sets. All orthogonal 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 semantics of event passing between orthogonal components will discussed in further detail in §sec:eventBasedCommunication.

Typically, FSMs used in games lack history states, inside-outwards transition resolution found in the Rhapsody semantics of statecharts [25], and an associated

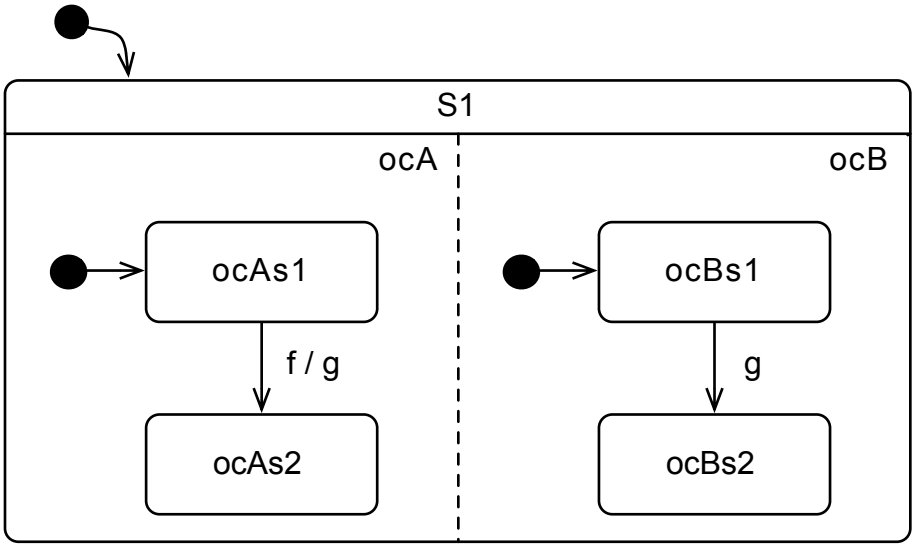


Figure 2–7: Orthogonal Components in Statecharts

class as in the *Unified Modelling Language* (UML). The lack of orthogonal states means that FSMs typically describe the entire behaviour of an NPC in a single state machine. Maintenance becomes increasingly complex as the addition of each new behaviour require a composition of existing behaviours. However, the single state nature becomes advantageous during testing; problems with an AI are isolated to the code for the current state.

Statechart-based AI

An appealing formalism was developed by Kienzle et al. [35], who introduced an AI based on an abstract layering of statecharts. Here, each statechart is a module implementing a single behavioural concern, such as sensing the game state, memorizing data, making high-level decisions, and so on. Due to the clear demarcation of duties, the modules are ideal for reuse. Each layer can contain any number of statecharts, or none at all.

Figure 2–8 gives an overview of the formalism. At the lowest layer lie *sensors*. These read the game state, typically through listeners or observers that generate events when a change is detected. Events are passed up to *analyzers* that interpret and combine sensing data to form a coherent picture of the game state. The next layer contains *memorizers*, which store analyzed data and complex state information for later reference. The highest layer is the *strategic decider*[2] , which interprets analyzed and memorized data to decide upon a high level goal. The high level goal is passed down to the *tactical deciders* to determine how it will be executed. Becoming less abstract, the next layer provides *executors* that enact execution decisions, translating goals into actions. Depending on the current state of the NPC, certain commands can cause conflicts or sub-optimal courses of action. Conflicts of this type are resolved by *coordinators*. The final layer contains *actuators*, which execute actions by modifying the game state.

This research focuses considerable effort on using this approach, and has expanded significantly on the formalism. The following chapter will go through Layered statechart-based AI in detail, noting new additions and modifications as appropriate.

2.2 Software Engineering

Model-Driven Engineering (MDE) [61] is a unified conceptual framework in which software development is seen as a process of model production, refinement

[2] Typically, there is only one strategic decider, but an AI that needs to perform orthogonal tasks could have a strategic decider for each of them.

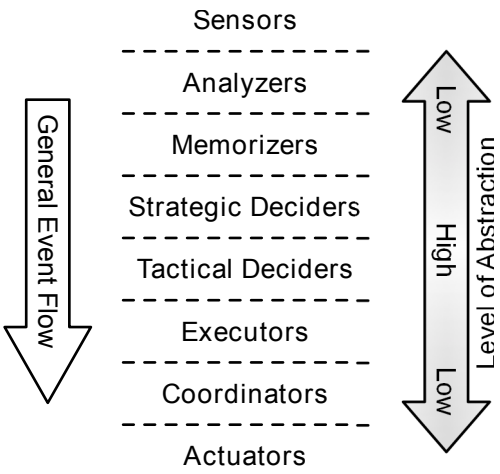


Figure 2–8: Layers in the Layered Statechart formalism

and integration. Models are built representing different views of a software system using different formalisms, i.e., modelling languages. The formalism is chosen in such a way that the model concisely expresses the properties of the system that are important at the current level of abstraction. During development, high-level specification models are refined or combined with other models to include more solution details, such as the chosen architecture, data structures, algorithms, and finally even platform and execution environment- specific properties. The manipulation of models is achieved by means of model transformations. Model refinement and integration continues until a model (or code) is produced that can be executed.

A primary goal in modelling is to limit *accidental complexity*, which is complexity introduced by the modelling formalism rather than the fundamentals of the problem. In games, it is common practice for an AI to sense the game state through events or by testing conditions. Conceptually, behavioural specifications

are highly related to these events as in the example "when the player is spotted, I will attack him". Given that statecharts are an event-driven formalism, it is straightforward to model specifications of this nature, thus limiting accidental complexity and making statecharts a highly appropriate formalism for modelling game AI.

2.2.1 Modularity and Reuse

Rather than address an entire programming task as a single challenge, complexity can be reduced by decomposing a problem into sub-problems. This is known as a modular approach [55]. Logically separate portions of functionality can be developed independently, then integrated. This requires well-defined interfaces, so that modules can correctly interact with one another.

Modularity leads naturally into encapsulation [56]. When a module interacts only through a well-defined interface, the internal structure of that module is not relevant. This is the notion underlying information hiding, whereby design decisions that are likely to be modified can be safely hidden behind the interface.

When a module implements a requirement and has a well-defined interface, it becomes possible to perform *modular reuse* [39]. Any new artifact that requires already developed functionality can simply reuse the existing module. Since the module communicates only through a well-defined interface, reuse is accomplished by importing the module and satisfying its interface. This approach has significant potential in the area of computer game AI, as many AIs reimplement existing behaviours and are thus prime candidates for modularization and reuse.

2.2.2 Verification and Model Checking

Verification of software is the act of examining or reasoning about possible program states to ensure that requirements are being met. In large, complex applications, especially in concurrent programs, the number of possible paths or interleavings can grow at an exponential rate. In a problem called *state-space explosion*, the number of possible system states grows to a such a size that manual testing methods are unmanageable, and overly difficult for a human to reason about. *Model checking* is an automated approach that performs an exhaustive search of the state space, examining every possible interleaving. Common tests include deadlock-freedom, assertion violations, and verification of temporal logic specifications. Emerson et al. provide a good summary of the topic [9].

Depending on the model checker employed, specifications can be given in a variety of different logic formalizations, with modal and temporal logics being the most common. Typically, a developer will take a specification, create concurrency controls to satisfy that specification, and then run a model checker with the specification as input. If the check passes, then the specification has been satisfied. If it fails, then the process repeats, with help from the check results. Often times this help comes in the form of an error trace showing the exact interleaving that led to the violation. However, it is also possible that the problem is intractable due to excessive program size and thus an unmanageable state space. In that case no answer is returned, and the user must adjust the model in hopes of making the problem tractable. Chapter 8 will address model checking and specification in greater detail.

CHAPTER 3
Layered Statechart-based AI

Statecharts allow for an intuitive representation of the decision making structure of an AI. States represent the current thought process of the AI, while events correspond to events in the game envi‘ronment or decisions made by the AI. However, sophisticated AIs require a statechart that is far too complex to be readily understood. Limiting this complexity through the use of an abstract layering of modular statecharts was the core idea driving the development of *layered statechart-based AI* [35].

Extensive use of this representation has allowed us to advance the basic model in several significant ways, making it more adaptable and easier to employ. The first section addresses previously unexplored details of event-based communication and synchronous method calls. Following is a discussion of the abstract layers that make up a layered statechart-based AI. This will follow the original treatment [35], but will add more details regarding the nature of statecharts at each layer. Overall, this description augments and completes the theory of layered-statechart based AI. The following chapter will detail a fully constructed layered-statechart based AI, including a complete description of how it was implemented in an actual game.

3.1 Event-Based Communication

Layered statechart-based AI requires event-based communication between statecharts. While the Rhapsody semantics of statecharts [25] make it clear how to react to events within a single statechart, interaction *between* statecharts is not addressed, nor is this explicitly addressed in the model-based AI [35] work. This means that if multiple statecharts are meant to cooperate, we must first determine the means through which statecharts will interact.

To consider this problem, we draw a parallel with orthogonal regions in a single statechart. Orthogonal regions act concurrently and maintain their own state. When the statechart fires an event, the current state in each orthogonal region is checked and any matching transitions are triggered. While the transition ordering between orthogonal regions is undefined, all orthogonal regions must process the current event before the next event is considered.

With multiple statecharts, the situation is similar. Each statechart acts concurrently while maintaining its own independent state. However, there is no description of how events should be distributed and consumed. In the interest of having a clear semantics for layered statechart-based AI, we need to develop a cooperation semantics for interacting statecharts.

Intuitively, one would expect cooperating statecharts to adopt an event propagation model analogous to orthogonal regions. In other words, one would expect that when an event is generated in our layered statechart model, it should be sent to all statecharts, with the result that any statechart with a matching transition from its current state would fire that transition. Like orthogonal regions,

we should ensure that each statechart processes the current event before any statechart processes another. Conceptually speaking, this means that cooperating statecharts will act as though each is an orthogonal region of a larger statechart, while maintaining modularity at the implementation level.

To accomplish this, we must synchronize event firing in cooperating statecharts. This means that each statechart should see the same events in the same order, as though they were indeed orthogonal regions in one single statechart. This can be done by introducing an *AI event queue* for the set of statecharts, with all new events going to this queue. Figure 3–1 shows a group of statecharts connected by an AI event queue. When event A is fired, it is copied to the event queue for each individual statechart. The statecharts process the event, and send new events B and A to the AI event queue.

This type of event distribution employs a *broadcast* distribution model. Events are distributed asynchronously to all statecharts, leaving the generating statechart free to continue execution. Using broadcast as opposed to narrowcast is not a requirement, though it creates more loosely coupled statecharts. This is an important distinction, and will be discussed in more detail in Chap. 6.

3.1.1 Game Events

As game play proceeds, changes in the game state must be communicated to the AI. These events are *external* to the AI, and thus external to the statecharts that comprise the AI. Reacting to these events is necessary, but is not explicitly considered in the original model-based AI work.

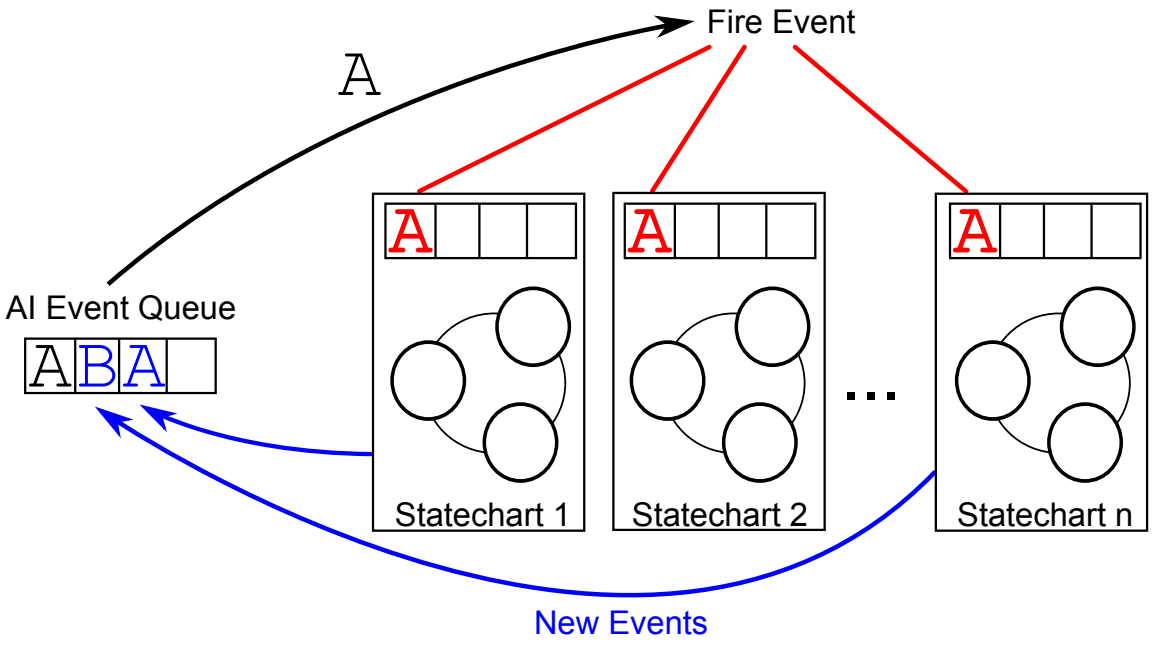


Figure 3–1: AI Event Queue for Cooperating Statecharts

The Rhapsody semantics describes external events as causing a *step*. Events generated by the statecharts in response cause *micro-steps*. After a step occurs, any resulting micro-steps are taken immediately. When there are no more micro-steps, the system is *quiescent*. Only when quiescence is reached can another step be taken.[1]

The statechart formalism assumes that transitions are instantaneous, but of course any real implementation takes non-zero time to execute transitions. We

[1] In rare cases, a system will not be able to reach quiescence due to infinite event generation. This would be considered a bug in statechart construction, much like an infinite loop is a bug in software development. In this case, the offending statecharts should be corrected to prevent infinite event generation

risk having an external event added to our queue before the system has become quiescent. This means that any implementation must distinguish between *internal* events causing micro-steps and *external* events causing steps.

We resolve this issue by introducing a second queue, called the *game event queue*. Any external events are sent to the game queue instead of the AI event queue. External events are processed as before, and all events generated in response are sent to the AI event queue. The AI event queue stores internal events that cause only micro-steps. The system is not quiescent until the AI event queue is empty. If new external events are generated while the system is not quiescent, they are stored in the game event queue and only fired when quiescence occurs.

Figure 3–2 shows both queues for cooperating statecharts. Here, events X and Y are generated externally and are placed in the external event queue, and initially the AI event queue is empty. Thus, event X is distributed to all statecharts, and the events B and A are generated in response. The AI queue is non-empty, and so the head of the queue, B will be distributed, followed by A, followed in order by any events resulting from B and A. Only when the AI queue is empty will event Y be distributed.

3.2 Synchronous Communication

When an action is being executed in a statechart, it may require information from other statecharts, without which the action cannot be completed. In this situation, asynchronous communication is not sufficient, and so we must allow some synchronous communication. This takes place using *synchronous method calls* between associated classes, where one class offers a synchronous call, and the

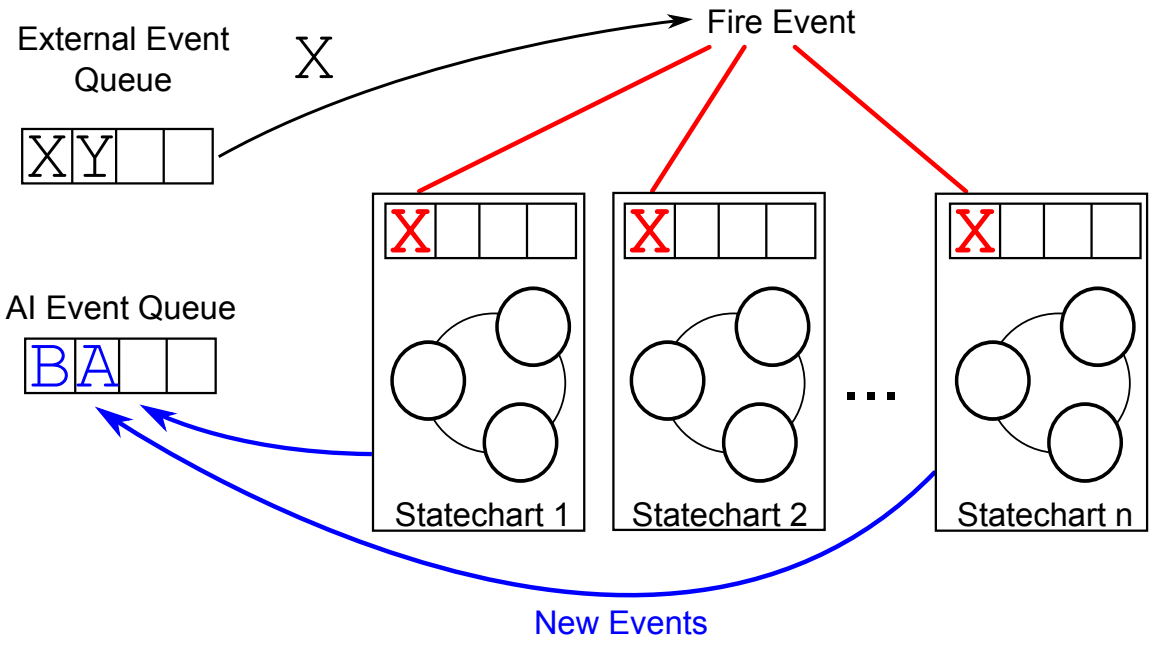


Figure 3–2: External Event Queue for Cooperating Statecharts

other associated class calls it. Synchronous calls allow for statecharts to gather information exactly when it is required. Unfortunately, achieving synchronicity introduces complications regardless of how it is implemented.

By allowing synchronous method calls, the layered-statechart formalism becomes complicated by the fact that the complete interaction profile is no longer expressed solely in the statecharts. Statecharts can now be connected through their associated classes, which does not appear in the statechart. Analysis and reuse of a layered-statechart based-AI must now consider the associated classes as well as the statecharts.

The obvious alternative is to allow statecharts to exchange data through request and callback events. A requester could send out a request event, then

transition into a blocking state where the only transition out is a callback event. The receiving statechart responds with the callback event, completing the communication. However, such an approach is problematic for two reasons. First, specific events must be sent in a specific order, implicitly defining a protocol. Violations of an implicit protocol are non-obvious, and introduce the potential for deadlock or failure if statecharts are mismatched or another transition interrupts the protocol. Secondly, passing paired events requires at least two micro-steps, and could take more if other statecharts insert events into the queue between the message pair. This delays calculation at the requesting statechart, and allows for the message passing protocol to be interrupted. If the protocol uses a blocking state, then interrupting events could be lost entirely. To eliminate these sources of error, we advocate the use of synchronous method calls over request and callback protocols.

To simplify consideration of synchronous calls, we adopt the rule that a synchronous method call should never change modal properties of a statechart. This means that offered synchronous calls are primarily 'getters'. Since modal properties are not modified, event passing completely determines state, avoiding the possibility for hidden or non-obvious behaviours in the statecharts. Similarly, we adopt the rule that synchronous communication should never be done through event passing, avoiding implicit protocol definition. This also sidesteps the need for protocol management at the statechart level using artefacts such as protocol state-machines.

3.2.1 Payloads

Events in a statechart are usually thought of as being a label and nothing more. However, any given implementation might allow for events to include *payloads*. This provides a convenient method to include pertinent information about an event occurrence as it happens. Including payload allows the receiving statechart to process information about the event immediately. The alternative would be for the receiving class to make a synchronous call to fetch the information.

Typically, the payload would consist of either a primitive or an object reference. As an example, a sensor generating the event `ev_item_spotted` could include as payload a reference to the instance of the item. Analyzers could then directly and immediately investigate properties of the item, allowing the analyzer to perform needed operations without any additional communication.

3.3 The Layers

Layered statechart-based AI is comprised of individual *modules* that cooperate to produce the behaviour of the NPC being controlled. Each module, consisting of a statechart and its associated class, implements a single aspect of the overall behaviour. Modules are grouped into *layers*, an abstract categorization based upon functionality and purpose. The layering provides a structure for modules during the design process, and helps inform a designer of the intended purpose of a module. Layers exist solely as a design artifact, having no further meaning at run-time.

Layers typically contain one or more modules, though they can be empty. As described in §2.1.3, there are eight layers in total. Going from input to output,

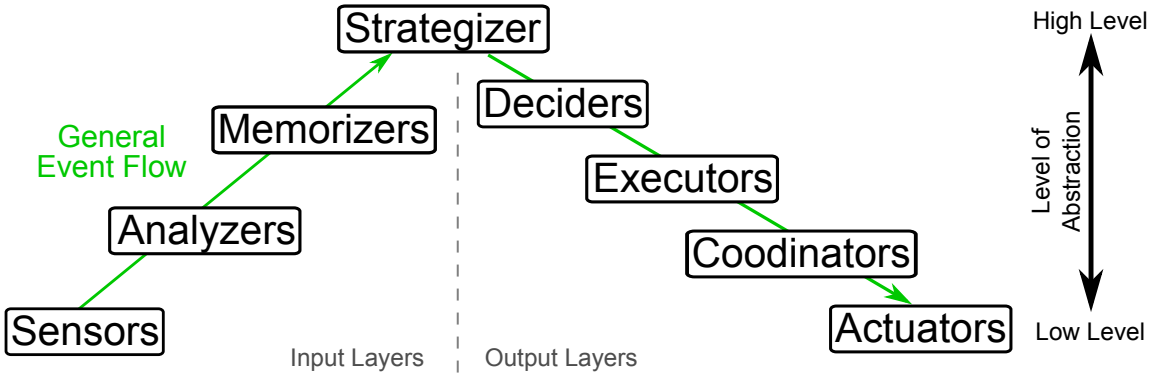


Figure 3–3: Layers in the Layered Statechart-based AI Formalism

they are sensors, analysers, memorizers, strategizers, deciders, executors, coordinators, and actuators. The remainder of this chapter will clarify the role and operation of modules at each layer.

Figure 3–3 summarizes the layers and indicates general event flow. The layers are arranged in a pyramidal fashion. This reflects level of abstraction: layers at the bottom of the figure have a low level of abstraction with respect to behaviours, while layers at the top have a high level of abstraction.

In general, the direction of event flow is as shown in Fig. 3–3. While directionality is not strictly enforced, our experience to date has not required event flow in the reverse direction. This means that events generated by statecharts move from input towards the strategizer, and from the strategizer towards the output. For instance, an event generated by a sensor would be consumed by an analyzer, and decider events would be consumed by strategizers. In some cases, events can be consumed at the same layer, or can skip over layers. This will be discussed in more detail as the layers are described.

To simplify high-level statecharts, we allow a form of *subsumption* [5] whereby input modules can send data to output modules directly, skipping over higher-level intermediary modules. This would allow sensed data originating from a sensor to be be received directly at an executor, for example. Allowing this type of communication is an important optimization, and will be described in detail in Chapter 5. Note that this process still respects the directionality of general event flow.

3.3.1 Input Layers

Input to the AI begins with the *sensor* modules. The primary task at this layer is to create events for the statecharts based upon spontaneous changes in the game state. This means that sensors have a special role in the AI, which is to act as the input portion of the interface to the game at large.

If the game uses events in its own internal logic, then the associated class of a sensor can perform a simple mapping, registering to receive game events and creating corresponding statechart events. If the game is not event based, then the sensor's associated class must proactively access the game state through active polling. When a relevant change is detected, the sensor generates a new statechart event.

Events generated by a sensor tend to be quite simple in nature, such as `ev_player_spotted`, or `ev_health_changed`. Typically, they include as payload a reference to the game object that caused the event. This reference can then be used by higher level modules to further interpret the game state. While this does enable other modules to access the game state, Sensors are unique in that they are

the only modules that react to spontaneous changes in game state. Other modules that access the game state through references only query it for obtaining current game state information (most typically to evaluate a guard).

To form a higher level view, these simple events are processed and correlated by *analyzers*. A sensor might spot a new player, but an analyzer will determine if that player poses a threat, if he is a ranged or melee opponent, and so on. This allows the AI to build the understanding of the game state necessary to make correct decisions.

Often, analyzers employ guarded transitions to clarify sensor data, e.g., a transition labeled with `ev_player_spotted`[*isEnemy(player)*]/`ev_enemy_spotted` would determine if a newly spotted player is an enemy. This means that the analyzer accesses the game state through the reference provided by the sensor, allowing it to gather additional information to complete its analysis. This provides a convenient way to translate sensed data it into a more complete understanding of the game state.

To build a high level view of the game, analyzers may need to correlate results from other analyzers. While this could be done through the addition of a 'super-analyzer' layer, this is not practical—what if we need to further analyze 'super-analyzed' data? Instead, we explicitly allow analyzers to accept input from other analyzers as well as sensors, allowing for event correlation. This means that events can be analyzed an arbitrary number of times, such that the AI reaches the necessary level of understanding of the game state.

The next layer contains *memorizers*. Receiving events from both sensors and analyzers, memorizers store information about the game state. An Enemy Memorizer module receiving an `ev_enemy_spotted` event from an analyzer would store a reference to the spotted enemy. Unlike other modules, memorizers do not generate events. Instead, other statecharts access memorized data through synchronous calls to the associated class.

The memorizer layer is not unique in its ability to memorize data, and it is common for modules to store information in their associated class during normal operation. If, however, several modules require the same data, if memorized data is large and requires special treatment, or if there is a complicated set of conditions to decide how data is stored, then it is useful to create a memorizer. This encapsulates the memorization process, makes memorized data available to all statecharts, and allows for optimization in the associated class. To be clear, the primary role of the memorizer is to modularize and encapsulate storage of data.

To summarize: changes in the game state are monitored by sensor statecharts. When a relevant change is detected, an appropriate event is generated and passed to the analyzers. These in turn examine the events generated by the sensors (and other analyzers), possibly querying the game state through payload references in sensor events. Anaylzers make high-level conclusions about the current game state (e.g., there is an enemy nearby and it is in melee combat range) and pass these on to the strategizers so that they can choose a course of action. Sensed and analyzed events can also be received by memorizers, building a history to be used in later decisions.

3.3.2 The Strategizer Layer

The *strategizer*[2] forms the highest level decision making structure of the AI. Receiving input from analyzers and sensors, the strategizer uses this knowledge of the game state to choose the most appropriate operating *goal*[3] for the AI.

At the strategizers, goals tend to be quite abstract. Fleeing from enemies, gathering supplies, and exploring the world would all be reasonable goals. Events such as `ev_enemy_spotted` might cause the NPC to transition from the exploring mode to the fleeing mode.

Usually, there is only one strategizer in an AI. This is due to the potential for confusion caused by multiple strategizers pursuing independent goals simultaneously. This could cause conflicting commands being sent to the output layers, leading to negative outcomes such as oscillation, incorrect action ordering, or more generalized failures. Use of a single strategizer means that only a single goal is being pursued at any time, eliminating the possibility of conflicting goals, and improving comprehension by providing a single source for the current behaviour.

---

[2] In the original paper [35], these were called strategic deciders, while statecharts at the next layer were called tactical deciders. Instead of having two layers whose name included decider, we rename 'strategic decider' into 'strategizer', allowing 'tactical decider' to be unambiguously shortened to 'deciders'.

[3] Goal is used in the most common sense of the word, and is unrelated to goal-oriented architectures.

3.3.3 Output Layers

High-level goals sent by the strategizer must be translated into lower-level commands understandable by the different actuators. 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 or through subsumptive communication directly from the sensors and analyzers.

Interpretation of high level goals is done at the *decider* layer. In our formalism, we use exactly one decider for each goal. The decider chooses how best to execute that goal using knowledge gathered about the current game state. For instance, if the strategizer chooses the goal of engaging in combat, the decider will choose how to perform combat (e.g., ranged combat, melee combat, taking cover, and so on). Decisions at this level are still highly abstract.

The next output level, the *executors*, map the decisions of the deciders to events that the actuators can understand. Decisions may require different actuations based upon game state. For example, a decision to engage in ranged combat involves in some order loading a weapon, selecting a target, getting a clear line of fire, shooting at the target, and so on. The executor chooses the actuators that will accomplish this task based upon the current game state. Moving, for instance, might only be necessary if no clear line of fire currently exists.

If an executor needs to perform a series of actions that is already implemented by another executor, it is explicitly allowed for executors to use other executors to complete their task. Again following the combat example, there might be a move-to-cover executor that is used by the ranged combat executor to first move to

a cover point, followed by the ranged combat executing its fire on target actions. Along the same lines, if a decider requires use of another decider, it is permissible for a decider to call a peer decider.

In the typical case, executors communicate directly with the actuators. On occasion, the series of events generated by executors may result in suboptimal or inefficient actuations. In such cases, a *coordinator* module is used to resolve the problem. One example occurs with relative actuations. Imagine a character wants to face north, but the only available actuations are turning left or right. The most efficient solution could be to turn left or to turn right depending on the direction the NPC is currently facing. A *Turn Coordinator* would be tasked with selecting the optimal turn actuation.

The final layer is comprised of *actuator* modules. These form the output half of the game interface, and are responsible for sending actions to the game. For example, if an executor decides that moving is the appropriate action, then an `ev_move` event will be sent to a *Move Actuator*. The actuator, through its associated class, calls the method in the game that causes the NPC to move.

To summarize: each high-level goal from the strategizer has a single decider. A decider subdivides its goal into high-level actions (e.g., ranged combat might include the actions taking cover, firing at an enemy, etc.). Each high level action is implemented in a single executor. Executors map high level actions into a series of actuations that together make the high level action occur. Coordinators refine these actuations if necessary, and then the actuators (one for each game action) signal the game and cause their specific actuation.

CHAPTER 4
Implementation

Creating a fully functional prototype AI is the next step in demonstrating the viability of layered statechart-based AI. In this chapter, we present a complete implementation of an AI designed to control a simple squirrel NPC. The value of this is in the creation of a proof of concept, demonstrating that implementation of an AI for a simple character is not burdensome in either scope or complexity.

The intended usage of this AI is as a test case for later research into layered statechart-based AI. The squirrel will be referred to extensively in later chapters, as it was used to test reuse, verification, and variation techniques. The first half of this chapter exhibits each statechart used in the squirrel, then illustrates how the modular statecharts interact to realize core squirrel behaviours.

The squirrel AI was implemented in the game *Mammoth* [36], a massively multiplayer online game (MMO) research framework. Written in Java, it provides an implementation platform for academic research related to multiplayer and MMOs in the fields of distributed systems, fault tolerance, databases, networking, concurrency, but also artificial intelligence, content generation, and others. The second half of this chapter provides details on Mammoth was extended to support layered statechart-based AI, and how the squirrel AI was implemented.

4.1 Building the Squirrel AI

In the setting of actual video game development, the requirements for the squirrel AI would be described at a high level by a designer. A set of loosely defined goals such as 'make it so that the squirrels run from the player' would be the norm. The AI developer would then use their own judgement and expertise to translate these goals into an implementation. The completed AI would be reviewed by the designer, and iterated upon until the designer decides the AI satisfies the vision for the game.

The squirrel NPC is intended to act as a background character in Mammoth. The behavioural objective is to act observably similar to a real squirrel, such that a player recognizes the behaviour as being squirrel-like under casual observation. Rather than a simulation of actual squirrel behaviour, it is enough that our squirrel NPC appears to exhibit some of the more obvious squirrel behaviours.

The subset of squirrel behaviours we will design are moving about an area, running from humans, and gathering acorns. This requires that the AI have appropriate knowledge of the current game state and game context, such that these behaviours will be performed appropriately. To add a game-play challenge, squirrel NPCs have a finite pool of energy that drains when moving and can only be restored by eating. Thus, the squirrel has the additional task of managing and restoring energy.

4.1.1 Requirements

Our goals for the squirrel AI stand in as goals supplied by a designer. The overall behavioural objective is to act observably similar to a real squirrel, such

that a player recognizes the behaviour as being squirrel-like under casual observation. This breaks down into a set of high level objectives that describes desired squirrel behaviour. The squirrel should:

- Interact with players by fleeing from them.
- Find and collect acorns.
- Replenish their energy by eating acorns.
- Wander the game world when not performing any other task.

Satisfaction of these goals would provide for an interesting background character. The squirrel will perform the basic task of collecting acorns, and will interact with the players to a basic degree. Indeed, this depth of behaviour is greater than that found in typical non-boss enemies in the successful game 'World of Warcraft'. In other words, the design goals for the squirrel AI are a reasonable approximation of the complexity of a non-major NPC for a modern computer game.

Development Requirements

In translating design goals into implementable requirements, an AI developer is given considerable leeway. While the final character must satisfying the overall design goals, the AI developer typically has room to add their own creative vision when defining specific behaviours. In making the squirrel AI, we exercised this flexibility to develop the following implementation details:

1. Squirrels have a low and high threat radius used to determine if a player character is dangerously close to the squirrel.
2. Squirrels will prioritize fleeing from high threats at all times.

3. When a squirrel is hungry, it will attempt to collect an acorn.
4. If there are only low threats, squirrels will flee from them except in the case they are very hungry. Thematically, this means a starving squirrel will overcome mild fear to gather food.
5. When a squirrel picks up an acorn, it will eat it if hungry
6. When a squirrel is neither hungry nor threatened, it will simply wander the environment.

The use of the high and low threat radius provides an interesting detail in the squirrel behaviour. While it makes the implementation more complex, it provides a more fine-grained interaction with the player. The ordering of priorities assigns the relative importance of the behaviours similar to that of an actual squirrel.

4.1.2 AI Modules

The squirrel AI is designed using the layered-statechart approach. The AI logic is decomposed into a number of small AI modules that together interact to produce squirrel behaviours. By employing small statecharts, the complexity of each statechart is limited, and thus each statechart can be readily understood using simple inspection.

A behavioural decomposition is reached through a consideration of how these behaviours break down in terms of the described layers. For instance, gathering food and fleeing from players requires seeing food and players, and thus we need a sensor that can sense these game objects. The exact choice of modules is a design decision and is non-deterministic in nature, but breaking down behaviours in

an intuitive manner will lead to a more easily understood design. The following modules represent our choices in how to modularize the squirrel behaviours.

Sensors

The squirrel uses two sensors to read the game state. The first, called the Energy Sensor, is given in Fig. 4–1. This statechart maps a continuous integer value energy into three distinct levels: high energy, low energy, and very low energy. Due to the event-less transitions, the statechart will check for changes in guard conditions every frame. When a change in energy level triggers a transition, the appropriate event is generated. The values that demarcate energy levels are parameters in the associated class.

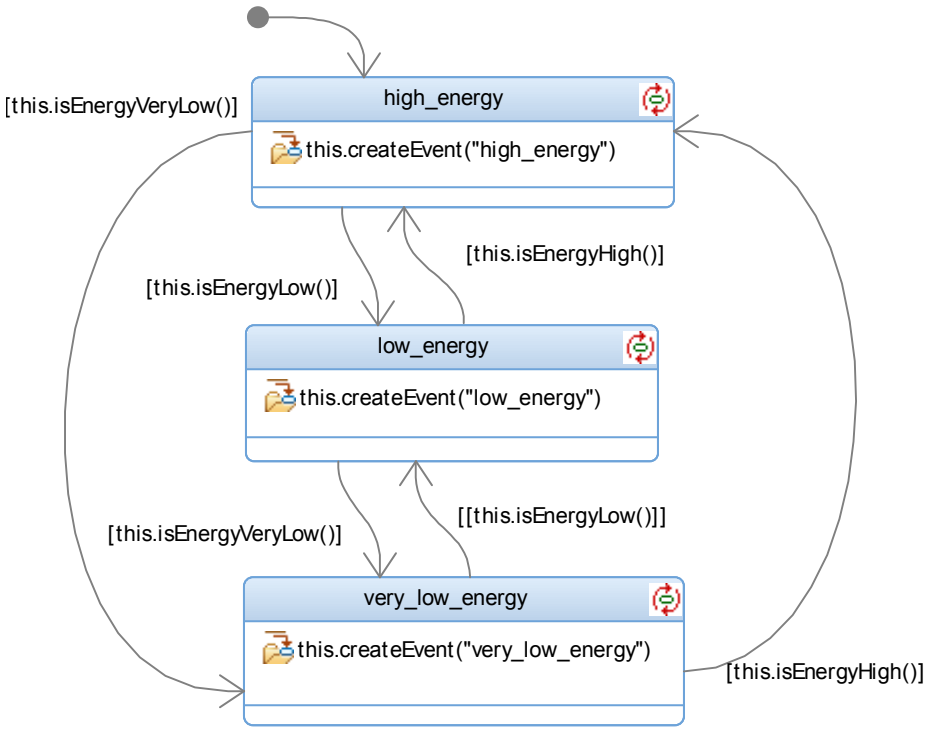


Figure 4–1: The Energy Sensor

The other sensor tracks players and items. The associated class is actually a simple listener that maps game occurrences into AI events. The statechart associated with this sensor is trivial, consisting of only a single state that reacts to no events. The game delivers updates to this sensor based upon the NPC's area of interest in the game. As game objects move in and out of this area, the associated class creates events *i_see_player* and *i_dont_see_player* for players, as well as *i_see_item* and *i_dont_see_item* for items.

Analyzers

The lone analyzer determines if other characters in the game are threats. The Threat Analyzer, shown in Fig. 4–2, uses the rule that any non-squirrel character qualifies as a threat. It maintains two threat levels as parameters: high threat range, for enemies that are very close; and low threat range, for enemies that are somewhat close but not in high threat range. The two threat levels prevent use of a binary analyzer. Instead, input tracking is separated from determination of threat levels through use of parallel states.

It could be argued that the Threat Analyzer acts as a memorizer, since it remembers and tracks current threats. However, the stored information is essential to the analysis task, and it makes little sense to have the analyzer ask a memorizer for status upon every update. Ultimately, storing the information in the analyzer is a useful optimization. While this may seem to weaken the boundary between analyzers and memorizers, it is important to note that the memorized data is not shared through any synchronous calls, and all event generations are directly related to the analysis task.

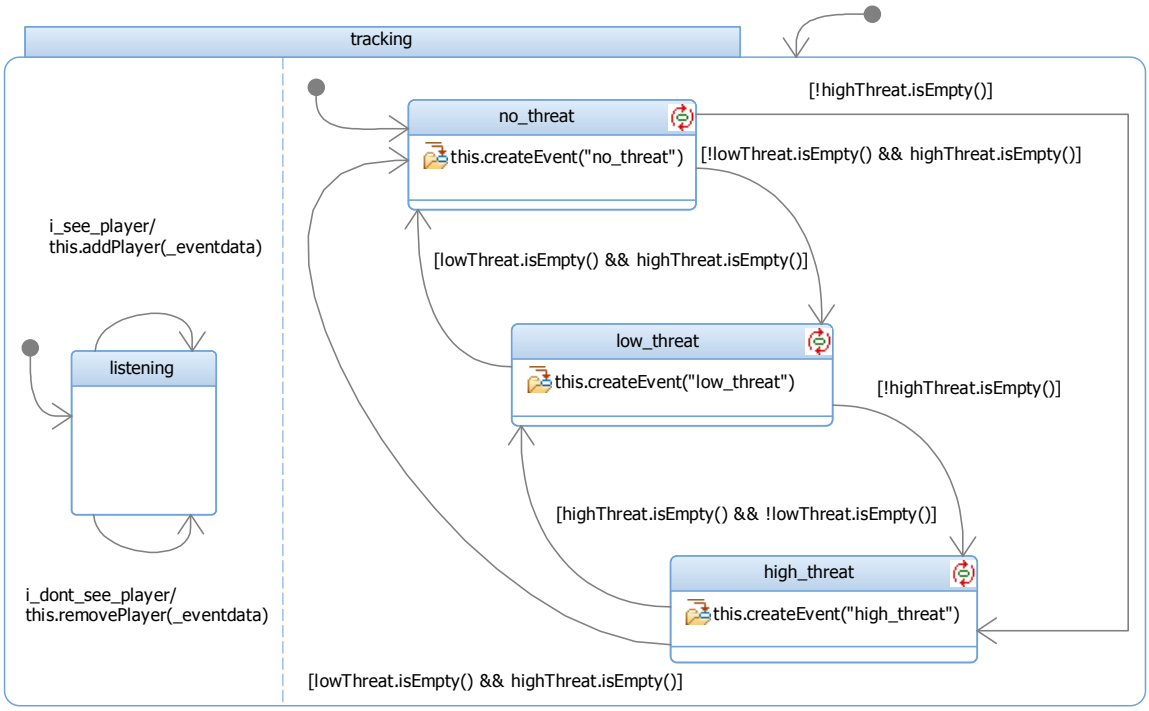


Figure 4–2: The Threat Analyzer

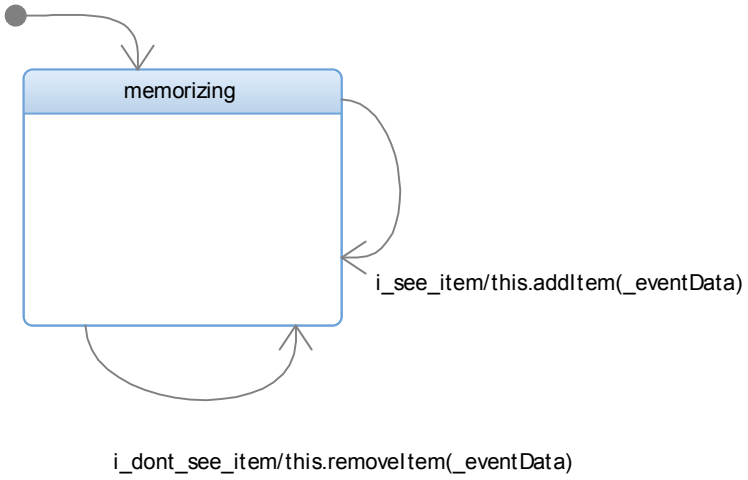


Figure 4–3: The Key Item Memorizer

Memorizers

As items are spotted, the squirrel must keep track of what items it can see. This is done by the Key-Item Memorizer, shown in Fig. 4–3. This memorizer, rather than memorize all *i_see_item* events, only remembers items that match the key item parameter. In the case of the squirrel, the key item type is acorn. As a memorizer, the associated class offers the synchronous call `getKeyItemList()`, which returns the list of key items that are currently visible.

Strategizers

In Fig. 4–4 we see the brain for the squirrel AI. This statechart is the highest-level decision maker in the AI, as it chooses the current high-level goal from one of wandering, getting food, or fleeing. Initially, the squirrel will always begin wandering. During this time, the squirrel will flee from all threats. Once the squirrel becomes hungry, the high-level goal getting food will be selected. If the squirrel is close to starving, it will ignore low threats in its desperation for food.

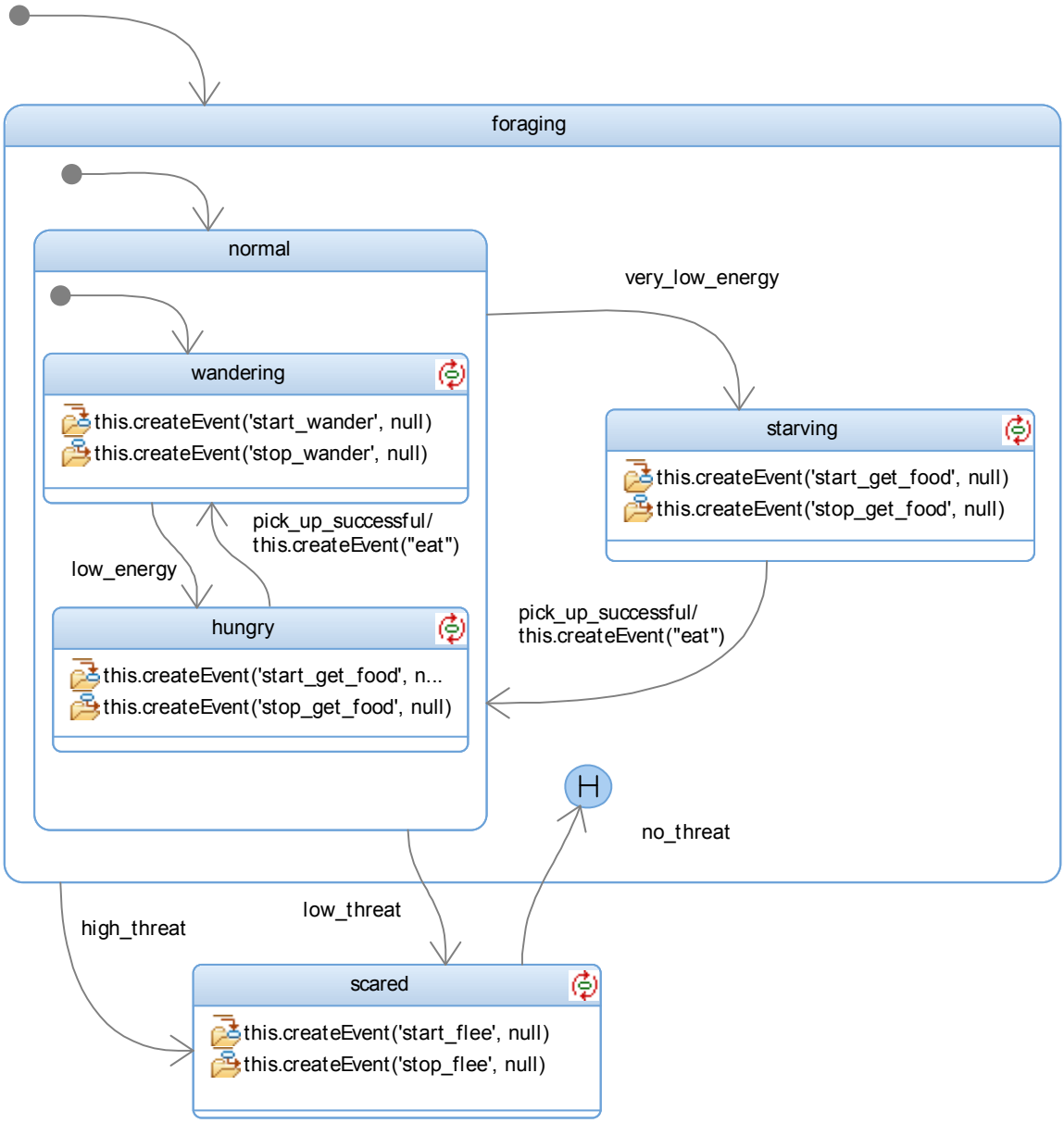


Figure 4–4: The Squirrel Brain

Tactical Deciders

The Flee Decider, shown in Fig 4–5, chooses how to implement the high-level goal to flee. It does this by tracking the current threat level. When the order is given to start fleeing, the appropriate method is called to begin fleeing from a low threat target or a high threat target as appropriate.

The Eat Decider, shown in Fig. 4–6, is more complicated. If food is visible, then the squirrel will try to pick it up. However, food may not be visible, and so the squirrel may have to further explore the game world by wandering. Additionally, pick ups may fail for a variety of game reasons (e.g., another squirrel has taken the acorn first), and so picking up food may have to be attempted multiple times. If during this process, all visible food is taken by other squirrels, then a `no_key_item_visible` event will be generated and the squirrel will return to wandering.

It would be possible to improve this behaviour by having the squirrel remember acorns spotted in the past. This would require the squirrel to distinguish between visible and non-visible food, and to update the lists as the squirrel moves about.

Executors

Part of eating involves analyzing the game state to determine when eating is appropriate. The Eat Executor shown in Fig. 4–7 listens to pick up events to determine if the squirrel is currently carrying an acorn, and listens to energy events as well. If the squirrel becomes very low on energy, the executor creates the *eat* event.

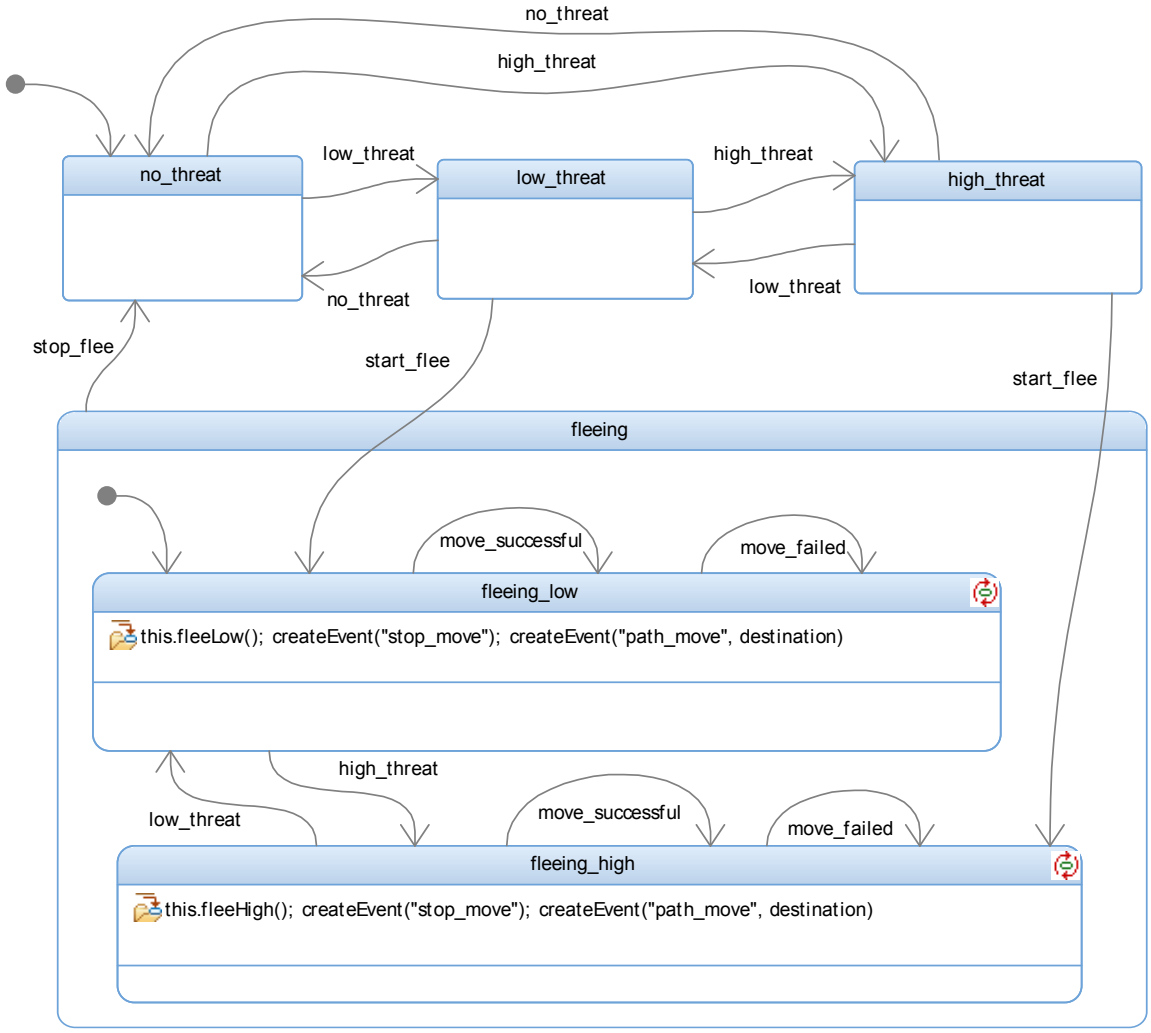


Figure 4–5: The Flee Decider

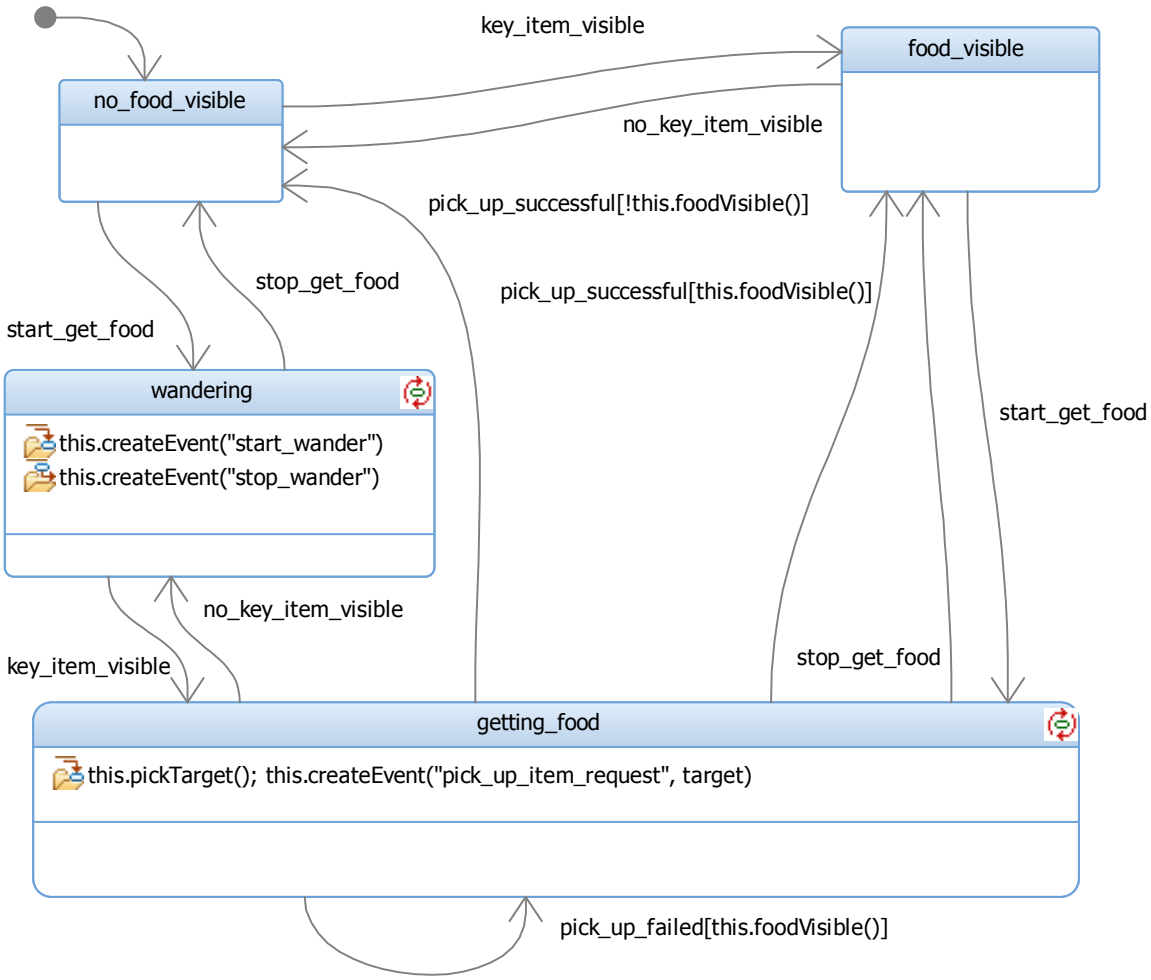


Figure 4–6: The Eat Decider

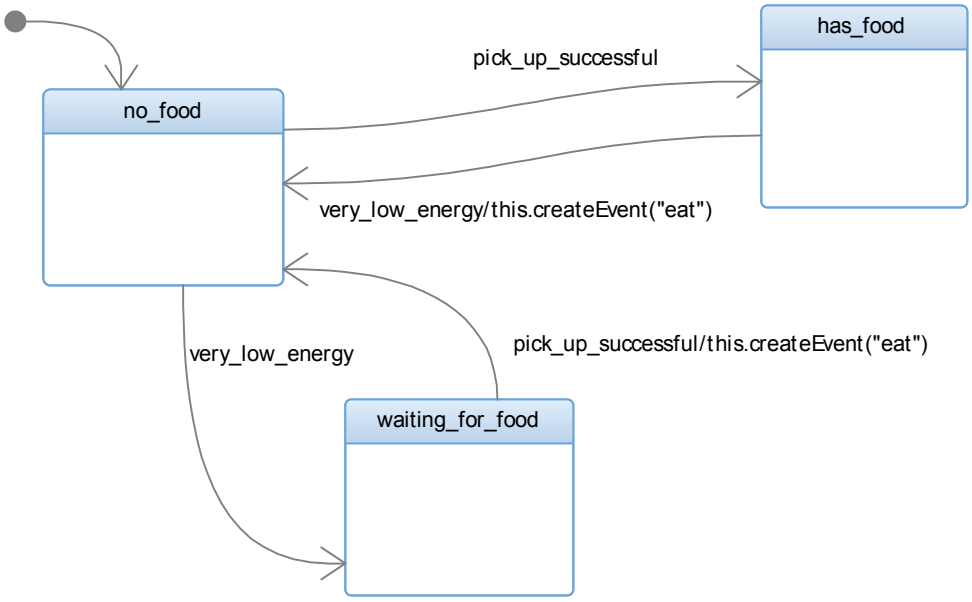


Figure 4–7: The Eat Executor

Wandering behaviour is implemented by the Wander Executor, shown in Fig. 4–8. When the squirrel wants to wander, this statechart selects wander destinations and moves to them. It does so by moving to a random location, then waiting for a brief period. The *time* event, generated by a timer in the associated class, determines how long the squirrel should walk, and how long it should pause. The wander timer and wander radius are parameters of the module. This behaviour is quite simple, and could be improved by implementing steering behaviours [58], exploration with the goal of discovering the entire map, or intelligent wandering where the squirrel first wanders to likely food sources such as the base of trees. That being said, it is sufficient for our purposes of creating a typical minor NPC.

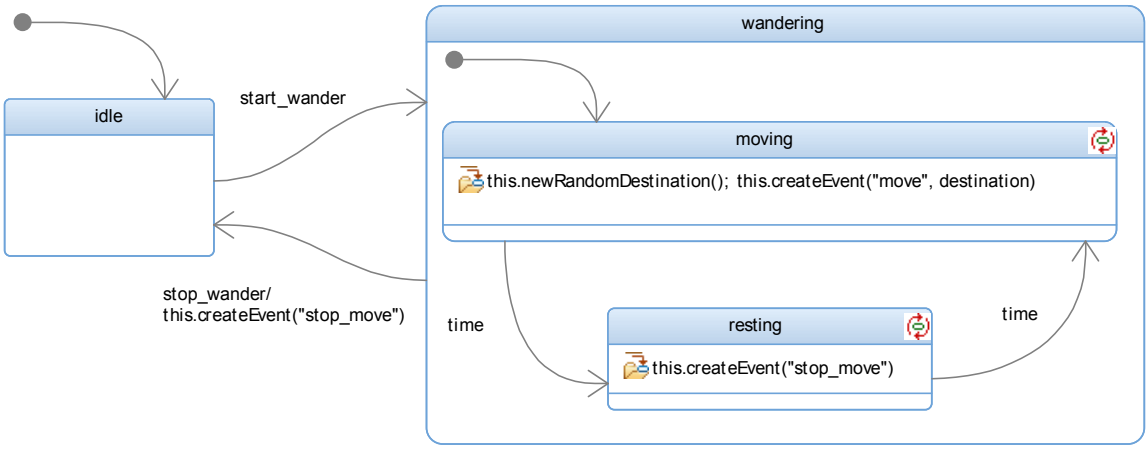


Figure 4–8: The Wander Executor

The final executor is the Pick Up Executor, shown in Fig. 4–9. In Mammoth, an item can only be picked up if it is close to the player. This executor manages that by first moving to the item, and only then picking it up.

Coordinators

No coordinators were necessary in the implementation of the Squirrel AI.

Actuators

The Pick Up Actuator, shown in Fig. 4–10, enacts pick up commands. These can fail for a variety of reasons, the most common of which is that the item has been picked up by another player. Because of this, the implementation is that of a feedback actuator, and thus it senses whether or not the pickup was successful.

The Eat Actuator is very simple. The sole state in Fig. 4–11 is that of waiting to be told to eat. Eating does not fail in the Mammoth implementation, and thus feedback is unnecessary.

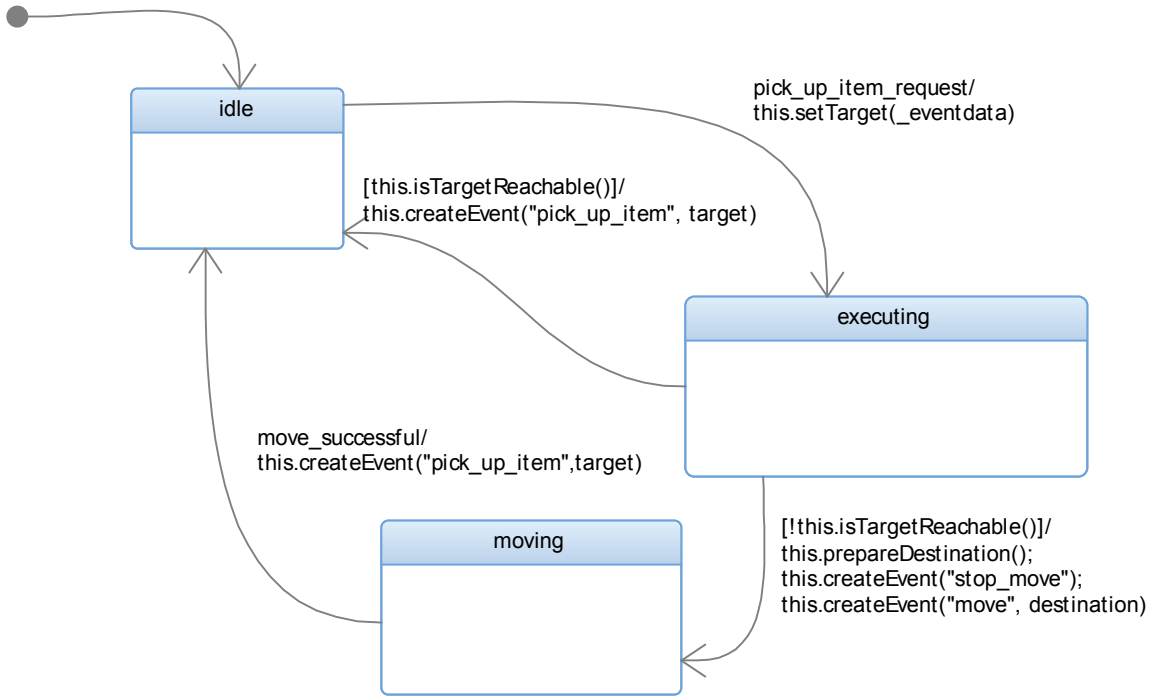


Figure 4–9: The Pick-up Executor

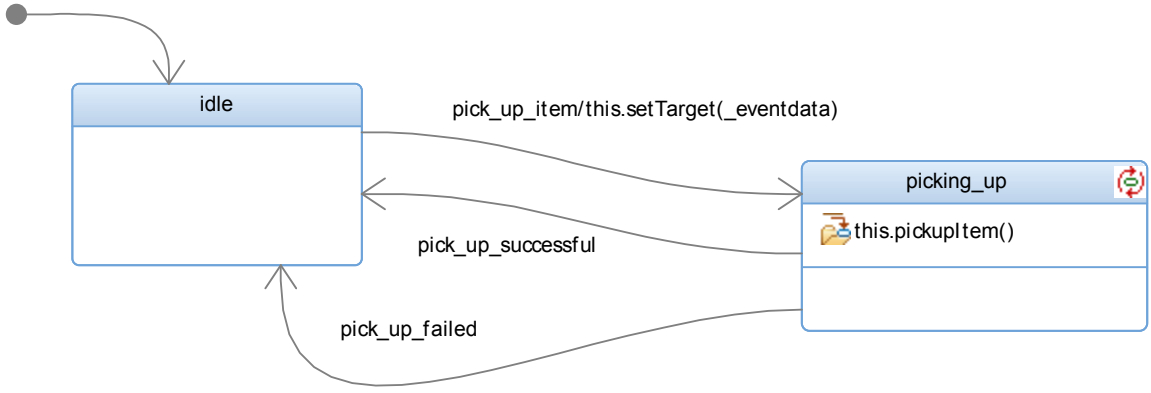


Figure 4–10: The Pick-up Actuator

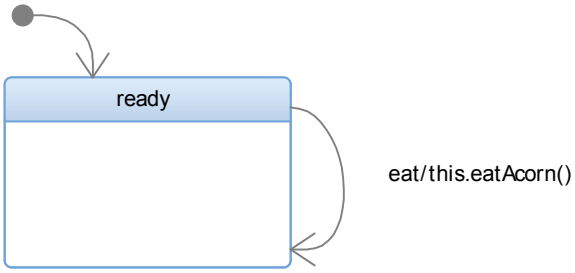


Figure 4–11: The Eat Actuator

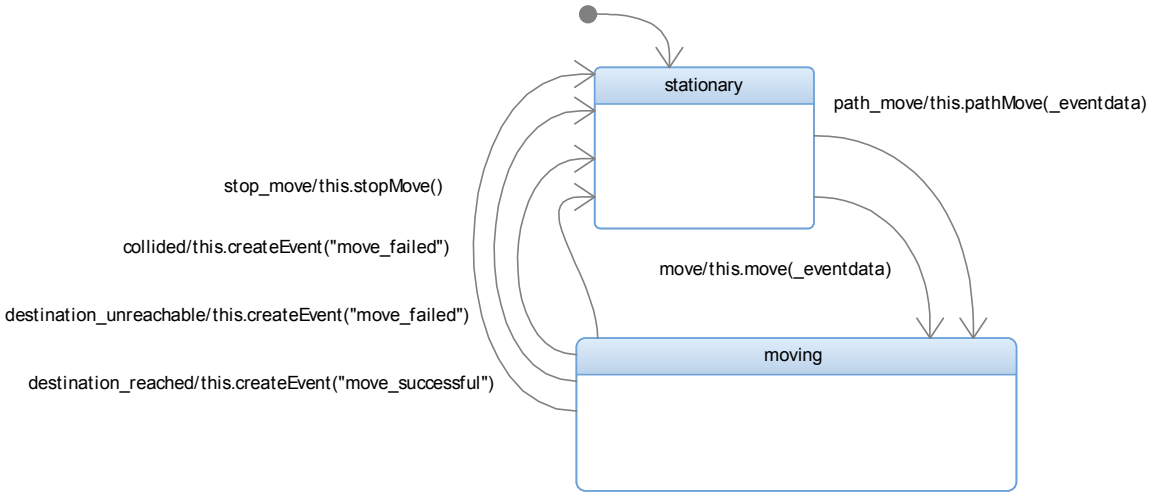


Figure 4–12: The Move Actuator

Finally, the Move Actuator, shown in Fig. 4–12, enacts movement commands. It can move in a straight line, or using path finding depending on the event being sent. Like the Pick Up Actuator, it is also a feedback actuator and detects a number of different move failures. However, the output is simplified to either say move succeeded or move failed.

4.1.3 Producing Behaviours

Together these modules interact to produce the overall squirrel behaviour. Using sequence diagrams, we will demonstrate how the eating behaviour is

generated by a squirrel in a typical environment. All event occurrences are communicated as broadcasts, but are shown as direct communications between the outputting and inputting statecharts.

When the squirrel begins execution, the Squirrel Brain chooses wandering as the default goal. This leads to the chain of events shown in Fig. 4–13. Other statecharts generate initial events not shown here (e.g., the Energy Sensor will create a `high_energy` event). We assume that initially the squirrel cannot see any food. Thus, the default behaviour is to wander, resulting in the Move Actuator executing the command at the game. Since it is a feedback actuator, the move actuator waits for a response from the game and communicates it to the rest of the AI.

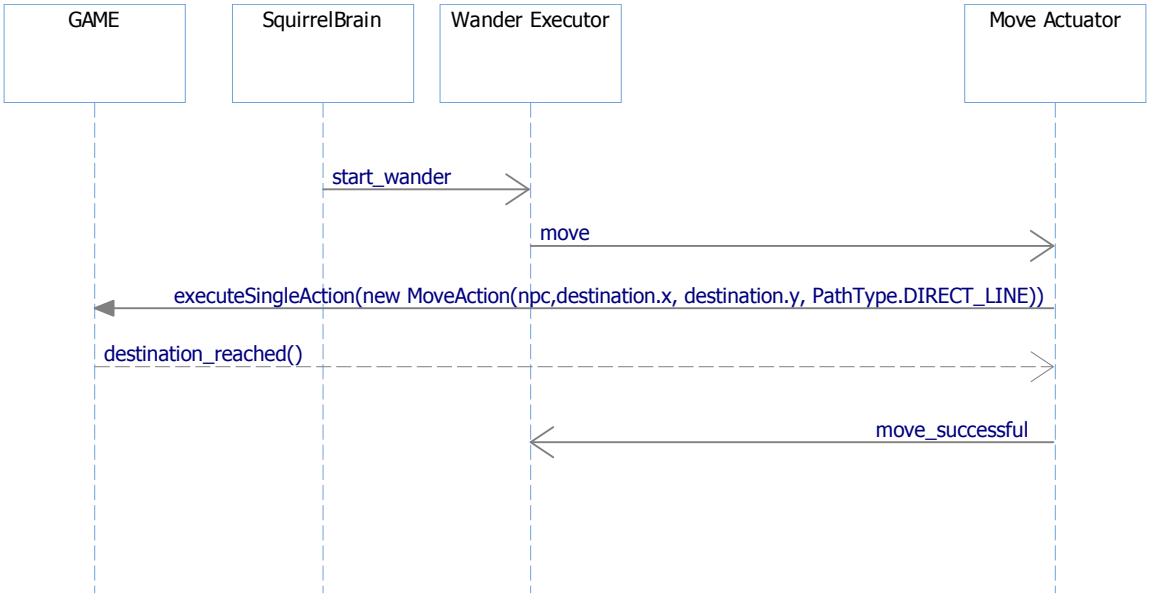


Figure 4–13: Initial Wandering Sequence Diagram

After wandering for a time, let us assume that the squirrel eventually spots food, causing the chain of events seen in Fig. 4–14. Spotting the food causes the game to call the associated class of the Mammoth Listener, resulting in the creation of a new game event. This will be received by the Key Item Memorizer, and passed along to the Eat Decider. The Eat Decider learning this directly from the memorizer is characteristic of subsumption. Since the Squirrel Brain prioritizes wandering, the squirrel brain will continue wandering until the squirrel reaches a low energy state.

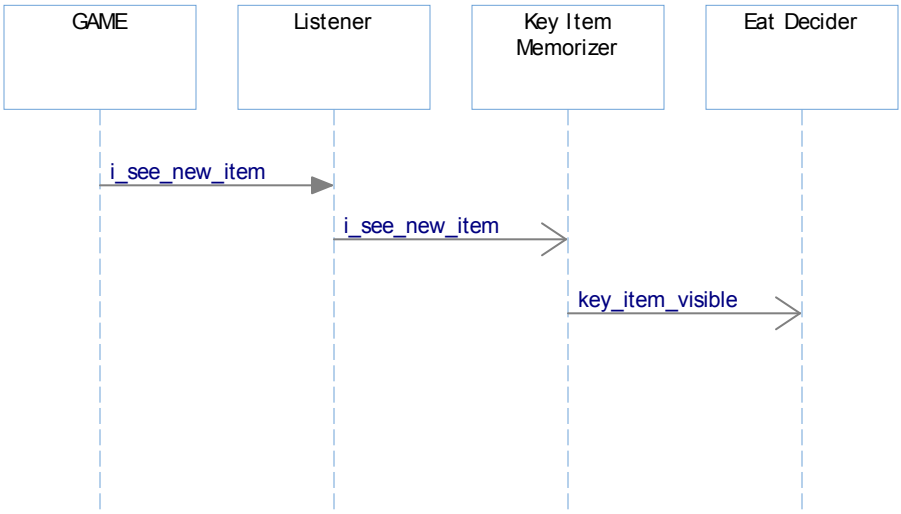


Figure 4–14: Food Spotted Sequence Diagram

Upon reaching low energy, the squirrel will attempt to gather the food it has spotted. Figure 4–15 shows the event sequence that will occur if the food is within reach. If the food was too far away, the Pick-up Executor would first have to move the NPC to reaching distance of the pickup target. One can infer the chain of events this would generate by referring back to Fig. 4–7. Again, there is a call and

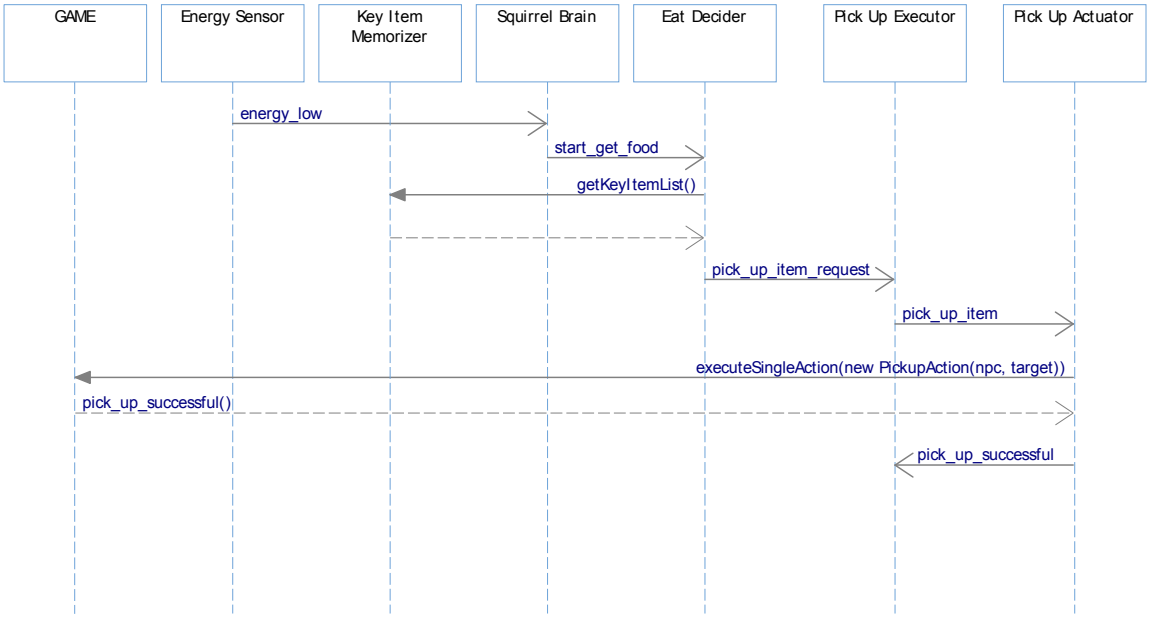


Figure 4–15: Low Energy Sequence Diagram

response pair between a feedback actuator and the game, which is characteristic of feedback actuators.

After the successful pickup, the squirrel now has food. Upon reaching a very low energy state, the squirrel will eat the acorn as shown in Fig. 4–16. The first event, `pick_up_successful` is the same event generated at the end of the pick up. Since it is broadcast, that single event is received by both the Pick-up Executor and the Eat Decider. The Eat Actuator is not a feedback actuator, since eating cannot fail, and so there is no callback. Instead, the Energy Sensor will see that energy is restored and create a corresponding event. The eating behaviour is now complete, and the Squirrel Brain will return to wandering.

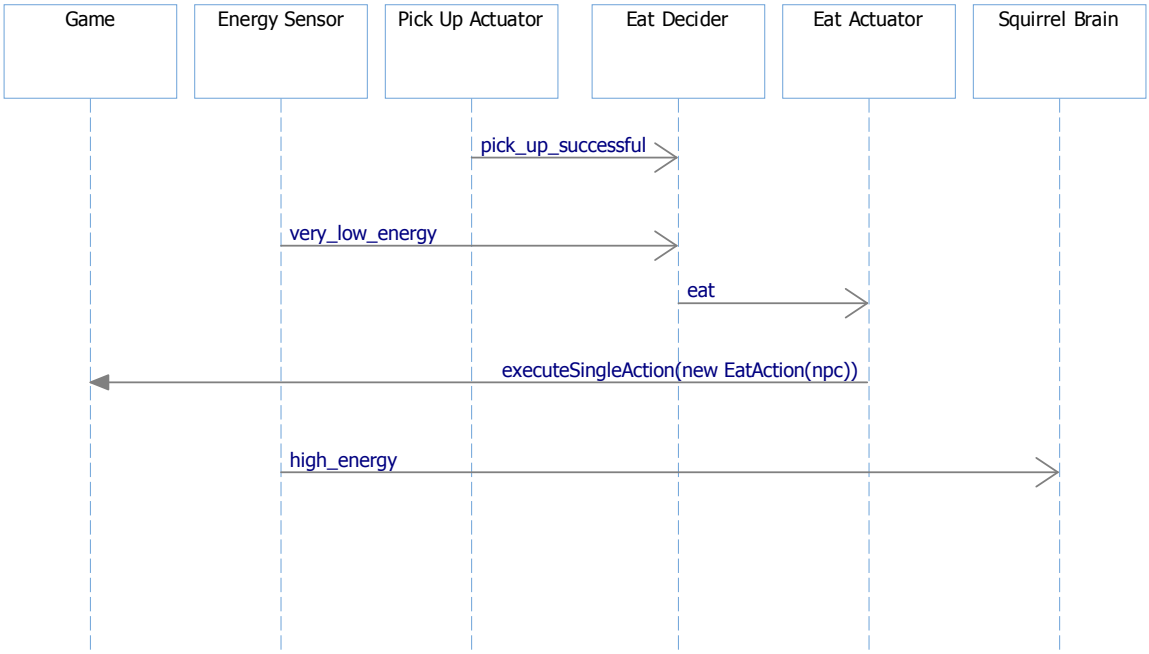


Figure 4–16: Eating Behaviour Sequence Diagram

4.2 Mammoth Implementation

In Mammoth, players take control of a game character called an *avatar*. A game session consists of moving around in a virtual 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 is constructed as a set 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.

4.2.1 NPCs in Mammoth

All non-player characters in Mammoth are controlled by an *NPC Manager*. On each pass of the game loop, the NPC Manager executes the internal logic of the AI for each NPC. This will determine what action, if any, the NPC will attempt. Common actions in Mammoth include moving, picking up or dropping objects, and sending messages.

Each AI is represented as a *role*, which act as high level containers for the various behaviours of each AI. Examples of roles would be a shopkeeper, a city guard, a knight, or a bandit. Multiple NPCs assigned the same role each maintain their own instance of the role, with their own internal memory. Behaviour is component-based and implemented as *tasks*; examples include moving, wandering, and eating. Each role contains a number of tasks such that the expressed behaviour meets the requirements of the role. Thus, an AI in Mammoth is synonymous with a role, and behaviours are synonymous with tasks.

4.2.2 Adding Statecharts to Mammoth

The architecture of AIs in Mammoth provided an excellent starting point to add layered statechart-based AI. Each statechart realizes a behavioural module, which corresponds to the modular format of Mammoth tasks. The goal was thus to enable tasks to use statecharts as their decision making structure.

Since the role contains all tasks for an NPC, this was the logical place to implement the event queues described in §3.1. A queue for AI events and for game events was added to the role, along with methods used by tasks to add events to the queues. When the role is given execution time, it delivers events from the game queue to the statechart tasks, accumulating and prioritizing AI events (micro-steps) from the AI queue in accordance with the Rhapsody semantics and our approach to cooperating statecharts.

To update statechart-based tasks, an `eventNotification` method was added to tasks. When the role had an event to deliver to a statechart, the role called the event notification method and delivered the event. Previously, tasks in Mammoth provided an `update` method, which received a time-slice from the role. This timing information was of little use to most statecharts, and so tasks were subdivided into tasks and timed tasks, with only timed tasks offering an `update` method. This provided a small boost to efficiency, but left the timing system in place. Importantly, this allowed pre-existing, non-statechart tasks to merge seamlessly with the new statechart tasks.

4.2.3 Statechart Representation

Statecharts are defined graphically, and so incorporating statecharts into an application requires some integration effort. Given that our long term goals include reusability, portability, and verification, a pure representation is most appropriate. By this, we mean a representation of statecharts that embodies a one to one mapping between the representation and the graphical formalism. A representation

that lacks a one to one mapping adds a layer of translation, making it harder to analyze and reason about the statechart in question.

A newer formalism, Statechart XML (SCXML) defines a representation of statecharts in a human-readable XML format. Proposed as a W3C standard [4], the working draft gives the authoritative definition of the language. It provides a clean representation of a statechart, with no tangential information or symbolic representations such as those found in XML Metadata Interchange (XMI).

To add SCXML support to Mammoth, we used Commons SCXML [2]. Developed through the Apache Commons project, it 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. In Mammoth, we created the `SCXMLTask` class, which supplements Mammoth tasks with Commons SCXML, making it possible for tasks to execute SCXML statecharts.

As an alternative, the SCXML representation could be compiled into native code for the target game and executed directly. Running as native code offers a dramatic speed advantage, while allowing for further compile-time optimizations to improve execution speed. An industrial application would typically opt for the higher efficiency provided by compilation. Often, statecharts compiled as native code will simplify execution by flattening the state hierarchy and symbolically representing event labels and state names. Creating a statechart compiler is a sizeable research topic in of itself, and outside the scope of this research.

Action Resolution

Commons SCXML supports Commons Java Expression Language (JEXL) [1], also provided through Apache Commons. Any executable content in the statechart, such as an action in a transition, is resolved by parsing the action as a JEXL expression. Indentifiers in the action are resolved through the use of a *context*, which is a simple hash table that can be populated with references external to the statechart. Any method calls are evaluated through the use of reflection, and the resulting action is executed.

To provide the statechart with non-modal storage and access to the program at large, Commons SCXML provides a unique *context* as part of the execution environment for each statechart. The context is a simple hash map that maps strings onto objects. When executing actions, JEXL uses the context as the look-up table for identifier resolution, allowing action execution to pass outside of the statechart to the appropriate object. This provides us with the means to link an associated class to its statechart. We do this by loading the associated class into the context with the key 'this', providing a reference to the associated class. To call a method in the associated class, an action in the SCXML can call `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. For example, a memorizer statechart would frequently require access to the data structure in the associated class that actually stores memorized data. This is handled by storing a reference to the memory data structure in the context, giving the statechart direct access.

SCXML Introduction

SCXML is used throughout this work, and thus a brief introduction will serve to clarify usage in later chapters. As well, there were some small issues with SCXML that were resolved.

In SCXML, states are defined using `<state>` tags. Attributes allow the specification of name and final status, amongst others. Transitions use the `<transition>` tag, with attributes defining the triggering event, condition, and target. States can contain inner states, and those doing so must have an `<initial>` block that contains a condition-less transition to the default sub-state. Orthogonal components (called parallel states in SCXML) and history states are also supported.

The `action` portion of a transition, as well as the `on-entry` and `on-exit` blocks of a state, are represented as *executable content* in SCXML. Upon a transition, these locations are checked and the context executed. All executable content contains either the `expr` or `cond` attribute. Upon execution, contents of these attributes are passed to the implementation-specific expression-evaluator.

While SCXML has a raise event action, it assumes that raised events should be immediately delivered to the executing statechart. This would violate our approach to dealing with cooperating statecharts, and thus we need to add custom SCXML tags. Using `<aiEvent/>` and `<gameEvent/>`, we added the ability for our statecharts to write directly to the event queues in the role, and thus respect our structure. This was supported by the creation of GameEventAction and

AIEventAction classes that subclass `org.apache.commons.scxml.model.Action` from Commons SCXML.

Unfortunately, SCXML lacks a tag with the sole purpose of evaluating an expression, complicating action execution. The `expr` attribute only appears under a few tags, and thus we had the option of creating another custom action or making use of an existing tag. We opted to employ `<log/>`, since it includes the `expr` tag. Since no logger is being used, nothing is logged, and thus the tag has no effect other than to execute the intended action. Figure 4–17 shows a sample statechart in SCXML format.

4.2.4 Loading NPCs

To facilitate the quick introduction of newly generated roles and behaviours into Mammoth, a run-time loading system for Mammoth was developed. Two distinct sets of information needed to be supplied. There needs to be a connection between roles and NPCs in the game world, and a description of the tasks each role contains. This led to the creation of *AI-maps* and *role definitions*. Both of these are XML files, and are loaded at run-time.

In an AI-map, NPCs in the game world are assigned roles. An AI-map is specific to a Mammoth level, and named `mapname.aimap.xml`. Figure 4–18 shows an AI-map that assigns roles to two squirrel NPCs. The first NPC uses an externally defined role, while the second uses a pre-existing Mammoth role that is defined internally.

```
<?xml version="1.0" encoding="ASCII"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml"
       xmlns:scai="http://www.scytheai.com/scxml"
       version="1.0" initialstate="idle">
    <datamodel>
        <data id="timer" />
    </datamodel>
    <state id="idle" final="true">
        <transition event="start_wander" target="wandering" />
    </state>
    <state id="wandering">
        <initial>
            <transition target="moving"/>
        </initial>
        <transition event="stop_wander" target="idle">
            <scai:aiEvent name="stop_move"/>
        </transition>
        <transition event="time">
            <assign name="timer" expr="timer + _eventdata"/>
        </transition>
        <state id="moving" final="true">
            <onentry>
                <log expr="this.newRandomDestination()" />
                <assign name="timer" expr="0"/>
                <scai:aiEvent name="move" payload="this.nextPos" />
            </onentry>
            <transition cond="timer>restTime" target="resting" />
        </state>
        <state id="resting" final="true">
            <onentry>
                <assign name="timer" expr="0"/>
                <scai:aiEvent name="stop_move" />
            </onentry>
            <transition cond="timer>restTime" target="moving" />
        </state>
    </state>
</scxml>
```

Figure 4–17: SCXML representation of a Statechart that manages wandering.

```
<?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>
```

Figure 4–18: External XML file showing an AI mapping.

To handle external role definitions, a special type of role, called an *external role*, was created. External roles are classes in Mammoth that contain no tasks by default. Instead, task information is supplied by an external `role.xml` file listing the tasks that should be loaded and what their parameters should be.

In Fig. 4–19, 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.

```
<?xml version="1.0" encoding="UTF-8"?>
<role xmlns="http://mammoth.cs.mcgill.ca" name="squirrel">
  <tasks>
    <task class="Mammoth.AI.NPC.SCXML.SCXMLWanderExecutor"
                                                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>
```

Figure 4–19: Squirrel.xml, defining a squirrel NPC for use in an external role.

CHAPTER 5
Statechart-based AI Design

Having demonstrated how layered statechart-based AI works, both theoretically and through an actual implementation, the next step is to demonstrate that layered statecharts are suitable for large scale AIs, such as those found in modern AAA games. Doing so will provide substantial support to the claim that layered statechart-based AI is a viable approach to game AI.

In this chapter, we develop an extensive statechart-based AI model for a non-trivial and commercially relevant game AI, derived from the behaviour tree implementation described for the *Halo* series of computer games [32]. As well as providing a practical demonstration that statecharts have sufficient and appropriate expressiveness for such a large-scale and complex AI, this effort reveals useful and interesting design considerations. This includes the classification of several behaviour patterns, best-practices for efficiency, and an examination of the complexity of the resulting AI model.

5.1 The Halo AI

Halo stands as the first popular commercial game to employ behaviour trees for their AI logic. The approach was well received and became highly discussed in influential venues such as the Game Developer's Conference [31, 16, 17]. Many later games derive their AI from the Halo implementation, with the AI for the game *Spore* explicitly doing so [28].

Halo is an FPS game (first-person shooter) game, where the player fights groups of aliens with the help of an allied squadron. The AI controlling both teammate and enemy NPCs organizes behaviours under high-level functions: search, combat, flight, self-preservation, and idle. Each of these contains subtasks; combat, for instance, decides between grenade use, charging, fighting, searching, and guarding. The tree for Halo 2 has a maximum depth of four, with the bottom layer consisting of leaf-nodes that execute concrete behaviours and trigger corresponding animations. Nodes can be cross-linked acyclically allowing a single behaviour to appear under multiple nodes [32].

Behaviour trees (BTs) recast HFSMs into a strictly hierarchical decision tree. While they clearly delineate how the system selects behaviour, the strict hierarchy impairs reactivity and lacks modal states that would encapsulate different behaviour groupings. Recent advances, such as event-driven and data-driven BTs improve efficiency [7], but sidestep reactivity issues through parallel nodes. This means that, despite their success, there is room for improvement in regards to reactivity.

5.2 Designing a New Halo AI

The goals in developing a statechart-based version of the Halo AI were as follows: to capture the basic behaviour of the reference AI without sacrificing key functionality, while showing how reuse practices and modularity lead to a well-constructed AI. The Halo AI was chosen due to its visibility as an example of good AI design, and its success as a commercial title. By developing a similar AI, we

can credibly conclude that layered statecharts are capable of handling industrial scale AIs.

A key element of our design approach was the use of subsumption. As low level events are generated in the input layers, they are received directly by output layers. This allows high level statecharts, such as the strategizer, to ignore these events, and thus greatly reduces complexity of these higher level statecharts. Typically, subsumption creates a coordination problem when low level reactions conflict with higher level behaviours. This is usually addressed through the addition of coordinators that resolve differences between low-level and high goals.

We instead solve this problem this by construction, never allowing lower levels to enact behaviours without prior permission. We call this approach *limited subsumption*. Here, low level sensors and actuators communicate information about the game state to the output layers to preconfigure behaviours, as is normal in subsumption, but these actions can never trigger a behaviour to start or stop.

An example illustrating limited subsumption is shown in Fig. 5–1, with subsumptive communications colored red. The figure gives a scenario relating to melee combat. Here, a sensor creates an event when the NPC equips a melee weapon. Rather than go through the brain, this event is received directly at a decider, which causes a transition into the appropriate start state for melee combat, but does not cause the AI to engage in actual melee combat. This preconfigures the statechart to execute the appropriate type of combat. When another event signals that an enemy is in melee range, the combat behavior is further configured, but still does not begin combat. Once the *Brain* decides to

enter combat, it will send out a start combat event, and only then will the relevant decider start executing the preconfigured combat approach. Using this approach, tactical details are subsumed from higher level statecharts, lower levels never take spontaneous action, and the need for explicit coordination is obviated. This pattern of configuration followed by later execution defines limited subsumption.

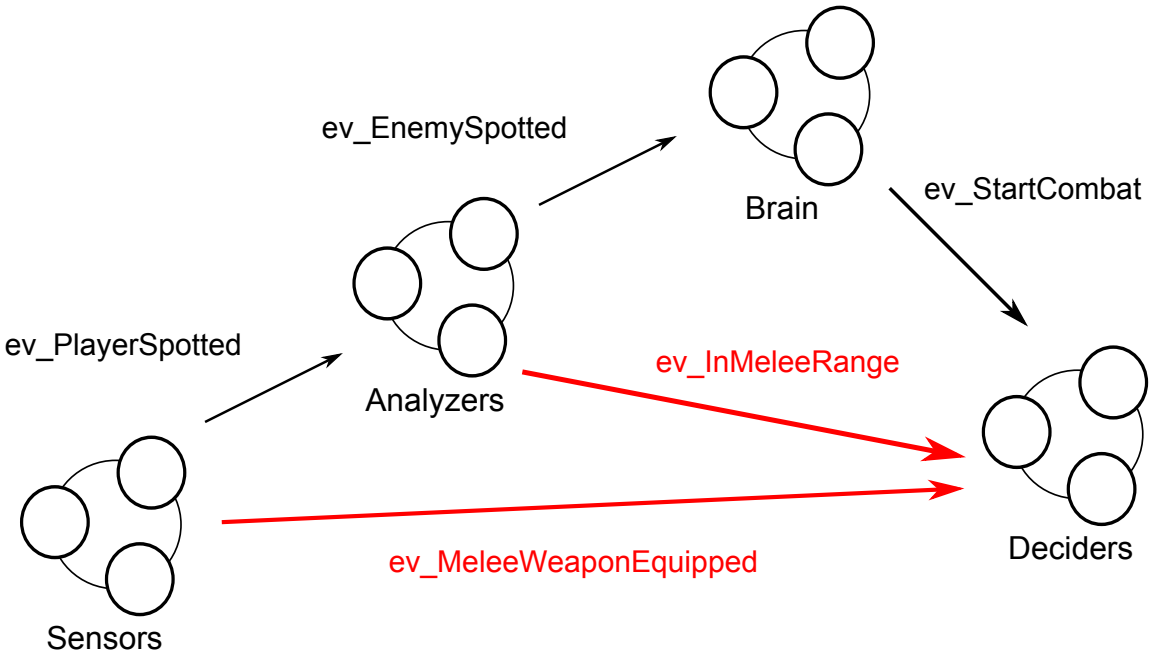


Figure 5–1: Limited Subsumption Approach

The layered statechart version of the Halo AI is included in its entirety as Appendix A. Consulting the appendix is not necessary to understand the ideas presented in this thesis, since noteworthy statecharts are reproduced within the main text. However, curious readers may be interested in a deeper examination of the AI, and so it is provided in full.

5.2.1 Input Layers

Halo presents a complex game state with many complicating factors. Other NPCs can be friends or enemies, vehicles can be manned or empty, and obstacles are dynamic. In total, 25 individual statecharts were needed to capture and comprehend the Halo game state. Table 5–1 lists each statechart created. Descriptions will be given as necessary in the following subsections for each layer.

Table 5–1: Input statecharts in the Halo AI

Sensors	Analyzers	Memorizers
Attack	Enemy	Character
Character	EnemyProximity	Command
Commands	GrenadeProximity	Obstacle
Grenade	LowMorale	Vehicle
Health	Special Event	
Item	Squad	
Obstacles	Threat	
Position	Vehicle	
Shield	VehicleProximity	
Vehicle	ThreatCompiler	
Weapon		

Sensors

Each sensor tracks a specific portion of the game state, generating appropriate events. For instance, the Self Sensor tracks the NPCs current health, Shield tracks the NPCs, and so. Others sensors notice changes in the game state, such as the Grenade Sensor which tracks nearby live grenades, Vehicle which tracks vehicles, and Commands which receives player commands given to the NPC.

In the squirrel AI, sensors made extensive use of polling to obtain information about the game state. This was done by using eventless transitions with a guard

that queried some value of the game state. At each execution loop, the guard would be evaluated, despite that fact that values would only change in a small percentage of overall frames. To improve efficiency, statechart polling was eliminated in the Halo AI. Instead, the associated class for each sensor obtained game state information by registering as a listener to the game. Upon receiving a listener callback, the class would generate a private statechart event for the sensor such as *ev_ healthChanged*. Transitions would use this new event name and include guards based on the new value in the game state.

Analyzers and Memorizers

Each analyzer was responsible for learning about a portion of the game state, listening to both sensors and other analyzers for input. For instance, the Enemy analyzer would determine if spotted players were friends or foes. This in turn would trigger the Enemy Proximity Analyzer to determine if the enemy was in melee range. Together, the output of the Threat Analyzer and the three proximity analyzers would feed into the Threat Compiler Analyzer to determine the overall threat to the NPC. Memorizers are straightforward, memorizing and making available the data indicated by their name.

In practice, the line between analyzers and memorizers appears fuzzy. For many analyzers, it was necessary to utilize previously sensed information about the game state when considering new input. Instead of storing this at a memorizer and retrieving it with synchronous calls, some analyzers simply stored game state information in their associated class. This provides a more streamlined system.

Memorizers are distinguished not by their ability to store data, but by their ability to *share* data. Each memorizer provides a synchronous method call to allow other modules to read their data, something that analyzers do not do. In other words, memorizers provide information to the entire AI, while analyzers only store information needed for their own internal operations.

5.2.2 Strategizer

At the highest level of abstraction, our *strategizer* uses states to store current goals: changing states implies that a new goal has been selected. This is communicated through the creation of an event using the on-entry block of the state. When a state is exited, an on-exit action creates a stop event notifying downstream statecharts that they should cease current behaviours. By conflating strategizer states with the current goal, the result is a statechart that is highly intuitive. As usual, we use only one strategizer.

This approach proved very practical. Since lower level statecharts are aware of the current game state through subsumption, it is sufficient for the *Strategizer* to simply give commands to lower levels, without having to deal with unnecessary details. The statechart itself, shown in Fig. 5–2, looks nearly identical to the high-level approximation of the combat cycle for the AI given in [32] and reproduced here as Fig. 5–3. This means that our statechart approach yields a high level strategy that is visually explicit, free of complication, and nicely conforms to the original designer's intuitive understanding of what it ought to look like.

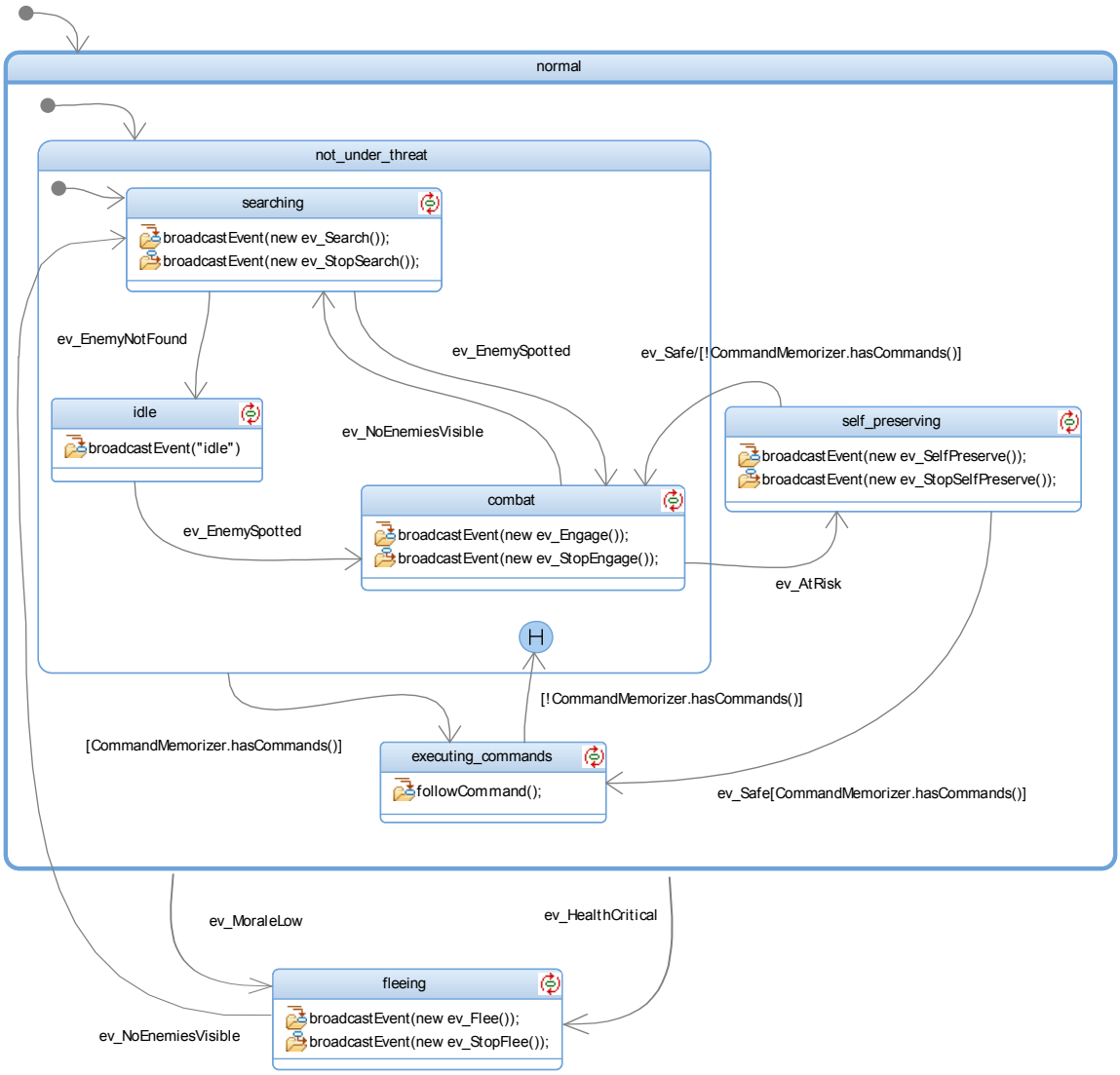


Figure 5–2: The high-level *Strategizer*.

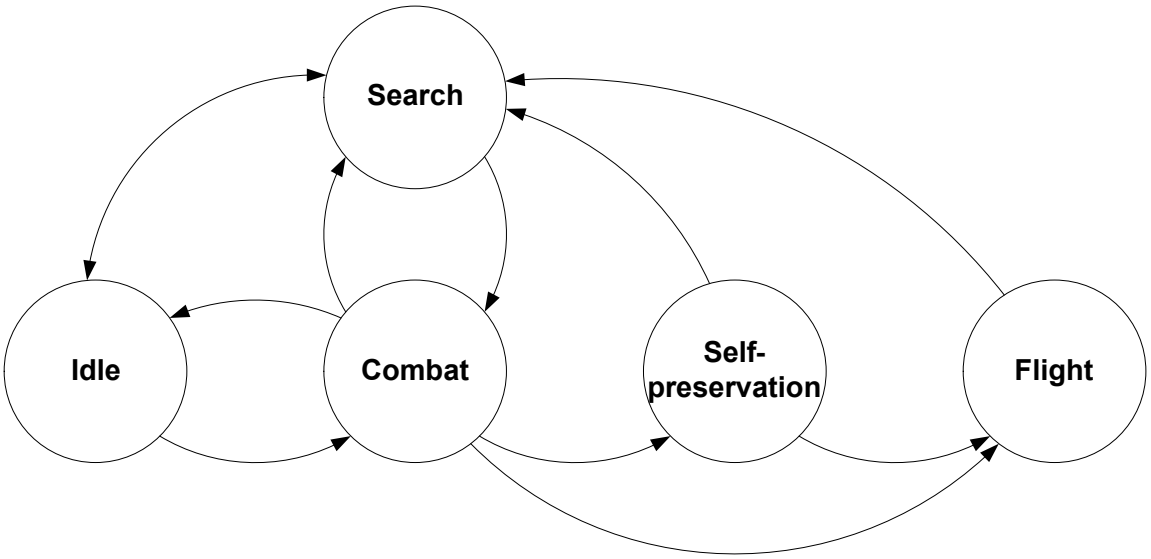


Figure 5–3: The Combat Cycle from the Halo AI.

5.2.3 Output Layers

In the output layers the effect of our limited subsumption becomes clear. While the Strategizer could store enough information to decide on a specific tactical strategy, this would unnecessarily complicate its structure. Instead, tactical deciders directly receive relevant sensor and analyzer data so that their decisions are prepared in advance. This has the additional effect of making tactical deciders more modular, in that their logic is self-contained. The complete list of output statecharts is given in Table 5–2.

Table 5–2: Output Statecharts in the Halo AI

Deciders	Executors	Coordinators	Actuators
Combat	Clear Area	Movement	Grenade
Flee	Flee All		Item
Idle	Flee Nearby		Melee Weapon
Search	Item		Ranged Weapon
Self Preservation	Melee Combat		Run
	Ranged Combat		Sound Actuator
	Search		Vehicle
	Take Cover		
	Use Item		
	Vehicle Combat		
	Wander		

Deciders

There are five deciders, one for each of the high-level goals. When the strategizer chooses to search, for instance, it sends an `ev_Search` event, which starts the *SearchDecider*. When the strategizer makes a new decision, it deactivates the *SearchDecider* with a `ev_StopSearch` event, and sends a new event to enact the new goal.

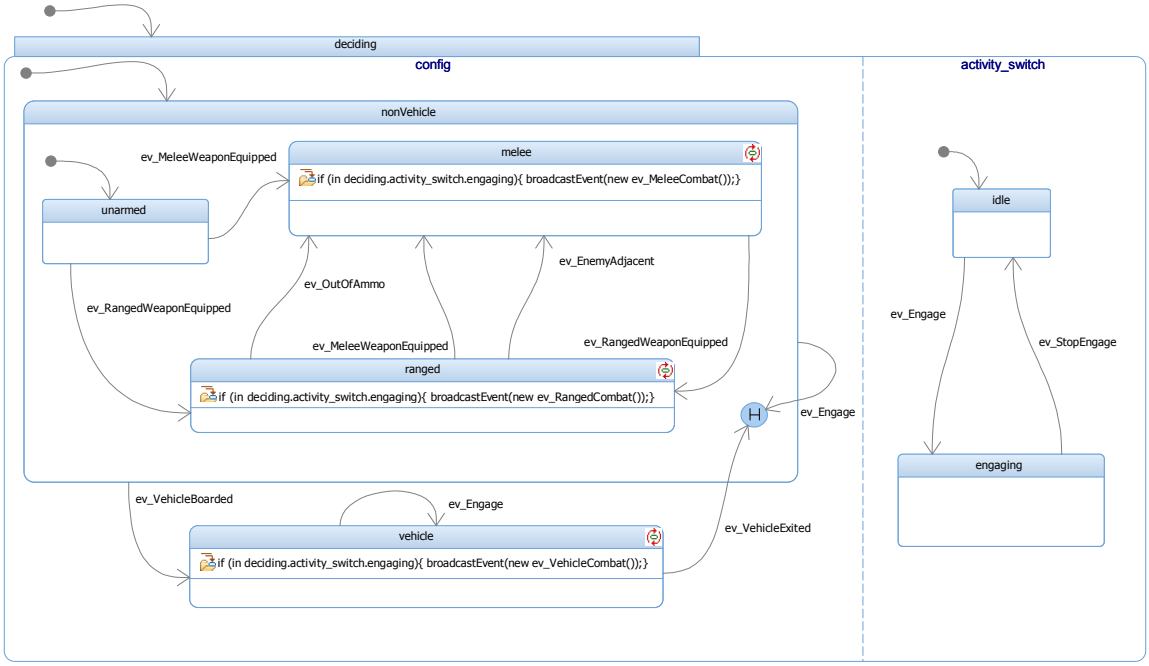


Figure 5–4: The *CombatDecider*.

The *CombatDecider* is presented in Fig. 5–4, since it serves as a good example of subsumption and usage of orthogonal regions. Events relating to equipped weapons and ammo level come directly from the *WeaponSensor*, while enemy location comes from the *EnemyAnalyzer*. Upon receiving an `ev_Engage` event from the Strategizer, the orthogonal region `activity_switch` enters the `engaging` state. This permits activity and immediately triggers an on-entry event in the main region. As new relevant inputs are received, the *CombatDecider* is free to revisit its own tactical decisions and execute new actions, so long as the activity switch remains on. This approach is only possible due to the orthogonal region—without it, the number of states would double as each state would have to have an active and inactive version.

Executors

Each executor represents a mapping of goal to actions. For instance, if the *CombatDecider* decides to engage in melee combat, it will call the *MeleeCombatExecutor* using a `ev_StartMeleeCombat` event. The executor, based upon information passed forward through limited subsumption, will in turn choose actions and send events that start the actuators. The purpose of the executors follows from their names with the exception of the flee executors. The *FleeAllExecutor* flees from all threats by using the *FleeNearbyExecutor* to flee from enemies in melee range, and the *ClearAreaExecutor* to flee from area threats such as live grenades.

Coordinators

Coordinators solve potential conflicts between actions and correct for changing conditions. Since subsumption does not result in action without prior permission from the *StrategicDecider*, there are no inter-layer conflicts to resolve. Instead, the only coordinator automatically transforms move actions into vehicle movements if the NPC is driving, or run movements otherwise, simplifying move events at higher levels.

Actuators

At the lowest level are actuators. Like executors, their purpose is strongly implied from their names. The *RunActuator* handles movement by foot, while the *VehicleActuator* handles all vehicle actions including driving.

5.3 Statechart Patterns

While designing the new AI, several statecharts possessed identical or isomorphic structures, differing only by event and state names. This observation allowed

us to successfully isolate 5 different *statechart patterns*. Like design patterns in software design, statechart patterns encode in their structure a solution to a recurring design problem, and thus are a valuable contribution to those seeking to design their own statechart-based AI.

Sensors

In designing sensors, we found two recurring statechart patterns. The first we called a *discretizing sensor*, which maps a continuous value to discrete events using threshold values. The number of states is equal to the number of discrete levels needed. Transitions between the states have guards constructed from desired threshold values. The *HealthSensor* given in Fig. 5–5 is an instance of a discretizing sensor with 3 states.

The second sensor pattern was a simple mapping from in-game events to AI events, creating a bridge between the game and the AI. This was typically a state-independent transformation, yielding a trivial statechart with a single state and no transitions. We call these *event-mapping sensors*, and use these as often as possible due to their overall efficiency. In the case where event-mapping is state-dependent, such as having on/off states, an event-mapping sensor can be expanded to have a second state that does not generate events, with appropriate transitions between.

Analyzers

At the level of analyzers, one pattern emerged, which we named the *Binary Analyzer*. Their primary function is to keep track of a countable aspect of the game state, and notify the AI when the count increases from zero, or decreases to zero. An instance is shown in Fig. 5–6, where the analyzer memorizes each

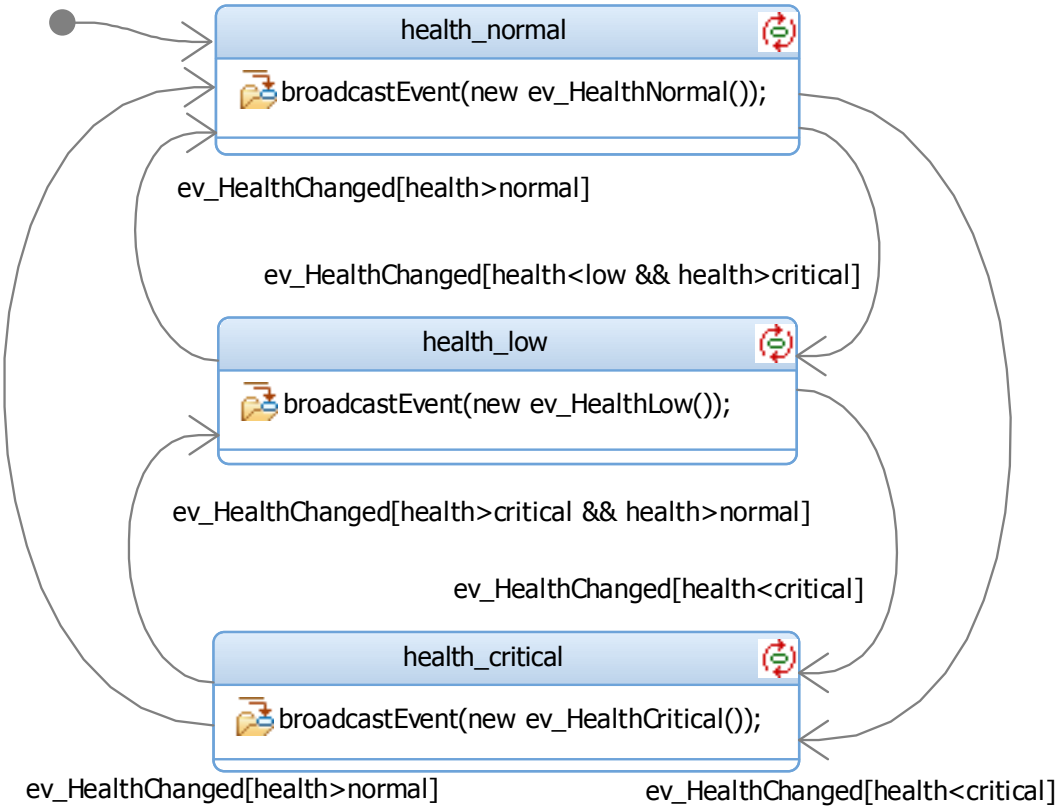


Figure 5–5: The *HealthSensor*.

individual grenade spotted. It creates an event for each nearby grenade, but does not give the all clear until all grenades are out of range. After each grenade explodes, or the NPC moves, the binary analyzer enters a state with two outbound transitions having mutually exclusive guards. This state determines if the grenade count has returned to zero, or if it remains greater than zero. This ensures that the analyzer only checks the count when the quantity of tracked grenades may have changed.

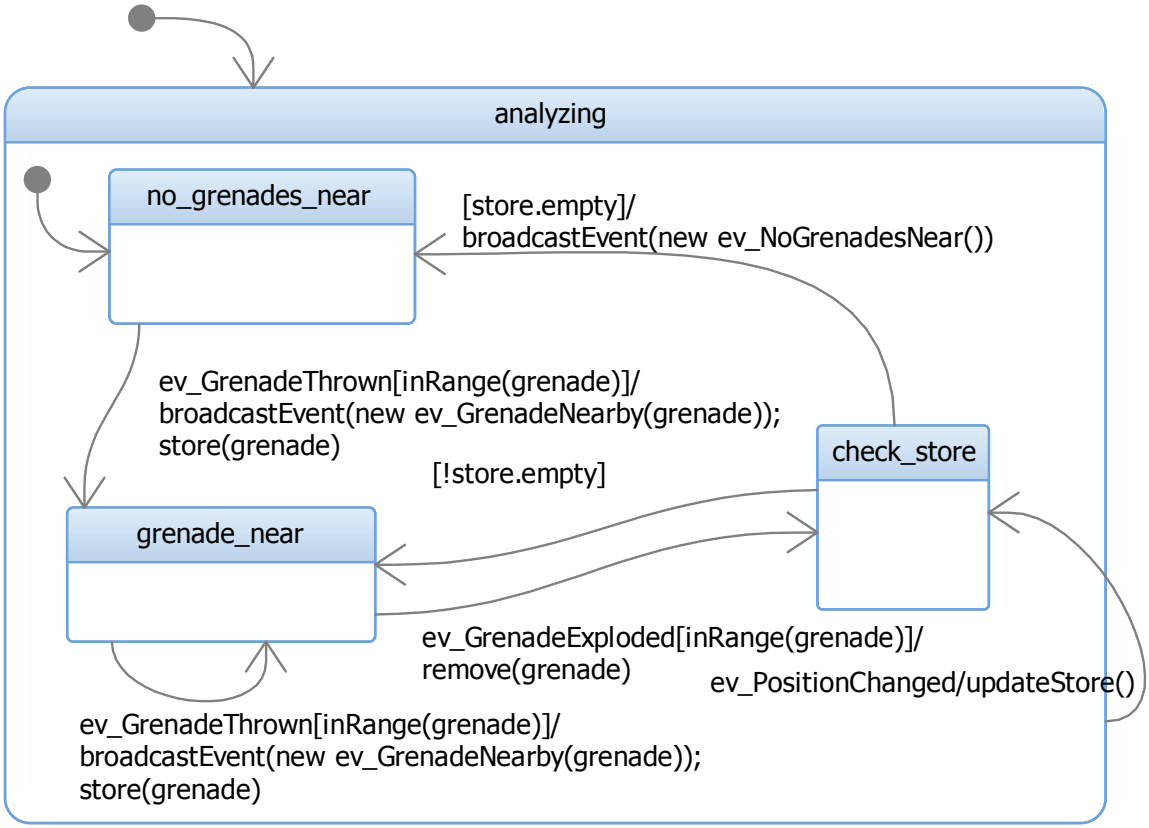


Figure 5–6: The *GrenadeProximityAnalyzer*.

Deciders

A fourth pattern was isolated at the decider level, the *Priority Decider*. These are used in the situation where a tactic A is the default activity, but always prioritizes tactic B when the triggering event is received. In Fig. 5–7, we see how *FleeAll* is the default behaviour, but is superseded by *FleeNearby* when an enemy is detected in melee range. Upon abatement of the threat, the decider returns to default. Note that this pattern can easily extend to include yet higher priority behaviours.

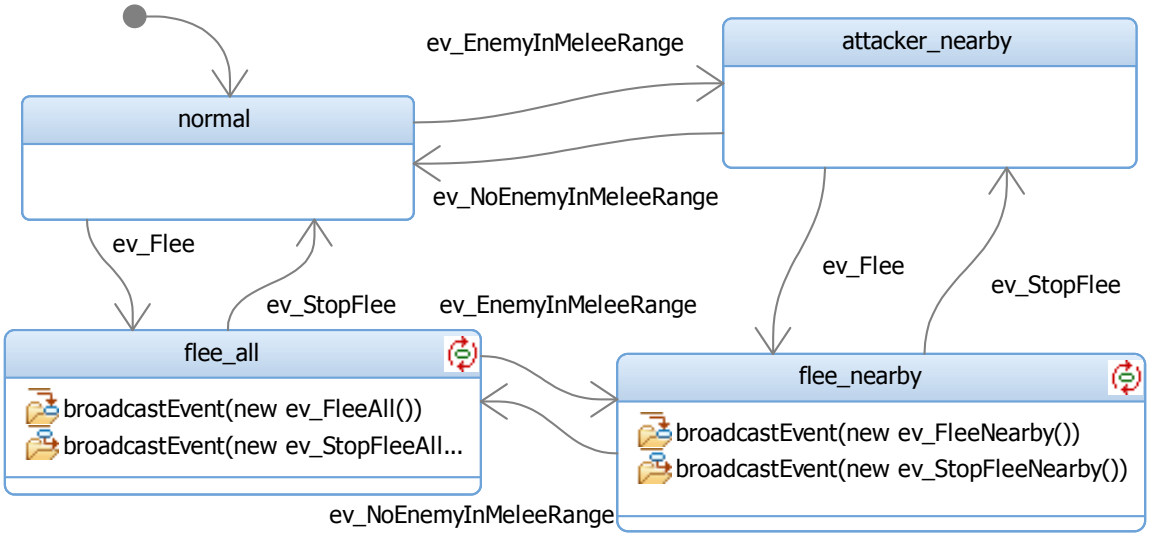


Figure 5–7: The *FleeDecider*.

Actuators

While most actuators trivially receive events and execute actions, one new pattern emerged. Similar to exceptions in code, actions can fail for a variety of reasons, e.g., trying to pick up an item that has just been picked up by another player. When these failures are relevant to the AI, it is useful to have the actuator itself track the results of an event and produce appropriate feedback. We call actuators of this type *Feedback Actuators*. Upon creating a move action, the feedback *MoveActuator* show in Fig. 5–8 can receive a callback event after executing its `pathfind(target)` call and react accordingly. For instance, if a move fails because a new obstacle has appeared, a reasonable course of action is to retry and allow the pathfinder to calculate a new path around the new roadblock; usage of a feedback actuator allows this. On the other hand, if the failure is due to no path existing, higher levels may wish to change behaviours. This is signalled by

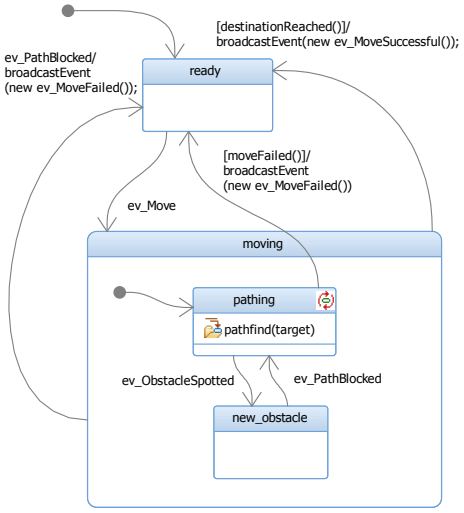


Figure 5–8: The feedback *MoveActuator*.

having the sensing portion of the feedback actuator create an *ev_MoveFailed* event, notifying higher levels that a new destination should be determined, or a new behaviour chosen.

5.3.1 Applicability

The statechart patterns proved to be quite valuable. In our Halo AI, the quantizing sensor appeared twice, binary analyzers were employed three times, feedback actuators were used twice, event-mapping sensors were needed four times, and priority deciders were used twice. This accounts for 13 statecharts, meaning that 28% of the statecharts were pattern instances, varying only in state and event names.

One could easily adopt these patterns into a variety of FSM approaches, or generate them automatically through tool support. In this AI for instance, this would have reduced by one quarter the time spent on generation of novel

statecharts. While there is a certain amount of overhead in instancing patterns, in my experience it is much lower than the time needed for novel development.

5.4 Key Features

In the various publications and presentations regarding the Halo AI (such as [32] and [16]), several key features were highlighted. These included a technique enabling efficient event reaction and a method to customize behaviours for individual NPCs. In the conversion to the layered statechart formalism, it was important to ensure that none of this functionality was lost.

5.4.1 Stimulus Behaviours

The Halo AI attempted to enable reactive behaviours by the creation of *stimulus behaviours*. The framework used a stimulus system that received events from the game at-large, and reacted by inserting special stimulus nodes into the behaviour tree at run-time. Upon the next execution through the tree, the new stimulus behaviour would run as expected. The insertion point of a node was carefully chosen, so that it respected the decision making process of the tree without overriding higher level behaviours that may potentially supersede the inserted behaviour. While this provided the ability to react to rare events without repeatedly checking for a condition, it came at the price of making the behaviour tree less understandable by obscuring behaviours.

Fitting an event-based reaction into a statechart-based approach is trivial. This can be done by adding a statechart that reacts to the event in question and produces an output that triggers the appropriate higher level statechart. The example of a stimulus behaviour in Halo was for the AI to flee if their leader was

killed. Our statechart approach accomplishes this by adding a new analyzer to check if the player involved in an `ev_PlayerKilled` event was actually their leader. If so, it reacts by sending a special `ev_ LowMorale` event to trigger fleeing behaviour at the *StrategicDecider*. The AI behaviour is clear at all times since dynamic behaviour modification is not required. As well, the new behaviour is modularized, with all operations related to leader tracking stored in one location, making it straightforward to comprehend.

5.4.2 Memory Usage

In game programming, especially in the case of console games, system memory is a limited resource. One of the major achievements of the Halo AI was the strict ceiling on memory usage. Here we compare memory usage of our new AI with that of the original behaviour tree AI.

For agent-based NPCs it is typical for each agent to have their own instance of the AI, customized to their state in the game-context. By this we mean that a game using behaviour trees would have several different behaviour trees, and each NPC would instantiate the appropriate tree and use that to drive decision making. This becomes impractical in a memory-limited environment such as a gaming console.

For Halo, memory constraints were quite significant and thus a new approach was needed. The behaviour tree was implemented as a single static structure, shared by all NPC instances. Stimulus behaviours were added dynamically for a short time to the executing method for each NPC as they arose. The fact that enemies and allies followed the same general behaviour made this solution feasible.

As an NPC executes the tree, it needs only the behaviours it is currently exploring, which is the simple hierarchy tracing from the root to the current node. For 100 NPCs, the memory usage in this arrangement was approximately 25KB, a very small profile.

However, many decisions are based on characteristics of the NPC, and thus NPCs require memory aside from the actual tree. In addition to the 25KB for the tree, there is also stored information regarding obstacles, enemy sightings, NPC attributes, and so on. This was stored in pre-determined locations in NPC instances so that the behaviour tree could consult NPC memory with universally applicable calls.

Approximating this approach in the layered statechart formalism follows the same basic formula. Instantiating every statechart for each NPC would use an unnecessarily large amount of memory and so we seek to create a single static structure shared by all NPCs. While a behaviour tree only explores one branch at a time per NPC, ensuring the size of that exploration hierarchy is bounded by the tree depth, a statechart always has a current state for each NPC, and thus the current state must be stored for every statechart, possibly more than once for each statechart if there are orthogonal regions.

Statechart execution requires additional information to broadcast events. They must be supplied with a reference to the NPC from which state information can be read. Additionally, information used by statecharts, such as counters, memory storage, and so on, must also be stored in well-defined locations in each

NPC, allowing the shared static structure to correctly read and write data from the individual NPCs.

Through the use of matching static structures, it is possible to store state information efficiently. The list of statecharts is managed by a statechart execution manager, responsible for storing the singular instance of each statechart. Each NPC maintains an array of bytes (assuming each statechart has an upper limit of 256 states), where each byte corresponds to a state ID within each statechart. By matching the array indicies with the statechart ordering in the manager, no extra reference needs to be stored to link a state ID with its statechart. We end up storing 50B of state information for 50 statecharts in each NPC, plus an extra byte for each orthogonal region. This is in addition to the cost of loading the statecharts themselves into memory, and the AI and game event queues which are shared by all statecharts. While there is no upper bound on the number of statecharts being used simultaneously, in practice, each event is only received by a few statecharts at most.

Stringent memory usage arose because of memory limitations on the Xbox360 console, which boasts a mere 512MB of RAM. On a platform with additional memory, such as a PC, more CPU-efficient approaches can be taken, such as creating a hash table that links events to statecharts, allowing broadcasts to be replaced with narrowcasts to just the subset of statecharts that have transitions on the event being processed.

Overall, memory usage is reasonable when compared to the Halo AI. Determining a final number for memory usage would require us to know the average

size of a compiled statechart in memory. For now, though, this knowledge is not available. However, if we generously assume that an efficiently compiled statechart takes 1KB of memory, there could be 60-70KB memory used for 100 NPCs. While this exceeds the 25KB allotment achieved with behaviour trees, it is still quite reasonable for most platforms.

5.4.3 Behaviour Masks

Halo employs a shared static structure for the behaviour tree which effectively limits memory usage, but this comes with a downside: using a single data structure prevents each character from having their own customized AI. While this can be tempered by clever design, for example by having characters that wield ranged weapons travel down a different branch of the behaviour tree than those with melee weapons, the problem remains that all AI characters use the same behaviour tree.

Halo 2 addressed this through *styles*. They provide customization for characters by providing a list of disallowed behaviours, effectively pruning branches from the behaviour tree. This evolved into *behaviour masks* in Halo 3, which gave designers the choice between 3 sets of disallowed behaviours, resulting in NPCs that were normal, aggressive, or timid. An aggressive behaviour mask, for example, disallows the branches of the behaviour tree concerned with fleeing and taking cover.

In our layered statechart-based approach, the situation is not as simple as trimming branches, since decision making and reaction to events is distributed across modules. Regardless, if a module does not receive a triggering event, no

behaviour will occur. Two approaches exist: filter events, or entirely ignore the generating statechart. Implementation-wise, the statechart approach can also employ a shared static structure, with the current state of each statechart with respect to an NPC stored in the NPC. By setting this to 0, it can be communicated to the statechart executor that an NPC is not using the referenced statechart, causing that statechart to be skipped when processing events.

Ignoring entire statecharts has the disadvantage of being relatively coarse-grained—a statechart may produce more than one event, and so ignoring the statechart may affect multiple downstream behaviours. Individual event filtering gives more fine-grained control, but suffers from larger storage requirements as well as additional computational costs in verifying individual events. Statechart behaviours that are fully filtered may also be a source of redundant computation. In our design we are able to exclusively make use of statechart removal to customize individual behaviours, relying on the use of small and relatively modular statecharts to achieve sufficient granularity.

To mimic the aggressive behaviour mask, we disabled the *ThreatCompiler-Analyzer* module and the *LowMoraleAnalyzer*. This removed the generation of the `ev_AtRisk` events, which would trigger the strategizer to choose the flee goal. Thus, the NPC would never choose to flee, which is the behavioural characteristic expressed by the aggressive mask.

5.5 Analysis

In total, there were 50 statecharts, containing an average of 4.00 states and 6.08 transitions. All of these are listed in full in Appendix A. Only 5 statecharts

had 10 or more states (the largest was the *ThreatAnalyzer* with 13), but all of these had orthogonal regions in the state at the top of the hierarchy. These regions could be separated into independent statecharts, and if this was done, the averages drop to 3.28 states with 5.18 transitions apiece. While the overall number of statecharts is high for such a complex AI, each statechart is small enough to be easily understood. The top level *StrategicDecider* is only 8 states with 12 transitions, a comprehensible size.

The value of modified subsumption in managing complexity was also clear. The size of the resulting statecharts was small enough to be easily comprehended, due in large part to the simplification of high level statecharts. The use of orthogonal states also proved to be a valuable tool in preventing combinatorial explosion of states, noteworthy because this technique is not a part of standard hierarchical finite state machines, and allows for the specialized activity switch that we employed.

5.5.1 Implementation

The statechart model we created is based on information made available through various presentations on the Halo AI; we had no special access to the source code or the actual production AI. This approach was chosen to ensure our work could be used outside of any proprietary context, but has the disadvantage that the public description is incomplete, and so our efforts represents our best approximation of what we interpret the full AI to be. Moreover, our modelling work did not involve the creating or coding of various algorithms invoked in the AI, such as target selection, pathfinding, or determining cover, limiting our ability

to validate that our statechart AI is a complete and faithful reproduction of the original behaviours.

Validation took place at two levels of abstraction. First, the model was examined to determine the effectiveness of the approach at the design level. Secondly, the AI was partially implemented to verify correctness of the logic in the statecharts, as well as to observe the model functioning in a game environment.

The AI was implemented in Mammoth [36], a highly extensible research framework for MMOs. While Mammoth lacks many features now common to AIs in FPS games, such as cover maps and navigation meshes, this implementation was sufficient to test core functionality of the AI, demonstrating that the various statecharts operated correctly in both their individual and collective roles. Testing was performed by simulating random external event generations and observing the event chain generated in response, as well as state changes within the AI.

Our implementation also allowed for practical verification of potential performance concerns. For example, since the design is modular, event profligacy is a concern. Event generation was thus examined by looking at the number of events potentially generated in response to various inputs. The maximum number of events was 14, assuming that every guard evaluated to true and that every statechart involved was in the appropriate state to react to an event thereby continuing the chain reaction. While this worst case is high, the number of events generated in practice was quite low. After several short executions of the AI, the AI generated a mean of only 0.3 events per execution pass, while the median

number of events in a single pass was zero. Aside from the occasional burst of 5-10 events, event generation was quite limited and did not cause a significant overhead.

Ultimately, testing in Mammoth was able to establish correctness of basic behaviours, such as wandering and searching, but was insufficient to effectively validate the AI. In an industry setting, AIs are extensively play-tested by developers, in-house testers, automated test kits (such as random input generators), and even beta testers. It is simply not feasible to validate the correctness of such a large AI without considerable investment of resources.

This experience helps justify the need for effective verification of statechart-based AI. In Chap. 8, we will develop a more complete approach to validating such an AI, and will apply those techniques to the layered statechart-based version of the Halo AI.

CHAPTER 6
Modular Reuse

At the Game Developer's Conference in 2011, Kevin Dill argued that the lack of behavioural modularity was stymying the development of high quality AI [62]. The lack of a consistent formalism used in representing AI logic, and in particular the use of non-modular custom approaches, has prevented AI reuse from becoming commonplace. Layered statechart-based AI, however, is inherently modular and thus forms an excellent basis for AI reuse through modularized behaviours. In this chapter, we detail an approach to AI reuse using layered statecharts, and offer a demonstration of reuse based on the squirrel AI developed in Chapter 4.

By performing a thorough examination of module communication in §6.1, we derive a module interface that compiles all information relevant to reuse. Using this as a basis, we present a complete approach to reuse that handles both AI logic and associated code. Additionally, this leads to the creation of *functional groups*, which group together connected modules as a single reusable component. The end result gives AI developers the ability to modularize and reuse known good AI behaviours.

We illustrate our approach by constructing a new AI for Mammoth using the squirrel as a basis. The target AI will be an NPC responsible for collecting garbage and placing it in receptacles. These NPCs fit different game AI contexts, and would typically be approached as unique development tasks. However, some of the

core behaviours are similar in nature, and by expressing and reusing AI behaviours at a suitable level of abstraction, our approach is able to capture many of these commonalities: over half of the *AI modules* in the trash collector are reused from the squirrel.

6.1 The AI Module

Layered statechart-based AI is built around combining individual statecharts to produce the overall behaviour. This implies that the fundamental module is the statechart itself. We define an *AI Module* as a statechart along with its associated class, and will use this as our fundamental module of reuse. This is similar to game graphics, wherein models built from texture mapped triangles form the basic reusable component. Models can easily be reused in new contexts, and by clearly defining the reusable component, we wish for AI modules to be just as reusable when building new game AIs.

The pairing of statechart and associated class is a highly appropriate choice for two reasons. First, statecharts act independently, and only cooperate due to higher level coordination. This means that the statechart itself has a contained execution environment, allowing for execution in any context. Secondly, each statechart has a link to an individual associated class, providing an elegant approach to inclusion of source code. This pairing is advantageous as we can now reason at a high level about AI behaviour without neglecting the implementing code.

6.1.1 AI Module Interaction

As modules are reused, they are removed from one context and placed into another. For reuse to be successful, the new context must be able to interact with the reused AI module. To enable reuse, there needs to be a clear description of how modules interact. Having this will allow us to build up an interaction profile for each module, supplying the information needed to correctly insert an AI module into a new context.

Event-based Interaction

The statechart portion of an AI module communicates using event-based message passing. Using the cooperative approach described in Chap. 3, statecharts generate events that are broadcast to all other statecharts in the system.

Events generated by a statechart and consumed by other statecharts are classified as *output* events from the perspective of the generating statechart. These same events become *input* events when viewed by the consuming statechart. If an event is neither generated nor consumed by a statechart, then that event is irrelevant to that statechart and is ignored.

In some rare cases, a statechart may both generate and consume an event, or the associated class may generate an event (such as the Health Sensor in the Halo AI) that is intended only for the statechart in the module. These self-communications express logic internal to a single statechart, and so these events are classified as *private*.

Synchronous Communications

As discussed in §3.2, associated classes in AI modules communicate through the use of synchronous method calls. When reusing a module that makes synchronous method calls to other classes, any target modules must also be reused, or the associated class edited to update the call with a valid target. Otherwise, the associated class will fail to compile.

When compared with event-based communication, synchronous communication leads to tightly-coupled modules. This reduces the ease of reuse, since the module cannot be reused on its own without modification. Because of this, we recommend limiting the use of synchronous methods calls where possible.

An alternative to synchronous calls is to employ events with payloads. Payloads can carry pertinent information to satisfy upstream data requirements. This allows for a more loosely-coupled system overall, better suited for reuse. An event with a payload can be given a *signature* stating the type of the payload, much like a function or method signature, making it clear the information that is contained in that payload.

As an example, a common task in the layered statechart module is for analyzers to refine information collected by the sensors. The sensor could store the data and make it available by synchronous method call, however, reuse of the analyzer would be complicated if the sensor is not also reused. Instead, the event notifying the analyzer of the data collection could include as payload a reference to the relevant game object. Now when the analyzer is reused, no modification of the associated class would be required, and reuse is simplified.

Miscellaneous AI Module Properties

Aside from event-based and synchronous communications, there are three properties of an AI module that impact reuse. Each of these properties is related to the associated class. The first such property is the the programming language. While statecharts are represented in language-independent SCXML, the associated class is a code artifact. Since it is programmed in a specific language (e.g., Java or C++), the AI module can only be reused in games programmed in the same implementation language.

The next property important to reuse is the set of links between the associated class and the game-at-large. Any imports in Java or includes in C++ create dependencies in the code. While references to core libraries (such as `java.*`, or the C++ STL) are always resolvable, references to custom libraries or to game-specific classes may cause issues. Successful reuse demands that such calls be updated to appropriate targets in the new context, or referenced classes must also be reused such that the associated class can compile correctly. If an AI module has no game imports, then it is *game-agnostic* and can be safely reused in a different game without issue.

The final property relates to non-modal properties of the AI module. Such properties, stored as variables in the associated class, allow for customization as the module is reused. An example comes in the form of the `KeyItemMemorizer` module, found in the definition of the squirrel AI. The associated class has a type parameter which it uses to determine if a spotted item is in fact a key item (an acorn, in the case of the squirrel). A game implementing several important items

(such as flowers, shirts, and boxes), could have a `KeyItemMemorizer` for each item relevant to the AI. When a `KeyItemMemorizer` receives an `item_spotted` event from a sensor, the payload would be inspected and compared against the key item parameter, and then memorized if it is a matching type. By exposing parameters in the interface, it makes it possible to modify them at time of reuse, and thus customize the module to the new context.

6.1.2 The AI Module Interface

The motivation behind an AI module interface is to collect information relevant to reuse in a single location. This is similar to the API (Application Programming Interface) for an application or library. Proper use of a library requires that the API be followed; proper reuse of an AI module demands that the module interface be followed.

The interface for an AI module collects and presents information on events, synchronous calls, parameters, and any imports. Pertinent information can optionally be added to the interface, such as a name for the module, an outline of the behaviour, and the intended layer. This means that all the information relating to the reuse of a module is found in a single location, making the usage of module interfaces a valuable tool when performing reuse. A straightforward depiction of a generic interface is given in Fig. 6–1.

By filling in descriptions and additional information, the interface effectively communicates the functionality of the module. Figure 6–2 presents the AI module interface for the Key Item Memorizer presented in the Squirrel AI, and clearly describes how the module works, and how it communicates.

Statechart Name	
Game: Mammoth *Description*: - Description of Statechart and its behaviour *Parameters*: - Type :: Description	*Language:* Java
Events	Calls
Input: - EventIn (PayloadType) - Event2In *Output*: - EventOut - Event2Out(PayloadType) *Private*: - InternalEvent	*Game Imports*: - import Game.Element - import Library.Class *Available Synch. Calls*: - method(signature) *External Synch. Calls*: - Module.method(signature)

Figure 6–1: A generic AI module interface.

KeyItemMemorizer	
Game: Mammoth *Description*: Filters item spotted events to memorize key items that have been spotted. *Parameters*: - ItemObject keyItem :: The item type to be memorized as the key item.	*Language*: Java
Events	Calls
Input: *- i_see_item (ItemObject)* *- i_dont_see_item (ItemObject)* *Output:* *- key_item_visible* *- no_key_item_visible*	*Game Imports*: - Mammoth.AI.NPC.Role, ".PhysicsEngine.PhysicsEngine, ".WorldManager.ItemObject, ".WorldManager.WorldManager *Available Synch. Calls*: - Vector<ItemObject> getKeyItemList()

Figure 6–2: The module interface for the Key Item Memorizer.

6.2 Component Integration

Integrating modules correctly and easily is fundamental to effective AI reuse. Each reused module must be *connected* to the other modules in the new AI, either through event-based or synchronous communication. Here we describe how modules can be reused and correctly connected in a new context.

6.2.1 Event Renaming

For event-based connections, modules communicate by pairing an input and output event. Under a broadcast model, event renaming allows connections to be formed by renaming the output event so that it matches the input event (or vice versa). Mechanically, this means modifying the source of the statechart to use the new event names.

To prevent interference with existing connections, pre-existing names must not be unintentionally duplicated in the a new module. Thus, if a new module uses event names that are already present in the target system but no connection is intended, then events in the new module should be given a new, unique name thereby preventing the formation of unintended connections.

As a rule, private events are not appropriate for use in forming connections. By definition, a private event is used internally by a statechart, and encapsulation implies that the event should be invisible to the rest of the system. Generation of the event may depend on the internal logic of the statechart in a non-obvious fashion, meaning that generation of the private event by another statechart may break that internal logic.

Technically speaking, consumption of a private event by another statechart cannot break the internal logic. Since this breaks notions of encapsulation, such a connection would complicate future reuse and modification. However, it could be the case that an event intended as output merely happens to be consumed by an orthogonal region in the same statechart. Here, the designer should classify that event as output so that it is clear that the event can indeed be used as a connection point.

In the special case where multiple statecharts are connected using the same event, renaming may create unintended connections. Consider Fig. 6–3, where a statechart A outputs event α that is received by statechart B. New statecharts C and D are added to AI with a pre-existing communication that also uses α. Since broadcast communication is name based, A will connect to D and C will connect to B on the event α. If these new connections are not intended, then renaming could be applied, by modifying C and D to communicate on β instead of α, for instance.

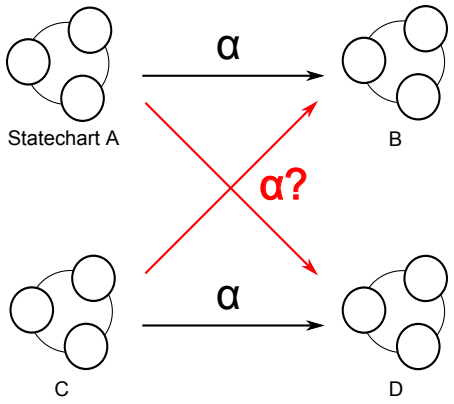


Figure 6–3: Reuse scenario with unintended connections.

Renaming can break existing connections, and can fail entirely in certain corner cases. Continuing our example in Fig. 6–3, imagine only one of the two cross connections is desired. For example, A should connect to D on α but C should not connect to B. If C outputs β, breaking the connection to B, then D will also lose connection. Renaming the input in D would break the intended connection to A. There is no broadcasting renaming solution in this scenario. In general, if a group of statecharts each want to connect to a different subset of receivers all using the same event, renaming will fail. Such situations should be avoided.

As an alternative to broadcasting with event renaming, narrowcasting can be employed. This means that each event generation specifies target statecharts, and only the targets receive the event. While this would resolve the event-renaming corner cases, it adds additional complexity at the design level by forcing developers to specify targets for each transition. Reuse of a module would force the re-specification of all event targets to match the modules in the new AI. In my experience, corner cases are quite rare and easily avoided during design, and thus do not justify the extra effort required to model event targets. Broadcast communication is more appropriate in the context of AI module reuse.

6.2.2 Associated-Class Connection

Integration relating to the associated class is less forgiving, as an unsatisfied import or synchronous method call will prevent compilation or cause run-time errors. In the case of unsatisfied method calls, either the target AI module must

also be included, or the associated class must be modified to point to a new implementing module.

In general, synchronous calls are more restrictive than event-passing, and limiting the number of synchronous calls simplifies integration. Event passing occurs at the modelling level and is easily addressed there using renaming. The following guideline helps clarify when each approach should be used: when a module receives an event and takes action immediately, that event should include relevant information as payload. When a module needs complex and dynamic state information at some undetermined point in the future, then a synchronous becomes necessary.

Module reuse across different games is constrained by imports and implementation language. The simplest case is when an AI module is purely behaviour driven and has no imports (aside from core libraries supplied with the language). Such modules are *game-agnostic* and may be freely moved between games. If a module has non-core imports, then reuse in a new game will require updating all imports, copying libraries, converting engine calls, or other more drastic rewrites. This may not always be trivial or possible, and thus designing for reuse implies that modules be made game agnostic whenever possible. The process is much more complicated if the target game is coded in a different programming language. In that case, only the statechart could be reused, while the associated class would require a total rewrite.

6.2.3 Functional groups

Reuse of AI modules allows for behavioural aspects of an AI to be exported, but the typical module is too fine-grained to fully capture a higher-level behaviour. For example, a fleeing behaviour would encompass spotting enemies with a sensor, analyzing threat, deciding on a fleeing tactic, and moving using an actuator. These modules are connected, and through their cooperative actions, realize a behavioural goal.

To allow reuse of module connections, and provide a way to modularize high-level behaviours, we introduce *functional groups*. A functional group is comprised of at least two AI modules that are connected either by an input-output event pairing or through synchronous calls.

A functional group interface can be built from the AI module interfaces of the contained modules. Input events that pair with output events can be reclassified as private to protect logic internal to the group. Unpaired events are copied to the new interface without change, and act as the connection points when the group is reused. Synchronous calls, parameters and game calls can all be added to the group interface. The end result is that member interfaces are subsumed, giving a single interface for the functional group.

Importantly, the resulting interface is identical in form to the interface for individual modules, and thus can be used interchangeably. Thus, a functional group interface is simply a special case of an AI module interface, and wherever an AI module can be reused, a functional group could instead be employed.

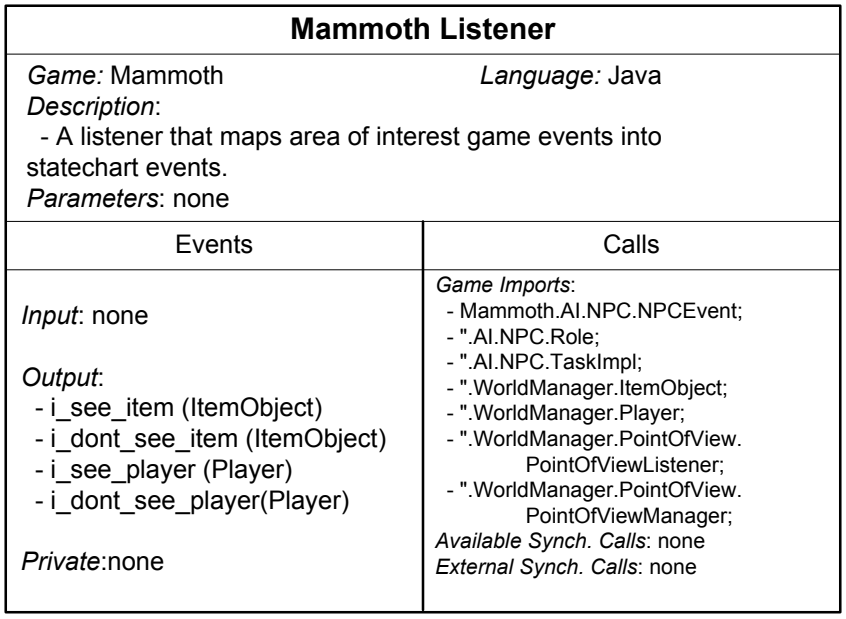


Figure 6–4: The module interface for the Mammoth Listener.

As an example, we create a functional group for identifying key items, called the *Key Item Tracker*. This will use the Key Item Memorizer, responsible for memorizing item locations and filtering key items, along with the Mammoth listener for the squirrel with interface shown in Fig. 6–4.

Group-Private events

Combining the Listener and Key Item Memorizer gives the interface shown in Fig. 6–5. Note how the `i_see_item` and `i_dont_see_item` events have been reclassified as *group-private* events. This indicates they are now part of the internal logic of the functional group, and being classified as private prevents interference from other statecharts.

Reclassification is optional, however, and decided upon by the designer creating the group. In some cases, it would be reasonable for other statecharts

Key Item Identifier Functional Group

Game: Mammoth *Language*: Java

Description: Spots items and players. If they are key items, they are memorized and a notification is created.

Modules: KeyItemMemorizer, MammothListener

Parameters:
- ItemObject keyItem :: The item type to be memorized as the key item.

Events	Calls
Input: none *Output:* - key_item_visible - no_key_item_visible - i_see_player (Player) - i_dont_see_player(Player) *Private:* - i_see_item (ItemObject) - i_dont_see_item (ItemObject)	*Game Imports*: - Mammoth.AI.NPC.Role, - ".AI.NPC.NPCEvent, - ".AI.NPC.TaskImpl, - ".PhysicsEngine.PhysicsEngine, - ".WorldManager.ItemObject, - ".WorldManager.Player, - ".WorldManager.PointOfView. PointOfViewListener, - ".WorldManager.PointOfView. PointOfViewManager, - ".WorldManager.WorldManager *Available Synch. Calls*: - Vector<ItemObject> getKeyItemList()

Figure 6–5: The module interface for a functional group.

to generate a group-private event. Continuing our example, we assume that our Listener works on line of sight. If another statechart was added that worked on sense of smell, then it too could create `i_see_item` events to easily include smell data in the AI decision making process.

6.3 Case Study: Squirrel to Trash Collector

To demonstrate the validity and usefulness of the presented approach, this section gives a concrete example of AI reuse. Here, we take the AI developed for the squirrel and reuse components of it to create a new AI for a trash collector. Using the described reuse techniques coupled with good modular design practices,

we find that many elements of the squirrel can easily be repurposed, greatly simplifying the development of the new AI.

This process is greatly streamlined by the fact that the source and target game are the same. Thus, the associated classes can be reused without modification, as the implementation language is the same and all imports are valid.

6.3.1 Trash Collector Specification

The purpose of the trash collector NPC is to clean up the Mammoth game world. This is done by collecting pieces of rubbish left by other players, then depositing the trash into garbage bins. The requirements for the NPC are for it to explore the game world, spot garbage bins and pieces of trash, pick up pieces of trash, and drop trash into receptacles.

The high level behaviour of the trash collector differs from that of the squirrel. While the squirrel has four high level goals (wandering, gathering food, eating, and fleeing), the trash collector has three: searching, picking up trash, and depositing trash. Searching and wandering seem similar, and gathering food and picking up trash are also similar, and we start our reuse from there.

Exploring

In the squirrel, wandering is performed by means of a `start_wander` event that is received by the Wander Executor. In turn, the Wander Executor uses the Move Actuator to effect movement plans. These already perform a wandering behaviour, and thus can be directly reused in the trash collector.

To preserve the existing connection, we create a new functional group called *Wander Move*, comprised of the Move Actuator and the Wander Executor. This

new group is connected through the `move`, `move_successful`, and `move_failed` events. However, other parts of the squirrel AI connect to the Move Actuator, and so in creating the group, we will not reclassify these events as being group-private.

Collecting Trash

To collect acorns, the squirrel performs two separate functions. First is finding them, and then next is collecting them. Finding requires object identification, handled by the Key Item Tracker group introduced in §6.2.3. This group can be reused by updating the key item parameter to use a 'Trash' item object instead of 'Acorn'. This allows the trash collector to spot and identify trash objects.

Once the squirrel brain has decided to collect food, the Eat Decider translates the goal into actions, including picking up food. Only the picking up portion of this behaviour is useful for the trash collector, and so we build a Pick Up functional group for reuse. Included are the Pick Up Executor and the Pick Up Actuator, connected via the *pick_up_item* event. The Pick Up Executor can also move to targets, and connects with the Move Actuator to accomplish this. Since the Move Actuator has already been added to the new AI through the inclusion of the Wander Move group, this connection does not need modification. The Pick Up Executor makes a synchronous call to the Key Item Memorizer, but this module is in place and so the call is satisfied.

Using Trash Receptacles

The new behaviour in trash collector is to use trash receptacles. Since the reused Mammoth Listener, a Sensor module, creates a game event for every spotted item, the AI is already aware of garbage cans. We can add a new instance

of the Key Item Memorizer, called a Trash Can Memorizer to receive these previously unused notifications and track garbage cans. However, this creates a unintended connection on the *key_item_visible* event. Event renaming resolves this, and so we have the Trash Can Memorizer generate *trash_can_visible* and *no_trash_can_visible* events instead of key item events.

Once trash has been collected, the NPC must move to the garbage can and drop the trash in the can. Fundamentally, this is the same logic as the Pick Up Executor, which handles moving to and picking up an object. We create a Drop Executor by renaming all events with *pick_up* into *drop*, e.g., *pick_up_item* becomes *drop_item*, and by changing the target of the synchronous call to the Trash Can Memorizer. The *drop_item* is currently unconnected, and will form the connection to a yet-to-be introduced drop actuator.

6.3.2 Building the NPC

With most behaviours now in place, the only design task left is to fill in missing behaviours with new modules, and connect them all with a new strategizer. Working from the module interfaces, we can determine the exact events that are as yet unconnected. These are summarized in Table 6–1.

Table 6–1: Unconnected Events in the new Trash Collector AI

Unconnected Inputs	**Module/Func. Group**	**Unconnected Outputs**
start_wander *stop_wander*	Wander Move FG	—
—	Key Item Tracker FG	*key_item_visible* *no_key_item_visible*
pick_up_item_request	Pick Up FG	*pick_up_successful* *pick_up_failed*
—	Trash Can Memorizer	*trash_can_visible* *no_trash_can_visible*
drop_item_request	Drop Executor	*drop_item*

The Trash Collector Brain will be somewhat different than the Squirrel Brain. The behaviour desired is to pick up a single piece of garbage and throw it in the trash, repeating indefinitely. The brain is thus a simple two state statechart, alternating between the two goals based upon if the NPC currently has a piece of trash.

A new decider was created to implement the high level collect goal. The Collect Decider, shown in Fig. 6–6, realizes the collect goal by searching for a piece of trash, then picking up a piece of visible trash. When the collection is successful, the Brain will stop collection, and start the Discard Decider. It is identical in form to the Collect Decider, searching for a garbage can, then throwing the trash in the can, with the events renamed to match the connections in Table 6–1. The astute reader will notice these are instances of the priority decider pattern from the previous chapter. In addition, a new Drop Actuator was created to receive the *drop_item* event and have the NPC actually drop the carried trash into the can.

6.3.3 Case Study Summary

In total, the new AI is comprised of 12 modules, 8 of which have been reused. This means that only one third of the modules in the new AI were newly designed. Of these four, two were statechart patterns, one had only two states, and the other had only a single state. By avoiding redevelopment of existing AI logic, the workload to develop the new AI was dramatically reduced.

The reuse process itself does take some effort. However, by representing module connections in a clearly defined fashion, and through the use of functional groups, the act of module integration was greatly simplified. Based upon this case

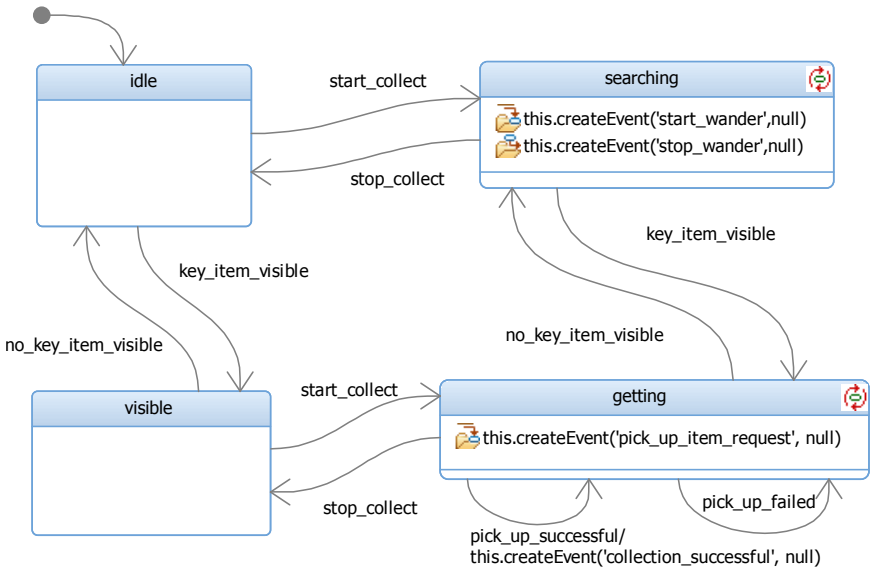


Figure 6–6: The statechart for the Collect Decider.

study, we can conclude that our reuse techniques are a valuable approach for AI designers. In the next chapter, we dramatically improve the process by providing tool support for the reuse process through the tool Scythe AI.

CHAPTER 7
The Scythe AI Tool

Modular reuse is greatly simplified through the use of the AI module interface. The next step is to make reuse practical by providing a work flow complete with tool support. Designed to be used by AI developers, our tool *Scythe AI* is the key piece of the reuse work flow.

In practice, performing reuse involves numerous small tasks, such as managing files and interfaces, checking that event names correspond, tracking synchronous calls, and so on. Small mistakes become possible as the number of modules grow. The intent of Scythe AI is to automate these tasks where possible and provide guidance as needed, thereby simplifying the reuse process and reducing the possibility of error.

This chapter describes how Scythe AI operates, and details the logic behind interesting design decisions. The discussion is divided by workflow, covering module importing, building new AIs through reuse, and exporting completed AIs. It will conclude with a discussion of avenues for future development.

7.1 Overview

One of the main factors enabling Scythe AI is the well-defined module interfaces in layered statechart-based AI. Reuse requires modules with defined module interfaces, and the tool is built around manipulating these interfaces. Scythe AI provides users with the ability to build a library of AI modules each

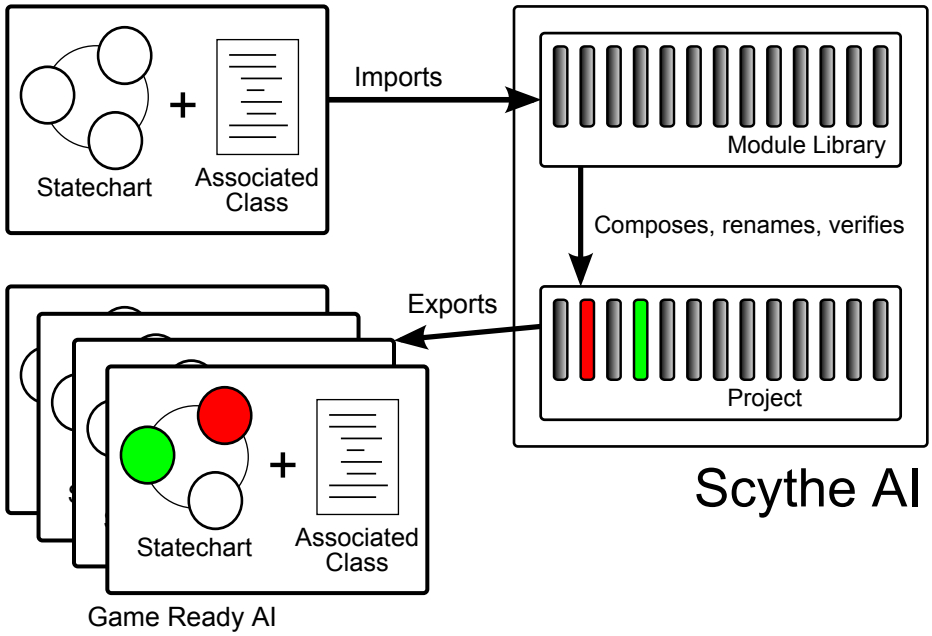


Figure 7–1: The Scythe AI Workflow.

with properly defined module interfaces. The overall Scythe AI workflow is illustrated in Fig. 7–1. After importing modules to the library, the developer can build a new AI by piecing together modules. Like a standard IDE, warnings and errors are generated to guide users through this process of connecting modules. This ensures proper interaction between modules and reduces the possibility for error. When an AI is completed, it can be exported directly from Scythe AI into to a target game. In doing so, Scythe AI executes the changes to the modules, then packages up the files for the target game.

The main interface for Scythe AI is shown in Fig. 7–2. Displayed prominently on the left is the Module Library, which lists all AI modules that have been imported into Scythe AI. Users will interact with this heavily when building

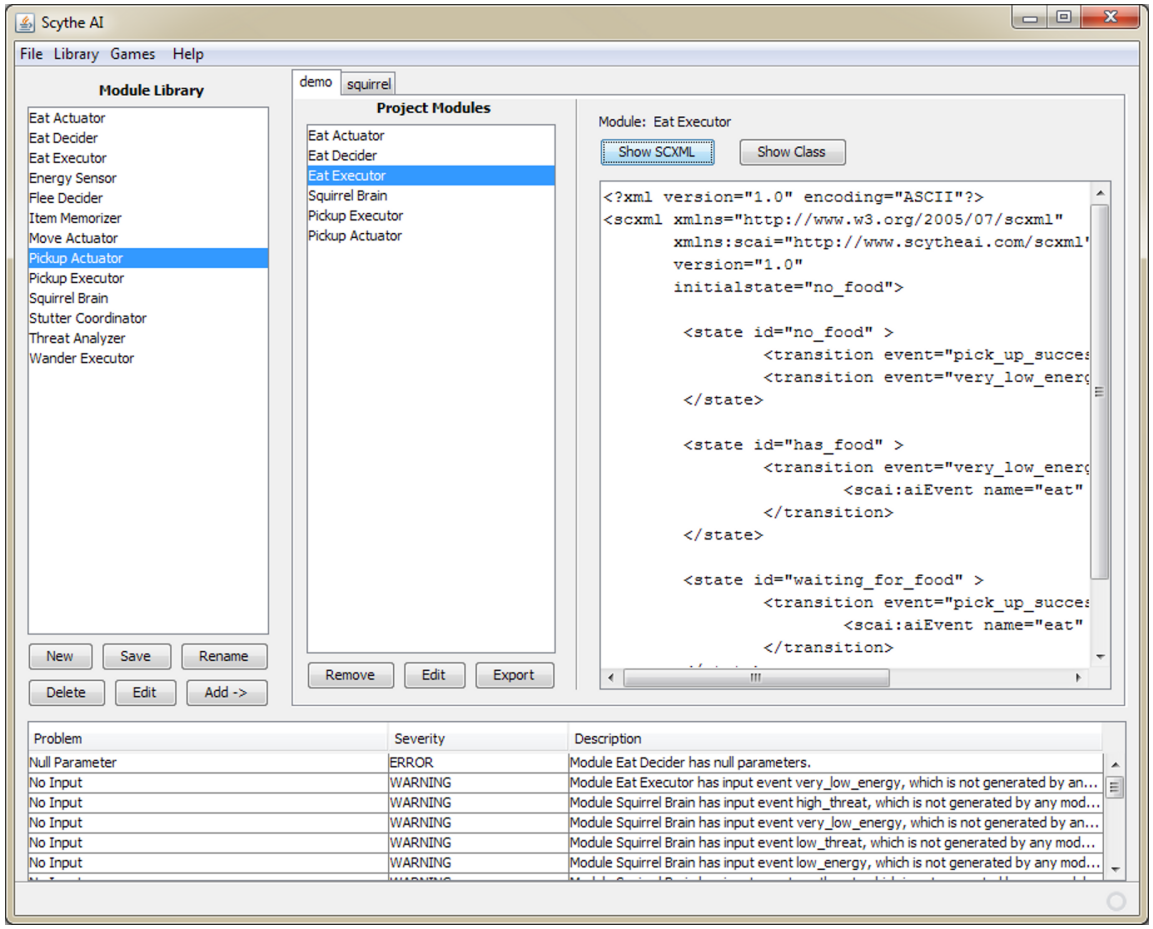


Figure 7–2: The Scythe AI Main Interface.

AIs. Any AIs under construction show up in the tabbed portion. This area lists modules in the project, and allows users to view the constituent files of a module. Making the source code and SCXML easy to view allows users to examine modules and learn how they operate. This allows users to make better informed decisions about how to integrate modules. The bottom of the screen lists problems generated by the currently selected project. The user will need to examine this regularly to ensure the modules are being integrated correctly.

File management is handled through the concept of a workspace. When Scythe AI is launched, the user selects a workspace folder. Inside, the program creates a 'projects' folder to hold all files related to user projects, and a 'library' folder to hold all files comprising modules that have been added to the Scythe AI library.

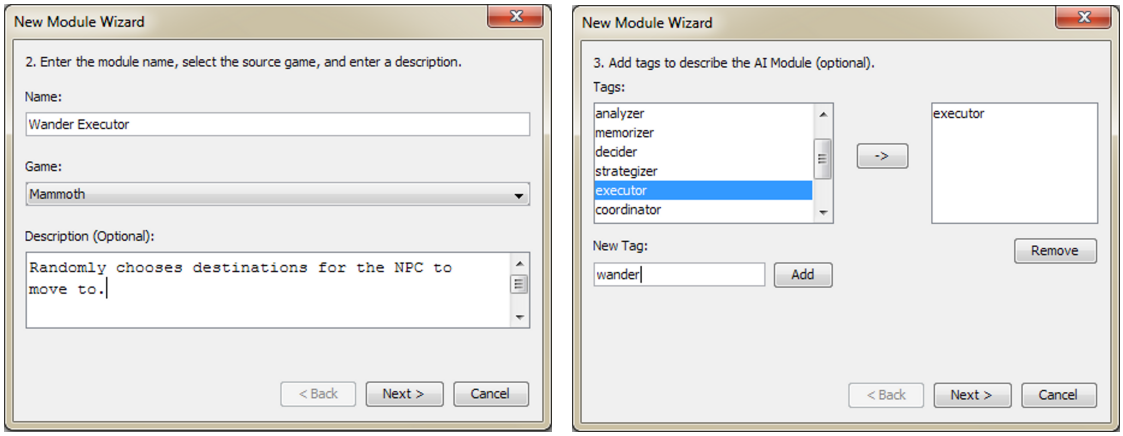
7.2 Importing Modules

The task of importing an AI module into Scythe AI is fundamentally the act of defining the module interface. Accordingly, a main feature of Scythe AI is the module import wizard, which guides the user through the process of correctly building the interface. This ensures that the module interface is both complete and correct, making imported modules ready for reuse.

The first step in the module import wizard allows the user to select the files that comprise the module. This includes both the Statechart in `.scxml` format and the associated class coded in Java. Scythe AI internally handles these files through extensible interfaces, allowing for future extensions supporting other formats and languages.

The next two wizard steps, shown in Fig. 7–3(a) and Fig. 7–3(b), prompt the user to provide meta-information about the new module. This allows the user to customize the module name, source game, and description. Additionally, it allows modules to be tagged with their purpose. This information will assist future users in ascertaining if a given module fits their needs.

Having completed the input of meta-information, the next step is to build the portion of the interface relating to event-based communication. This derives



(a) Import Module Step 2.

(b) Import Module Step 3.

Figure 7–3: Entering general description of a module.

directly from the Statechart. Rather than have the user manually enter events, Scythe AI parses the `.scxml` file and provides a list of events to the user. This is done using the Commons SCXML parser [2]. Scythe AI searches for all event generations in the XML namespace `xmlns:scai="http://www.scytheai.com/scxml"` using the tag `<scai:aiEvent name="ev" payload="payload"/>`.

Having completed parsing, Scythe AI can show step 4 of the import wizard, where the user specifies the direction of event interactions. This is shown in Fig. 7–4. Scythe AI supports users in this by offering a guess about how events are used. If an event is generated within the Statechart, it is assumed to be an output event. Each event used by a transition is assumed to be an input. These must be modifiable, since any of these events could be private. Additionally, events have a source which is either the AI or the game itself. While this is not formally part of the module interface, specifying the event origin is useful in finding warnings and errors as we will see later.

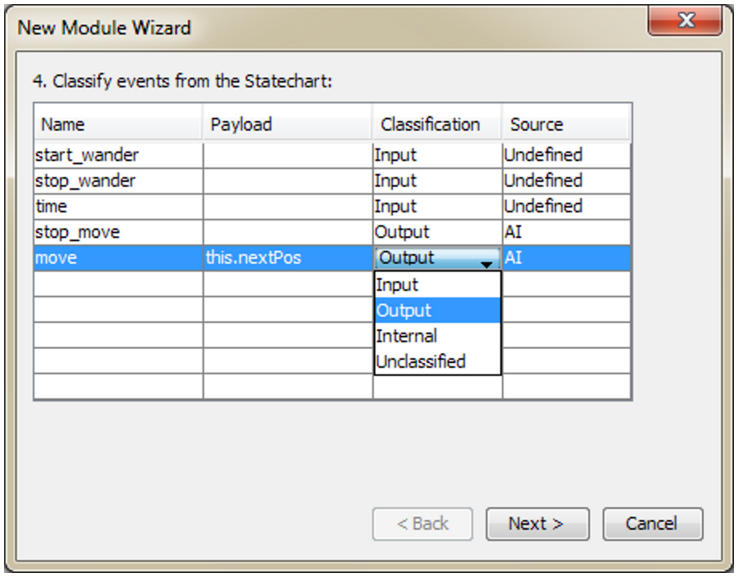


Figure 7–4: Import Module Step 4: Event Specification

Next, the user must build the portion of the interface deriving from the associated class. Once again, Scythe AI supports the user by parsing the provided .java file. This allows the user to define the remainder of the interface by working through a few customized menus, greatly simplifying the process for the user.

In step 5 of the wizard, the user is given a simple list showing all imports in the associated class. Any game specific imports should be should be checked off, and the rest unchecked. These are then added to the interface.

Step 6 gives the user a listing of all public methods defined in the class. Those that are to be made available to other modules as synchronous method calls should be checked, and the rest should be unchecked. Since we only parse the raw Java source for a single class, inheritance presents a problem. Lacking the source for any

super classes, no inherited methods will appear. In practice this has not caused any issues, but it is a limitation of the current approach.

Automatic detection of outgoing synchronous calls is not supported at this time. This is due to the inherent limitations of static analysis with respect to determining the runtime type of an object. Java supports polymorphism, and so examination of a specific method call cannot definitely indicate the destination of the call. For example, a synchronous call to another module may be programmed using a call to the superclass of the module. This would not be meaningful without understanding the class hierarchy of the module. Since we do not know the module hierarchy, this presents a roadblock. As a workaround, any outgoing calls must be entered manually. Future versions of Scythe AI could alleviate this by introducing a more complete code analysis tool, such as Soot [42].

In step 7, shown in Fig. 7–5, a list of all non-final parameters is given to the user. The user checks off the parameters that should be included in the module interface; the rest are ignored. Again, because only the single class file is provided, parameters originating in a super class cannot be detected.

With completion of step 7, the AI module interface is now completely specified. Upon closing the wizard, Scythe AI adds the new module into the workspace. In the library folder in the workspace, a new folder is created and populated with copies of the Statechart and class files. Next, Scythe AI writes an XML representation of the interface for the newly defined module. Thus the module is stored and later users of Scythe AI can access it. In the main interface, the new module is added to the library module list, and is available for reuse.

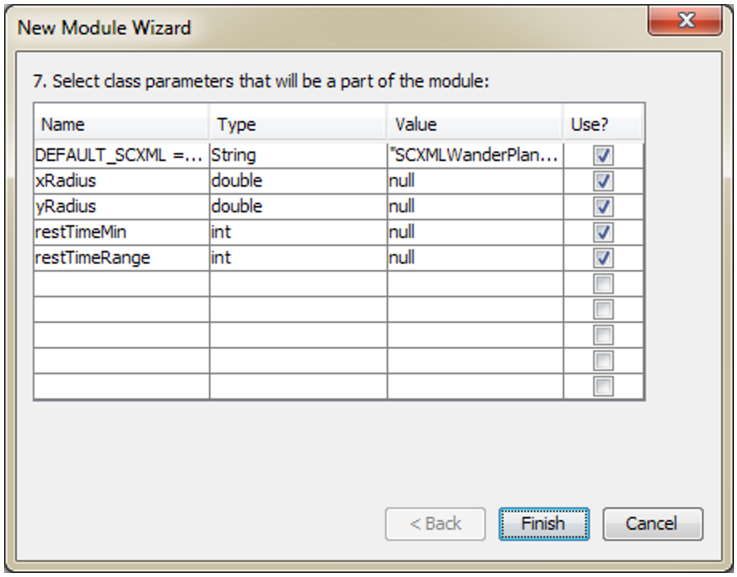


Figure 7–5: Import Module Step 7: Parameter Selection

7.3 Building an AI

In Scythe AI, building an new AI is done by composing AI modules that have been added to the library. A new project is created to store the AI, then modules are added to the new project. After adding the desired modules, the user must customize modules as needed such that the modules communicate as intended.

Scythe AI creates a new folder in the workspace for each new project. When a user adds a module to the project, the tool copies the Statechart, associated class, and module XML file into the project folder. This allows the module to be customized for the current project without affecting the original copies in the library, or other projects reusing the same module.

Modules are customized by selecting either a library or project module and pressing the edit button. This brings up the edit module dialog shown in Fig. 7–6.

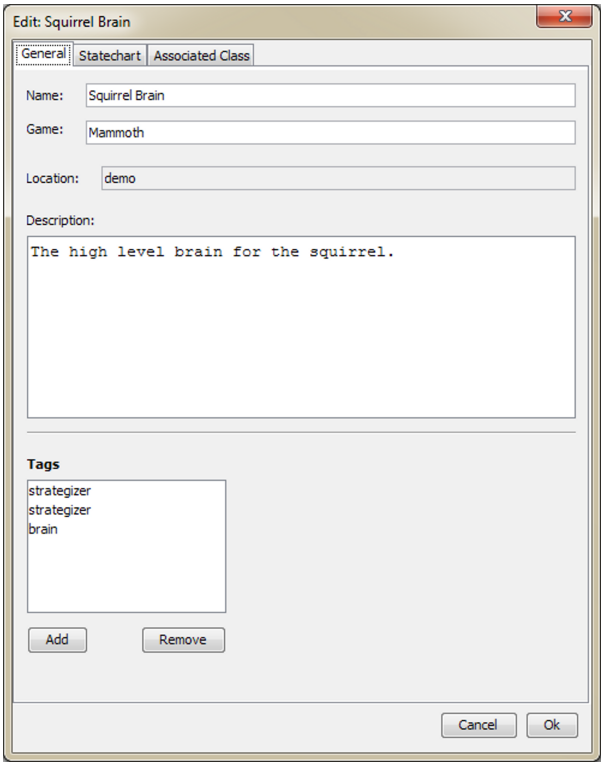


Figure 7–6: Editing a Module in Scythe AI.

The primary task for the user is to ensure that events sent by a Statechart are correctly received by target Statecharts, while not being incidentally received by non-target Statecharts. This is done by event renaming as described in Chapter 6. Scythe AI supports this by allowing users to rename events in the edit module dialog.

When a module is added to a project, the user must specify values for any parameters in the module. For example, the Key Item Memorizer found in the Squirrel AI takes an item name as input that determines the key item to be memorized. For the squirrel, it is an Acorn, but in a reuse scenario, the user will

want to customize the key item parameter as needed by the new AI. This is also done in the edit module dialog, under the Associated Class tab.

7.3.1 Errors and Warnings

A number of problems can arise while constructing a new AI. By identifying problems and notifying the user, Scythe AI provides excellent support to developers performing reuse.

A problem is classified as an *error* if it will prevent the AI from running (e.g., a synchronous call is not satisfied), and must be corrected before exporting the AI. If the problem would not prevent the AI from running, it is instead classified as a *warning*. Each problem is listed in the main interface so that the user has a continual description of potential issues. As the user makes changes to the AI, the list is continually updated. Table 7–1 gives the listing of errors and warnings that can be generated by Scythe AI.

Table 7–1: Warnings and errors generated by Scythe AI

Severity	**Problem**	**Description**
Error	Event Interference	Event e is private in module x, but is used by module y.
Warning	No Input	Module x has input event e, which is not generated by any module.
Warning	No Receiver	Module x outputs event e, which is not received by any module.
Warning	Game Conflict	Module x has input event e sourced by the game, but e is generated by another module y
Error	Game Mismatch	Module x has game imports for g when target game is j.
Error	Unsatisfied Call	Module x calls m in $class$, which does not exist.
Warning	Unused Call	Module x provides method m which is never called.
Warning	Null Parameter	Parameter p in module x is null.
Warning	No Actuators	Project has no actuators. Resulting AI cannot act.

The warnings and errors follow from the module interface. Looking at the event-based communication portion of the interface, it should be expected that input events are generated by some other module, outputs are received by another

module, and so on. If these conditions are not met, it could indicate a problem with how the module is being used.

As an example, the 'No Input' warning means that a module expects an input event that no module generates. This might be due to a name mismatch, accidental omission of another module, or may be intentional. By presenting this as a warning, the user is notified of the potential issue and can address it as they see fit. Other problems can be clearly identified as errors. An event classified as private implies that no other module should interact with that event. If another module does interact with that event, this violates the module interface of the first module, and must be corrected by the user. Errors and warnings relating to the associated class are similar in nature: outgoing calls must be satisfied while unused offered calls present a warning.

Worth noting is that events originating at the game cannot trigger a 'No Input' warning. Since they are generated by the game, it is correct for there to be no module that generates them. Indeed, if another module does generate a game-sourced event, then a 'Game Conflict' warning will be generated.

One additional error, 'No Actuators', does not follow from the interface. It refers to the layered Statechart approach wherein only actuators can modify the game state. This is a type of meta-error, in that all module interfaces can be properly satisfied, yet the constructed AI is likely to be incorrect. Future iterations of Scythe AI could potentially include additional warnings of this type.

7.4 Exporting an AI

Once the AI is complete, the final step is to export it to the target game. Doing so is a three part process. First, any changes made in the AI construction phase must be applied, necessary information from the module interfaces must be included, and the AI must be packaged up appropriately.

The changes made in the building phase are primarily a result of event renaming. Applying this change means editing the SCXML source such that event names are updated. Since the source is being modified directly, care has to be taken to ensure that changes are applied correctly. An event name x is only matched and replaced when it occurs inside the tags `<scai:aiEvent name="x">`, `<scai:gameEvent name="x">` and `<transition event="x">`.

Currently, Scythe AI can output to the game Mammoth. Importantly, Mammoth allows NPCs to be defined externally in an XML file. Scythe AI knows how to write this XML file, and thus can output the AI in a format ready for use in Mammoth. The basic Squirrel AI, detailed in Appendix A, yields the output shown in Fig. 7–7 when exported from Scythe AI. As you can see, the output specifies the class and SCXML files that make up the module, and provides values for each parameter. Along with generating this XML definition, the export operation also copies all SCXML and class files to the output folder.

7.5 Future Development Plan

In its current state, Scythe AI is an effective tool to manage reuse of Statechart-based AI. We have modelled the AI for several NPCs within Scythe AI, and found it to be an excellent environment for the rapid composition and

```
<?xml version="1.0" encoding="UTF-8"?><role xmlns="http://mammoth.cs.mcgill.ca" name="Squirrel2">
    <tasks>
        <task class="SCXMLEatActuator.java" type="scxml">
            <scxmlFile value="SCXMLEatActuator.scxml"/>
        </task>
        <task class="SCXMLEatDecider.java" type="scxml">
            <scxmlFile value="SCXMLEatDecider.scxml"/>
            <reference value="Mammoth.AI.NPC.SCXML.SCXMLItemMemorizer"/>
        </task>
        <task class="SCXMLEatAnalyzer.java" type="scxml">
            <scxmlFile value="SCXMLEatAnalyzer.scxml"/>
        </task>
        <task class="SCXMLEnergySensor.java" type="scxml">
            <scxmlFile value="SCXMLEnergySensor.scxml"/>
            <lowThreshold value="15000"/>
            <veryLowThreshold value="7000"/>
        </task>
        <task class="SCXMLFleePlanner.java" type="scxml">
            <scxmlFile value="SCXMLFleePlanner.scxml"/>
        </task>
        <task class="SCXMLItemMemorizer.java" type="scxml">
            <scxmlFile value="SCXMLItemMemorizer.scxml"/>
            <keyItem value="Acorn"/>
        </task>
        <...>
        <task class="SCXMLProximityMemorizer.java" type="scxml">
            <scxmlFile value="SCXMLProximityMemorizer.scxml"/>
            <lowThreat value="2.0"/>
            <highThreat value="1.0"/>
        </task>
        <task class="SCXMLWanderPlanner.java" type="scxml">
            <scxmlFile value="SCXMLWanderPlanner.scxml"/>
            <xRadius value="2"/>
            <yRadius value="2"/>
            <restTimeMin value="5"/>
            <restTimeRange value="5"/>
        </task>
        <task class="Mammoth.AI.NPC.Tasks.ListenerTask" type="internal"/>
    </tasks>
</role>
```

Figure 7–7: The Mammoth XML NPC generated by Scythe AI

deployment of game AI. That being said, there are several specific improvements that could be made that would greatly add to the capabilities of Scythe AI.

Verification of the correctness of the AI is a major concern. With the error and warning system, Scythe AI can tell the user that module interfaces are satisfied, but it cannot verify that the behaviour of the assembled AI will exhibit the desired behaviour. Thus, it would be extremely valuable to have Scythe AI perform some of the verification techniques discussed in Chapter 8. At the time of writing, Scythe AI can generate Promela representations of the Statecharts in a project, but cannot include specifications in that code. Scythe AI could be expanded to allow users to make behavioural specifications for an AI, and then include them in the Promela output for verification.

Along the same lines, Scythe AI should support generation of varied AIs, as detailed in Chapter 9. To do this, it would have to be possible to specify ranges or choices for parameters, add Statechart composition tags (i.e., addition, removal, and replacement candidates), as well as specification and application of rule-based transformations.

Improving the user experience would also be of great value. One useful improvement would be adding support for functional groups as defined in Chapter 6. Other wish-list features include making modules searchable and sortable by tag, adding visualization of module connections, and making it easier to alter and view SCXML and class files. Additionally, Scythe AI uses interfaces to internally represent Statechart and associated class files. This was done so that it would be possible to extend the program to allow C++ classes, C# classes and `.xmi`

Statecharts as input. Doing so would allow Scythe AI to work with a larger number of games.

One last feature that would be highly useful would be the addition of output profiles. This would allow users to define in a domain-specific language how to output to a specific game. Users would then be empowered to use Scythe AI for their own games. As part of this, it would useful if Scythe AI could export Statecharts as C++, C#, or Java code, removing the need for an SCXML execution engine in the target game. This would allow users to output from Scythe AI into a general target, such as Unity or the Unreal engine.

CHAPTER 8
Verifying Correctness

A major issue in game AI is verifying that NPCs behave as expected when performing their role in the game context. Testing is the typical verification approach, where developers perform informal tests during initial design, followed by dedicated game testers that manually interact with the AI. In some instances, this is followed by beta testers during a closed beta or external testers in an open beta, who return feedback through forum posts. The process is time consuming, inefficient, and leaves gaps in test coverage. When the final product is released to thousands or in some cases millions of players, these bugs will begin to appear, negatively impacting the game play experience. With the rise of social media, even a rare bug might be seen by a large number of current or potential customers [8].

Layered statechart-based AI, as a model-based approach, allows for a new approach to testing game AI. Since the model is built on a clearly defined formalism, it is possible to establish the correctness of the model itself. This allows designers to verify behavioural correctness at the same level of abstraction as the design goals, giving them a potent new tool with which to validate the correctness of their AI. Especially in the case of modular reuse, this verification is essential to the development process.

In this chapter, we develop an approach to the verification of layered statechart-based AI. Our treatment divides the problem into two types of verification. The first is *syntactic correctness*, where we establish that modules are correctly connected and capable of communication. Next, we explore *semantic correctness* whereby we verify that the AI logic meets behavioural specifications. This includes a discussion of how to express behavioural requirements as specifications, and outlines a method to generate game scenarios for automated testing purposes.

8.1 Syntactic Correctness

Layered statechart-based AI is fundamentally modular, with each module communicating by generating and receiving events through broadcast communication. While this loose coupling simplifies the modular reuse process, it creates the potential for error through mismatched event names. Thus, it is essential to the design process for developers to quickly and easily verify that statechart events are correctly named and thus statecharts can communicate. We call this *syntactic correctness*.

At its core, this idea is quite straightforward. Each event generation and reception was intentionally created by the statechart designer, and so our verification starts with the assumption that every event must be able to effect a change within the system. This means that every generated event should be consumable, even if specific instances are not consumed. Similarly, it should be the case that every event that can be consumed is indeed generated at some point. Finally, we need

to make sure that private events are respected, and neither consumed nor generated by other statecharts. We summarize these requirements as a list of syntactic verification goals:

- All events intended to be used by the AI are indeed generated.
- All events intended to be generated by the AI are indeed usable.
- No private event is used in any connection between modules.

With this, we can guarantee that the events in the AI are correctly utilized, and that statecharts are correctly connected. This gives reasonable assurance to the developer that the AI is correctly assembled, an important first step in verifying the overall correctness of the AI.

Putting this in the context of AI module interfaces, we treat the problem in terms of input and output events. Given a modular AI, let O be the set of all output events from all statecharts, and let I be the set of all input events. In principle, we wish to verify that $O = I$, as this implies that all events are both generated and consumed. In other words, we wish to verify that:

$$\forall o \in O, \{\exists i \in I \mid o = i\}$$

$$\forall i \in I, \{\exists o \in O \mid i = o\}$$

This will catch any errors related to simple misnaming (e.g., `pickup`, `pick_up`, `pickUp`, and `ev_pickup`). It will also catch errors due to missed statecharts. For example, a `move` event that is generated but not received might indicate to the designer that a Move Actuator was accidentally omitted during reuse.

With the correct set of input and output events, we can now verify that private events are used correctly. Letting P be the set of all private events from all statecharts, then it should be the case that:

$$\forall p \in P, \{p \notin I \wedge p \notin O\}$$

This tells us that no statechart accepts as input or generates as output any event marked as private.

8.1.1 Reuse Considerations

Practically speaking, it might be the case that some events are deliberately not employed. In a reuse scenario, a module producing multiple outputs might only be reused to produce one specific output, while the rest are not relevant to the new context. Similarly, a designer might be leaving extra events as hooks for future modifications or future reuse. Regardless of the reason, the designer should actively build ignore sets $G_o, G_o \subset O$ for output ignores, and $G_i, G_i \subset I$ for input ignores by adding events that are to be ignored during verification. Thus, we instead verify that:

$$\forall o \in (O - G_o), \{\exists i \in I \mid o = i\}$$

$$\forall i \in (I - G_i), \{\exists o \in O \mid i = o\}$$

Ignore lists used in this regard tell us that the designer does not care if the ignored output has a matching input, or vice versa. It is not an exclusion list, and thus it would be acceptable for a non-ignored event to connect to an ignored event.

8.1.2 Applying Syntactic Correctness

This approach to syntactic correctness is fully implemented in Scythe AI. Through the error and warning system described in §7.3.1, any violation of syntactic correctness is displayed to the user so that they can correct the error. Being able to create a tool that automatically checks for these violations is one of the primary benefits of the model-driven approach.

Verifying the Halo AI

To demonstrate utility, the Halo AI developed in Chap. 5 was loaded into Scythe AI and analyzed for syntactic correctness. The full complement of 50 statecharts was used, which in total had 309 events defined in their interfaces. This lead to a total of 116 unique input events, 118 unique output events, and 9 private events. In total, 67 violations of syntactic correctness were present, indicating the ease with which syntactic errors can appear in a large AI. Behaviours relating to vehicle use and command handling were added to the ignore list, since these are in fact hooks for unimplemented behaviours. Even so, this left 49 syntactic errors.

A number of errors were due to typos. In a broadcast environment, there is no indication that these errors are present, making them difficult to isolate. Checking syntactic correctness is a valuable tool to spot these subtle mistakes. For example, the Enemy Proximity Analyzer outputted the event `ev_NoEnemiesInMelee`, while two other modules instead had `ev_NoEnemyInMeleeRange` as input, which generated 3 errors in total. Another module had `ev_PlayedSpotted` versus the correct `ev_PlayerSpotted`, for another 2 errors. Once apparent, fixing these errors was trivial. These types of errors also reveal that this kind of syntactic checking

leads to an artificially high error count, as a single error in an event name can result in a number of unsatisfied input or outputs, and thus a number of syntactic correctness violations.

Aside from minor errors, syntactic correctness verification can also identify more major oversights. For example, several modules needed to track the NPC's current position, and had as input `ev_PositionChanged`. However, no sensor existed to track NPC position. Seeing these errors pointed out an important oversight in the design process. The final version of the Halo AI shown in Appendix A adds in a position tracking sensor to generate this event. Similarly, it was often necessary to cancel movement using `ev_StopMove` event. Yet, neither the Movement Coordinator nor the Run Actuator had this functionality included, an error resulting in 8 violations. Again, seeing these errors isolated the problem, allowing the bug to be caught before testing. Fixing these errors involved modifying the Movement Coordinator to support the `ev_StopMove` event, and adding a sensor for player position.

In total, exactly 20 fixes were applied, ranging from simple typos, up to the addition of a new module and major modifications of the move coordinator. Along with the addition of 12 events relating to vehicles and commands to the ignore list, this resolved all 67 reported violations of syntactic correctness.

8.2 Semantic Correctness

Behaviours in the layered statechart model are selected based upon interactions between all statecharts, meaning the overall behaviour of the NPC is an emergent property of statechart interaction. The chain of events leading to a single

actuation can sometimes reach across all 8 layers of statecharts, each acting in accordance with their current state as established by earlier chains of events. This complexity demands an effective form of verification for developers and testers to be able to adequately establish that the AI meets its specifications.

What is needed is proof of *semantic correctness*. We define a semantically correct AI as one where the behaviours expressed in the AI logic when controlling an NPC in the game context matches the intended behaviour as specified by the designer. In other words, we want to verify that the logic encoded in the statecharts expresses what the designer meant for them for express. This represents an ambitious mark as it relies on generating a clear specification of what is correct AI behaviour, a task which is challenging.

In this section, we will develop and validate a strategy to verify the semantic correctness of a layered statechart-based AI. Such an approach is intended to be usable by designers during the creation of an AI, and as a 'unit testing'-like approach during reuse. As well, special attention will be paid on creating workable specifications to verify important elements of AI behaviour.

8.2.1 Bounding the Problem

Statecharts have well-defined semantics, and clearly defined methods of interaction. This allows us to deterministically analyze how statechart execution will proceed, given a sequence of input events. Thus, verification of semantic correctness can be performed by simulating statechart behaviour to determine if behaviours are being expressed correctly. For instance, a likely requirement would be that if the NPC is in a dangerous position, then the AI should make

the decision to flee. This would eventually be translated into a function call at an actuator. In this case, we can say the flee behaviour is semantically correct if, given a game state where the designer would like the AI to flee, the AI logic produces the decision to flee, and effects the appropriate actuation strategy.

Of course, semantic correctness does not guarantee the NPC will function correctly. If, for instance, there is a bug in pathfinding, then moves may fail, the NPC may become stuck, and so on. These types of problems, however, arise based on errors in the game code, rather than in the AI logic. In this case, the correct decision is made, but the underlying structure executing that decision is flawed.

Classifying AI bugs as either failures in logic, or failures in the code used to realize behaviours creates a clear demarcation point. If the AI logic makes correct decisions based on the specified behaviour, then we will consider it semantically correct, regardless if those decisions are correctly executed by the underlying game implementation. While there do exist many approaches to verify code correctness (such as static analysis [11] or automated theorem provers [46]), we will focus on verifying AI logic exclusively.

8.2.2 Model-Checking

To verify that a behaviour correctly meets requirements, we must explore how the AI reacts to any given event occurrence from any and all potential AI states. However, the number of potential AI states is extremely large, as it is equal to the number of states in the Cartesian product of all statecharts in the AI. A useful approximation of this is n^m, where n is the number of states in an average

statechart in the AI, and m is the number of statecharts. Calculating this precisely for the Halo AI, this works out to 2.41×10^{22} states.

The vast majority of these states are spurious. Executors act in mutual exclusion to each other, and have an almost hierarchical relation with states in the deciders. Similarly, deciders act in mutual exclusion and are essentially hierarchical substates of states in the strategizer. For example, the size of the Cartesian product of states of deciders in the Halo AI is exactly 3000. This was determined by counting the number of states in each one and multiplying. However, manual inspection determined that only 84 combinations are actually viable, given our strategy of using start and stop events to ensure that deciders operate in mutual exclusion. While this effect reduces the size of the state-space by several orders of magnitude, the fact remains that the number of viable combinations precludes manual testing as an effective approach.

What is needed is a verification approach that will effectively explore the non-spurious state-space of the AI, a problem that is directly addressed by *model-checking*. One of the primary advantages of model-checking is its ability to effectively and exhaustively explore large state spaces. A model checker will examine every possible ordering of events such that it can establish correctness of a given specification. By transforming an AI into a verifiable model, and by expressing design requirements as formal specifications, it becomes possible to use a model checker to verify layered statechart-based AI.

The Spin Model Checker

One of the most venerable model checkers is Spin [30], designed with the principle goal of efficient verification of multi-threaded software. Spin is highly appropriate for verification of statechart behaviour, and there exist several detailed approaches to using it for this task [51, 44, 60]. For these reasons, we used Spin to verify the semantic correctness of layered statechart-based AI.

Through the use of on-the-fly state-space generation, support for multi-core systems, and use of partial-order reduction [57], Spin can efficiently model-check large software systems. Both safety and liveness specifications given in Linear Temporal Logic (LTL) can be verified. A full guide to Spin usage can be found in the Spin Primer [29].

The process meta language *Promela* is used by Spin to represent the process behaviour of the system to be verified. A typical approach to verification is to first transform the system to be verified into Promela code, add specifications describing correct behaviour, and then model check that representation for specification violations. Relevant details regarding Promela will be given as necessary; readers interested in a complete description are referred once again to the Spin Primer [29].

8.2.3 Promela Representation

To represent statecharts in Promela, we will largely follow the treatment given by Latella et. al. [51]. There, statecharts were first transformed into *extended hierarchical automata*, due to difficulties in the resolution of *inter-level* transitions (an inter-level transition is one that connects two states that do not share the same

parent state). Rather than perform this complex treatment of state hierarchy, we instead flatten the state hierarchy using the inside-out Rhapsody semantics as was done by Schafer et. al. [60]. The size of the resulting transformation is larger using this method, but it is equally valid while being easier to implement.

The overall transformation procedure must address three core requirements. The first is the representation of each statechart. This must correctly capture the state structure, including complex arrangements such as orthogonal regions and history states. As well, it must correctly include on-exit, transition, and on-entry actions. Secondly, the transformation must correctly implement the game and AI queues, and must provide a mechanism to distribute events in accordance with our semantics for cooperating statecharts. Finally, the transformation must provide an abstraction of the game context to simulate the operation of the game as it relates to the AI logic. This subsection addresses each of these three requirements in turn.

Statechart Representation

Each statechart is represented in Promela as a single process. Statecharts with orthogonal regions maintain independent state for each region, and so are represented instead with a process for each region. Each process in Promela is independent and asynchronous, reinforcing the need for a mechanism to distribute events correctly. Figure 8–1 shows the basic structure, where each statechart is defined as a separate process. The Combat Decider has orthogonal regions, and so multiple processes are generated, denoted as `R0` for region 0 and `R1` for region 1. Once started, these statechart processes will run indefinitely.

```
proctype Brain() {
  do
  /* statechart contents */
  od
}

proctype Move_Actuator() {
  do
  /* statechart contents */
  od
}
proctype Combat_Decider_R0() {...}
proctype Combat_Decider_R1() {...}
```

Figure 8–1: Defining Promela processes

It is important that the process for a statechart remains active for the duration of verification, as this allows Spin to correctly identify repeated states. We achieve this by means of a global *do-od* loop in each process, as shown in Fig. 8–1.

The statechart structure itself is represented using a series of `if`-statements as shown in Fig. 8–2, which shows the Wander Executor from the Squirrel AI. The current state is matched in the outer if-statement, while the incoming event is matched in the inner if-statement. When a new event is generated, it is added to the tail of the AIEventQueue.

```
byte Wander_ExecutorState = 0;
bool Wander_ExecutorStart = false;
bool Wander_ExecutorCallback = false;
proctype Wander_Executor() {
  mtype currentEvent;
  do
    :: Wander_ExecutorStart -> atomic {
        AIEventQueue?<currentEvent>;
        if
          :: (Wander_ExecutorState == 0) ->
            if
              :: (currentEvent == start_wander) ->
                 Wander_ExecutorState = 2; AIEventQueue!move;
              :: !( currentEvent == start_wander ) -> skip;
            fi;
          :: (Wander_ExecutorState == 1) ->
            if
              :: (currentEvent == stop_wander) ->
                 Wander_ExecutorState=0; AIEventQueue!stop_move;
              :: !( currentEvent == stop_wander ) -> skip;
            fi;
          :: (Wander_ExecutorState == 2) ->
            if
              :: (currentEvent == time) ->
                 Wander_ExecutorState = 3; AIEventQueue!stop_move;
              :: (currentEvent == stop_wander) ->
                 Wander_ExecutorState = 0; AIEventQueue!stop_move;
              :: !(currentEvent==time || currentEvent==stop_wander)
                   -> skip;
            fi;
          :: (Wander_ExecutorState == 3) ->
            /* state 3 omitted for brevity */
        fi;
        Wander_ExecutorStart = false; Wander_ExecutorCallback = true;
      } /* end atomic */
  od;
}
```

Figure 8–2: The Promela Statechart for the Wander Executor

The initial state is set outside the process declaration, ensuring that execution begins in the correct default state. Since the process is always running, it requires a flag to control execution, in this case the `Wander_ExecutorStart`. In Promela, executing a flag set to false causes the process to wait. When the flag is set to true, the process wakes up and is free to resume execution. This flag will be controlled by the event processor, which will only signal when there are events for the statechart to process.

Promela if-statements are resolved non-deterministically by Spin. If more than one branch (designated by ::) has a guard evaluating to true, Spin will choose one at random and continue execution. Over a full model-checking pass, Spin will eventually backtrack such that all paths are eventually explored. In the case of events that cannot cause a transition in the current state, a guarded skip is provided, allowing execution to proceed. When model-checking, Spin will explore all active branches, meaning that the entire state-space is eventually explored.

Normally, Spin will explore all possible interleavings between each Promela statement. This is undesirable; statechart semantics assume transitions are instantaneous, and thus interleaving the steps that comprise a transition in a statechart would violate the Rhapsody semantics. We prevent this by wrapping an `atomic` block around statechart execution, preventing interleaving during statechart activity. This has the additional benefit of limiting interleaving possibilities, making model-checking more efficient.

Transitions and Actions

Executing a transition has two primary effects: actions must be executed, and the state must be updated. It is at this level that the effects of flattening the state structure become apparent. Examples of transitions represented in Promela can be found in Fig. 8–2. In our Promela abstraction, only actions that generate events are relevant. Actions that do not generate events are important to the functionality of the NPC, but have no direct effect on the logic encoded in the statecharts and are thus out of scope of the verification.

Transition actions must be fired in a specified order. To start, the transition executes any on-exit actions from the source state, which must also include those defined at parent states. Actions defined on the transition are executed next, and then finally any on-entry actions in the target state and its parent states are executed. Care must be taken here to correctly determine what states are being entered and exited (e.g., a transition between states sharing a parent should not fire on-entry and on-exit events from the parent).

The target state must also be determined. If the target state has substates and a default transition, then the actual transition target should be the default transition target. This may trigger additional on-entry actions.

History states are a special case. Like flattening substates, we again abstract away the history state and replace it with a number of transitions. When a state has a sibling history state, it is a potential target of a later transition to that history state. A tracking variable is introduced to remember the current history target. At each state with a transition to that history state, multiple transitions

are added that point to each potential history target. These are guarded by the value of the current history tracker. Thus, all potential transitions for the history structure are explicitly represented.

8.2.4 Environment Model

The behaviour of the AI is in large part a reaction to the observed game state. As the game state evolves, sensors in the AI create the events that cause the AI to change states, eventually resulting in the execution of behaviours.

As an example, the decision to flee in the Halo AI is based upon the NPC being in danger. This state is recognizable through the series of events: {`ev_PlayerSpotted, ev_EnemySpotted, ev_AttackDetected, ev_UnderFireThreat, ev_AtRisk`}. The events generated at the sensors in response to changes in the game state are `ev_PlayerSpotted` and `ev_AttackDetected`.

In addition, events may be guarded based upon the current game state. For instance, `ev_EnemySpotted` is only generated in response to `ev_PlayerSpotted` when the guard condition `payload.isEnemy()` is true. This represents another path through which the AI takes input from the game.

In this subsection, we will describe how our Promela representation can generate game events, along with a method to model guarded events.

External Event Generation

The random, non-deterministic resolution of the if-statement in Promela provides the means to simulate external event generation. By placing each possible external event as an if-branch, Spin will generate random external events, simulating game activity. The external event generator process, shown in Fig. 8–3,

```
proctype externalGenerator() {
  do
  :: atomic {
      quiescent = false;
      if
        :: (exGuard_out_of_energy > 0) ->
            ExternalEventQueue!out_of_energy;
        :: (exGuard_time > 0) ->
            ExternalEventQueue!time;
        :: (exGuard_destination_reached > 0) ->
            ExternalEventQueue!destination_reached;
        :: (exGuard_destination_unreachable > 0) ->
            ExternalEventQueue!destination_unreachable;
        :: (exGuard_collided > 0) ->
            ExternalEventQueue!collided;
      fi;
      externalEventFire = true;
    }
    quiescent;
  od;
}
```

Figure 8–3: External Event Generation in Promela

implements this approach. It lists all events generated by the game as indicated in the AI module interface.

By only generating an event when it can be received, the branching factor inherent to external event generation can be minimized. Each external event is guarded with a flag `StatechartName_exGuard_EventName`. The guard is enabled when the statechart using that event is in an appropriate state to receive it. When a statechart enters such a state, the guard is set to true, allowing the game event to be generated.

The external event generator is trapped in a `do-od` loop, meaning it will continue to generate external events indefinitely. This allows model-checking to explore all event generation paths, looking for specification violations. After generating an event, the `externalEventFire` guard is set to true, which will trigger the event processor to begin. The generator will wait until the system becomes quiescent, at which time the process will repeat.

Guarded Transitions

In statecharts, a guarded transition is only available for execution when the guard evaluates to true. This typically depends on some aspect of the underlying model, in this case the game state. For example, the `ev_PlayerSpotted` event might result in a threat event, but only when the spotted player is close to the AI. Our Promela simulation must simulate this behaviour correctly.

Ultimately, the effect of a guard is to allow the transition to fire, or to it prevent it from firing. We model this in Promela by providing a non-deterministic choice in the event matching statement. One branch matches on the event name, simulating the situation where the transition guard is true. The other branch matches on the same event name, but no further action is taken, simulating a false transition guard. This makes it possible to simulate both cases, and allows model-checking to simulate the behaviour of the guard at the statechart level. This behaviour is shown in Fig. 8–4. This solution avoids the need to model the underlying game. Doing so would require some form of static analysis and game state modelling, both of which are outside the scope of this verification approach.

```
if
  /* guard is true */
  :: (currentEvent == stop_wander) ->
     Wander_ExecutorState = 0; AIEventQueue!stop_move;
  /* guard is false */
  :: (currentEvent == stop_wander) -> skip;
fi;
```

Figure 8–4: Guarded Transitions in Promela

Unfortunately, not all configurations of guards can be dealt with in this manner. The Binary Analyzer presented in Chap. 5 uses mutually exclusive guards on eventless transitions from a single state. In practice, this means that one transition is always enabled. When such a state is reached, it is always exited in the next micro step. In this case, adding a 'guard is false' state would assume that both guards are false, which is never the case. Without a semantic understanding of the guard conditions, this situation cannot be identified when creating our Promela representation, and so the simulated behaviour is spurious in this corner case.

As a deliberate design decision, we avoid polling caused by eventless transitions that have non-exclusive guards (see §5.2.1). This allows us to indirectly identify the correct case by assuming that any state with only eventless, guarded transitions must have mutually exclusive guards. In this case, we suppress the addition of the skip transition, thus forcing a branch to be followed. While this approach is not universally applicable, it is a suitable workaround that matches our design approach. Again we note that a complete solution requires a semantic understanding of guard conditions.

8.2.5 Event Processing

The definition of layered statechart-based AI includes two queues: one for AI events and one for external, game-sourced events. Promela provides FIFO (first in, first out) message channels for sharing data between channels, shown in Fig. 8–5. We create a queue for external events, with a size of one since only one external event is created at a time.

The second queue for AI events has a size of 255. This deliberately allows for buffer overflow. An overflow failure most likely indicates the presence of an infinite loop that generates events on each iteration. The other possibility is that the AI allows for the generation of unusually long chains of events. Given that efficiency is of primary importance in games, chain lengths should be designed to be as short as possible, and an overflow with such a large buffer indicates a failure at the design stage. In either case, a buffer overflow indicates that the AI logic should be corrected or redesigned.

Events themselves are typed as Promela `mtype`. These act as symbolic identifiers for a simple integer mapping, but show up as the identifier string instead of the numeric value during verification and simulation. This is vital in the comprehension and interpretation of execution traces.

Event distribution is handled by the `processEvents` process. It is within this process that steps and micro steps are correctly ordered. This is done by ensuring that all state charts reach quiescence before a new step occurs. Only at that point are the statecharts activated and a new step is taken. By representing this logic as a separate process, it separates the game simulation (the external event

```
mtype = collided, no_threat, start_get_food, stop_flee,
start_flee, path_move, pick_up_successful, stop_wander,
destination_unreachable, time, stop_get_food, execute_stand,
very_low_energy, high_threat, low_energy, low_threat, eat,
move_failed, move_successful, move, start_wander,
destination_reached, stop_move, out_of_energy, INITIAL;

chan AIEventQueue = [255] of mtype;
chan ExternalEventQueue = [1] of mtype;
```

Figure 8–5: The message passing channels in Promela

generation) from the implementation of layered statechart-based AI, modularizing the representation and simplifying any future modifications to game simulation.

The Promela representation of `processEvents` is shown in Fig. 8–6. When a new external event is received, it is placed in the AI event queue for distribution to the statecharts. Next, the execution flag for each statechart is set to true allowing statecharts to process the head of the AI event queue. New events are placed at the tail of the queue. This is repeated until the AI event queue is empty, meaning the system has reached quiescence. The event handler then goes to sleep, and waits for a new external event to be delivered.

8.3 Specifications

Verifying semantic correctness requires a specification, which is a precise statement of the expected behaviour of the AI. This is done in the form of formal logic, where the exact requirement is clearly defined as a logical statement. The model-checker can then explore the state-space of the AI and determine if the specification holds true. If not, it produces a counter example demonstrating how the specification can be violated.

```
proctype processEvents() {
  mtype newEvent;
  mtype processedEvent;
  do
    :: externalEventFire;
      /* get the external events and distribute them */
      ExternalEventQueue?newEvent;
      AIEventQueue!newEvent;
      do
        :: nempty(AIEventQueue) -> atomic {
            /* activate statecharts */
            Squirrel_BrainCallback=false;Squirrel_BrainStart=true;
            Squirrel_BrainCallback;
            Wander_ExecutorCallback=false;Wander_ExecutorStart=true;
            Wander_ExecutorCallback;
            Move_ActuatorCallback=false;Move_ActuatorStart=true;
            Move_ActuatorCallback;
          }
          /* clear processed event*/
          AIEventQueue?processedEvent;
        :: empty(AIEventQueue) -> break;
      od;
      externalEventFire = false;
      quiescent = true;
  od;
}
```

Figure 8–6: The `processEvents` Process in Promela

In this section, we will introduce linear temporal logic, the formalism used by Spin to express specifications. This is followed by the creation of several specifications useful in verifying semantic correctness of an AI.

8.3.1 Linear Temporal Logic

Specifications in Spin are given in linear temporal logic (LTL), which allows for propositions to be given temporal modalities. This allows the specification of future occurrences, allowing for the definition of both safety and liveness specifications. In addition to the basic logic operators (not: $\neg$, and: $\wedge$, or: $\vee$, equals: $=$, and implies: $\rightarrow$), LTL uses the following temporal modifiers on LTL formulae Ψ and Φ:

- $X\Psi$: Ψ is true in the next state.
- $\Phi U \Psi$: Φ holds true until Ψ becomes true. Ψ must eventually become true.
- $G\Psi$: Ψ is true globally, also written as $\Box\Psi$.
- $F\Psi$: In the future, Ψ is true. Also written as $\Diamond\Psi$
- $\Phi R \Psi$: Ψ is true up and including when Φ becomes true. Φ releases Ψ.

Some definitions also define a 'weakly until' operator for convenience, but it is not essential to using LTL and will not be used here. The semantics of LTL are straightforward when compared with other logic languages and will not be described here. Interested readers are instead referred to one of the many available references on LTL, such as the chapter on Temporal and Modal Logic in Emerson's Handbook of Computer Science [19].

8.3.2 Specifying AI Behaviour

The current behaviour of the AI is represented by the superposition of states. In other words, accurately specifying the behaviour requires a correct representation of the states that make up the specification. Consider a flee behaviour, where being at risk should cause the NPC to flee. In terms of states, this could be detected when an analyzer determines the NPC is at risk, and as an eventual response the move actuator should enter the moving state. In LTL this would be:

$$\Box(ThreatAnalyzer.state == high\_threat \rightarrow F(MoveActuator.state == moving))$$

This is interpreted to mean that in any state where the Threat Analyzer state is *high_ threat*, all subsequent paths will at some point in the future set the Move Actuator state to *move*. Essentially, the statement before the implication describes the states we are interested in, and the statement after describes the expected behaviour of the AI logic.

The simple flee specification is not sufficient to work as intended. First, there is no guarantee that a subsequent move is due to fleeing. The brain could chose to ignore the threat, instead deciding to initiate a wander behaviour. This would cause the squirrel to move and thus satisfy the specification. The flee behaviour itself did not cause the move and so this would be a false positive. Instead, we need to specify that once the brain decides to flee, the move actuator begins moving while the brain wants to flee. In other words, the specification must reflect both the action and the intent by matching states in multiple statecharts. We

could then refine the specification as:

$$\square(((ThreatAnalyzer.state == high\_threat) \wedge (Brain.state == fleeing)) \rightarrow$$
$$((Brain.state == fleeing) U (MoveActuator.state == moving)))$$

This more accurately captures the intent. When the threat analyzer is at risk and the brain wants to flee, then we know the AI has established the intent to flee. Our intended specification is that the AI does indeed begin moving as a result of this flee decision. Thus, we know that until such time as the Move Actuator enters the moving state (which must happen to satisfy the until modality), the brain state should remain fleeing. If the brain state changes without the actuator having entered the moving state, then the squirrel has not moved in response to wanting to flee, and the specification is violated. Since the specification is global, every arrangement where the brain state is fleeing will be checked. If this specification is verified as true, then we know that moving happened in response to the brain deciding to flee.

However, this specification is still overly broad. If the Brain decides to flee, and the Move Actuator starts to move, it is possible that it completes the move while the Brain is still in the flee state. Since the Brain wants to flee, the specification indicates that it should stay there until a move occurs, regardless of if it has already occurred! Thus, we need to be a little more specific, and indicate that we only care about moving when the Brain first decides to flee. This is

accomplished using the next operator:

$$\Box(((ThreatAnalyzer.state == high\_threat) \wedge \neg(Brain.state == fleeing) \wedge$$
$$(XBrain.state == fleeing)) \rightarrow$$
$$((Brain.state == fleeing)U(MoveActuator.state == moving)))$$

Finally, this specification is precise enough to verify the exact requirement. In general creating accurate specifications is a complex task. As requirements grow more intricate, creating specifications becomes more challenging. Publications exist on creating good specifications [43], and again the Spin Primer is a useful resource [29].

Reachability

Reachability is a common specification. The idea is that any state in the AI should be reachable, and if it is not there is a bug in the logic. This seems to be a straightforward specification. For a given statechart, a state s is reachable if there exists a series of events such that the statechart state becomes s. Since LTL does not allow explicit definition of path existence, we must declare this in terms of future occurrences.

It is tempting to say directly that a state s is reachable if:

$$F(Statechart.state == s)$$

However, the future operator declares that the condition *must* become true. If there exists even a single cycle where s is never reached, then verification will fail even if there exist other paths that reach s. A common trick in model-checking is

to instead search for the absence of the opposite condition. For reachability, we instead search that the state is never reached, which forces Spin to explore the full state-space. To wit:

$$\Box!(Statechart.state == s)$$

When Spin attempts to verify a global specification, it must examine or reason about every state to complete the verification. This prevents the verification from failing if it encounters a cycle. Thus, if there exists any path that leads to a violation, Spin will find it and return it as an error trace. A violation of 'unreachability' proves that the state is in fact reachable.

8.4 Verifying Semantic Correctness

In this section, we demonstrate the effectiveness of our approach to verifying semantic correctness. Validation proceeds in three steps. First, the correctness of the implementation of our testing approach must be established. In other words, we need to test that code is being generated correctly, that external event generation is working as expected, and so on. This is done by testing the squirrel AI against behaviours that are not implemented in the squirrel AI. If the testing implementation is correct, it will detect and report the expected failures. This is essentially a sanity check to prove that the implementation is working. Next, we perform an exhaustive (with respect to the design goals) verification of the statechart logic of the squirrel AI. A successful verification further reinforces the correctness of the testing implementation, and more importantly, validates that the squirrel is correctly implemented. Additionally, this thorough testing will allow us to gather statistics on the time and memory usage of the verification

approach. Finally, we apply the verification approach to the Halo AI. This effort is exploratory in nature, rather than exhaustive, primarily focussing on the question of scalability while demonstrating that the system can successfully detect previously unknown errors.

Generation of Promela code was done through Scythe AI. The Squirrel AI Promela code is given in its entirety as Appendix B. By having push-button generation of code, it was straightforward to iterate on the Promela representation until it could generate correct code. Promela code was built from a parse of each statechart to get the transition and event structure, augmented with information from the module interface to determine external events. The generated code was faithful to the representation presented in this chapter. Code generation itself was very fast, taking only a second or two to complete.

All tests were done using iSpin version 1.1.0, backed by Spin 6.2.2. The testing machine was a quad-core Intel i5 CPU running at 2.67GHz with 4.0GB of RAM, running Windows 7. All running times are as reported by Spin. The running times do not include compilation of the Spin libraries, which adds a few seconds to the length of each verification pass.

8.4.1 Verifying Statechart Representation and Generation

Before verifying the correctness of an AI, we need to establish the correctness of the testing framework and debug as necessary. This means we must first ensure that the code generation process correctly transforms SCXML to Promela, and that the Promela statechart representation (e.g., event generation, passing events) is correct. We did this using the Squirrel AI as our initial test-bed. It is an

important first step to verify a known working model, as it allowed verification of properties of which the result was already known. Doing these detected several bugs in features that were known to be working properly, thus informing us of several bugs in the generation of the Promela model.

An example of such an error in Promela code generation was in the flattening of history states. When attempting to verify flee behaviour, error traces indicated that on-exit actions from a state transitioning to a history state were not being correctly added to the triggering transition. Similarly, transitions from a single state with duplicate names but mutually exclusive guards were being incorrectly filtered as non-deterministic duplicate events. Both of these bugs were corrected.

After several testing and debug passes, the Promela model was free of unexpected results. While this does not guarantee that the representation is fully correct, it provides confidence as we move on to the complete validation of squirrel behaviours. Since the squirrel AI is a known good AI, and since it uses all features of the statechart formalism (including history states, intra-level transitions, and orthogonal regions), a successful complete verification will help to further reinforce the correctness of the generation process and statechart representation.

Optimizing Event Delivery

During the initial testing phase, an important optimization was found. Initially, a single start flag was used by the event delivery process to activate all statecharts, rather than the earlier described approach of having a start flag for each statechart. Using a single start flag causes Spin to examine each different ordering of event delivery to all statecharts. Since the number of orderings grows

exponentially with the number of statecharts, this alone would prevent meaningful verification of a large AI.

We overcome this by enforcing an ordering on event delivery. The first statechart processes an event, with the next statechart only beginning after the first statechart has completed. By enforcing this ordering, the same reachability verification on the complete squirrel AI went from taking 52.9s to 0.017s, an improvement of four orders of magnitude. Certainly, this optimization is necessary for larger AIs such as the Halo AI.

Theoretically speaking, this optimization is acceptable. Neither the Rhapsody semantics nor our description of cooperating statecharts place any requirements on the order in which a single event is distributed to orthogonal regions or separate statecharts. Enforcing an arbitrary ordering is thus valid, though it constrains implementation, as the ordering of event delivery in the implementation must match the verified ordering for the verification results to be applicable.

8.4.2 Complete Verification of the Squirrel AI

With a reasonable assurance that the verification framework is working correctly, the next step is to verify it on a known-good AI. This accomplishes three major things. First, a successful verification with no unexpected results greatly increases confidence that the verification framework is correctly implemented. Secondly, it allows us to establish baselines for time and memory usage of various model-checking steps. Finally, it proves that the extensively tested and debugged squirrel AI is in fact correct with respect to the design goals in all cases, and that no undiscovered corner cases exist.

```
Move_Actuator_destination_reached
Move_Actuator_destination_unreachable
Move_Actuator_collided
Pickup_Actuator_pick_up_successful
Pickup_Actuator_pick_up_failed
Threat_Analyzer_i_see_player
Threat_Analyzer_i_dont_see_player
Threat_Analyzer_threat_changed
Wander_Executor_time
Eat_Decider_key_item_visible
Eat_Decider_no_key_item_visible
Energy_Sensor_energy_changed
Item_Memorizer_i_see_item
Item_Memorizer_i_dont_see_item
```

Figure 8–7: External Events in the Squirrel AI

The environment for the squirrel is normally the Mammoth game world. However, the view of the environment for the squirrel consists mostly of sensed events. In addition, some statecharts will generate events from their associated classes based upon game state. A list of these events is given in Fig. 8–7. Most are prefixed by the name of the statechart that receives the event.

Reachability

The first property verified on the squirrel AI was reachability, which consisted of verifying that every state in every state chart was reachable. Specifications were given as global negatives (as described under Reachability in §8.3.2) so that cycles were avoided. The possible events were those listed above in Fig. 8–7.

The entire AI has only 38 states total. Twelve of these are initial states, leaving 26 states to be verified as reachable. All states were found to be reachable, with the exception of parent states that possessed default transitions. Due to the

flattening of the state hierarchy, no transitions should target such states, making this result the expected one.

On average, the time needed to verify the reachability of a single state was only 10^{-2}s. This tells us that an automated process could verify reachability for the entire AI in much less than a second. This is fast enough that it could be easily automated, allowing for use within Scythe AI as a background check, with results being reported in the errors and warnings bar. Certainly, a formal verification step in such an application could very reasonably include a reachability pre-pass at very little cost.

Such a straightforward verification is useful, as it allows us to guarantee that all behaviours in the squirrel AI can be fully expressed. This type of guarantee is especially valuable for complex modular AIs, where different combinations of modules may never present the correct chain of events that would allow a given module to enter every state.

Squirrel Specifications

Verifying behaviour means answering the following question: given an AI and a set of external events, does the behaviour of the AI satisfy the design goals? A proper verification will discover any violations of the design goals. This does not provide any universal guarantee of correctness - indeed, errors in the associated classes, incorrect mappings between game states and generated events, or even poor design goals could easily result in a non-functional AI. However, a behavioural verification will tell us that the AI logic correctly implements the design goals.

At this point, we revisit the design goals of the squirrel AI, first introduced in §4.1.1. Each of these goals breaks down into a small number of specifications, as shown in Table 8–1. The purpose of this step is to have specifications that can easily be translated into LTL. Some ambiguity at this stage is permissible, so long as the design goal and the specification together are clear.

This first set of specifications tells us that the decision making of the squirrel is correct. A complete verification requires that we also test for correct execution of squirrel decisions. That is, when the squirrel chooses a high level goal, the expected actuation will occur. These tests are given in Table 8–2.

Verifying Brain-to-Actuation functionality implicitly tests the functionality of intermediary statecharts. If, for example, we verify that moving will occur when the brain chooses to wander, this confirms that the brain to wander executor connection is working correctly and that the executor to actuator functionality is also correct. Similar functionality arguments apply to the other brain goals. Using the same argument, testing correctness in the sensing apparatus is included when doing verifications of specifications going from event to actuator (e.g., when the squirrel is hungry, a specific action occurs).

The full set of specifications from Tables 8–1 and 8–2 will be formalized in LTL, then validated. Together, these specifications are sufficient to validate that the squirrel AI correctly implements the design goals. We reason this as follows: specifications (i) to (x) tell us that when a condition is met that matches a design goal, the AI makes the correct decision, while specifications (xi) to (xiv) tell that when a decision is made, the correct output results. We make no claims as to the

Table 8–1: Squirrel Design Goals and Resulting Specifications

Design Goal	**Specifications**
Squirrels have a low and high threat radius used to determine if a player character is dangerously close to the squirrel.	{Implementation detail, nothing to verify}
Squirrels will prioritize fleeing from high threats at all times.	(*i*) Globally, high threat event causes fleeing.
If there are only low threats, squirrels will flee from them except in the case where they are very hungry.	(*ii*) If there are only low threats and the squirrel is not very hungry the squirrel will flee. (*iii*) If there is a low threat while and the squirrel is very hungry, the squirrel will continue collecting food. (*iv*) When a very hungry squirrel collects food under low threat, it will eat immediately.
Under no threat, when a squirrel is hungry it will gather one of the visible acorns. *[These specifications are prefixed with "When a squirrel is hungry and under no theat"]*	(*v*) ..., and if the squirrel does not see an acorn, it will wander until it sees an acorn. (*vi*) ..., and the squirrel first sees an acorn, and it is not carrying one, it will move to the acorn. (*vii*) ..., and it moves to an acorn, it will try to pick it up.
Whenever a squirrel is very hungry, it will eat the acorn it is carrying.	(*viii*) When very hungry and has an acorn, the squirrel will eat.
When a squirrel is neither very hungry nor threatened, it will wander.	(*ix*) When no threat and not hungry, the squirrel will wander. (*x*) When a squirrel picks up an acorn, it will resume wandering until threat or very hungry.

Table 8–2: Squirrel Implementation and Specifications

Implementation Functionality	**Specifications**
Brain to Actuator Functionality (Goal Verification)	(*xi*) When a squirrel decides to wander, it will move before leaving the wander state. (*xii*) When a squirrel decides to flee it will move before leaving the flee state. (*xiii*) When a squirrel decides to eat it will eat before leaving the eat state.
Pickup Functionality	(*xiv*) When a squirrel decides to pick up an item, it will either move to the item, or pick it up. [we cannot guarantee a successful pickup - another squirrel may grab the item]

correctness of the squirrel in unspecified scenarios, though this set of design goals does attempt to fully specify the squirrel behaviour. That there is no way in which to empirically validate the completeness of a set of specification is a shortcoming of any specification approach.

Formalising and Testing Specifications

Each of the specifications from Tables 8–1 and 8–2 must now be formalised and verified. Here, we will document the validation of two representative specifications. This illustrates the process in sufficient depth as to clearly communicate the process and allow for results to be reproduced. The remainder of the specifications have their results summarized at the end of this subsection.

Verifying Flee Functionality: Here, we will verify specification (*ii*): "When a squirrel decides to flee it will move before leaving the flee state." The formal specification used was the one developed in §8.3.2.

Initial attempts at verification failed due to spurious error. In any state, the external generator is able to create an `i_see_item` event. Investigation revealed that the event is only consumed by the Key Item Memorizer, and that consuming this event does not change state or create a new event. By generating this event repeatedly, the AI will live-lock and Spin is unable to validate the claim. The events for spotting and losing key items are modelled as external events and so we can entirely remove the Key Item Memorizer from the model for verification purposes. This prevents the occurrence of this live-lock, without any loss in capability of the Promela abstraction

With that change made, the verification was able to complete. The elapsed time was 7.91 seconds, while the search reached a depth of 1121574 states. No errors were discovered, thus proving that whenever the Brain makes the decision to flee, movement will always result.

External Events and Cycles

External events, such as the `i_see_item` that was just discussed, present special issues when evaluating temporal claims. Specifically, claims using operators such as Eventually Ψ or Until Ψ must hold true on all future paths from that state. If there is a path where the temporal condition has not yet occurred, and that path is a cycle, then that path is a counter-example and will halt verification. The condition will never occur, since the path will prevent progress indefinitely. External events can easily lead to such cycles.

As an example, imagine a specification stating that when the squirrel tries to flee, the external event `move_succeeded` or `move_failed` will eventually be

generated while the squirrel brain is still in the flee state. To satisfy this, external event generation must eventually generate one of those two events. The problem arises when the external event generator can infinitely generate another series of events. For instance, a `high_threat` event can be generated at any point. This will change the state of the Threat Analyzer, but will not affect the brain since the squirrel is already fleeing. Next, the external generator could create a `low_threat` event. By repeating these events, a path is found whereby the eventually claim is never satisfied and so verification fails.

Of course, such a cycle is spurious. Under no normal game condition would an NPC be infinitely alternated between low and high threat on every frame. However, the simplicity of the external event generator lacks the precision to describe such scenarios. Instead, we must work around this through clever specification.

A similar but related type of failure with respect to the model is the lack of logical connection between events. For example, an eat event always succeeds in Mammoth, and causes the squirrel to return to full energy. This would be sensed by the Energy Sensor, transitioning it into the high energy state. Thus, it seem reasonable to specify that when the squirrel eats, it is returned to full energy. However, since there is no representation of the connection between eating and energy levels, attempting to verify this would fail. Furthermore, this type of specification is unacceptable for another reason: it describes how the game should react to an event, rather than how the AI logic should react. Thus, it is out of scope. We point this out to clarify how scoping impacts specification.

Three main tactics were employed to handle these types of error:

- Single External Event Specifications: An interesting external event causes a state change in the AI (e.g., the Threat Analyzer moves from no threat to low threat), which in turn may generate additional events. Any future claims in the specification should describe a state that must be reached by the time the AI event queue has cleared, and not extend to how it will respond to future external events. This avoids spurious external cycles.
- External Tagging: The Eat Actuator is a single state - how can we tell that it has transitioned? Noting that an eat event has been generated is insufficient, since it may not trigger a transition in the actuator. Instead, we add a custom flag `eatFlag`, that is set to true when the transition occurs, and returned to false when the AI event queue is clear. This allows us to specify conditions that involve the act of eating (as in specifications $iv, viii, xiii$).
- Capturing State changes: To ensure single external event specifications, the LTL Next operator was heavily employed. This was used to capture the exact event that caused an interesting transition, and limited the specification to just that external event. For example, capturing the change in not getting food to getting food can be done with $(EatDecider.state != getting\_food) \wedge (XEatDecider.state == getting\_food)$. Using this as the left hand side in an implication correctly limits the scope to the exact moment the squirrel decides to get food.

Verifying Acorn Collection: Each behaviour is intertwined with all other behaviours that form the complete NPC logic. Specifying and validating

each one is a process of discovering the requirements that are higher priority and eliminating them, such that the exact specification can be tested. We demonstrate this active specification process by validating specification (*vii*): when a hungry squirrel moves to an acorn, it will try to pick it up.

We began verification by creating a specification where the squirrel should always pick up an acorn when hungry. This forms a starting point that will be iterated upon. We specify that food collection begins with the Brain entering either food collection state. This was specified as:

$$\Box(((Brain.state == searching) \vee (Brain.state == starving) \rightarrow$$
$$(\diamond(PickupActuator.state == picking\_up)))$$

This specification found an acceptable failure. If there are no acorns present, the intention is for the squirrel to simply wander until an acorn is spotted. Verification failed due to this, as it found live-lock failures where the Eat Decider was stuck in the wander state and no key item spotted events were generated. This is a shortcoming of the specification, and so it was modified to the following:

$$\Box(((Brain.state == searching) \vee (Brain.state == starving) \rightarrow$$
$$((\diamond((EatDecider.state! = getting\_food) \wedge (XEatDecider.state == getting\_food)))$$
$$\rightarrow \diamond(PickupActuator.state == picking\_up))))$$

Model checking this specification also produced a failure due to another cycle. In the Pickup Executor, we see that if an item is out of pick up range, a move command will be issued that moves the squirrel into range. Since moving into range is not guaranteed, verification finds another cycle at this point. We avoid

this by limiting the verification to states where the item is in range by specifying this in the Pickup Executor instead of the Eat Decider:

$$\Box(((Brain.state == searching) \vee (Brain.state == starving)) \wedge$$
$$((PickupExecutor.state == idle) \wedge (XPickupExecutor.state == executing) \rightarrow$$
$$\diamond (PickupActuator.state == picking\_up))))$$

By including the next operator, no external cycles are allowed when the PickUp Executor begins. However, once the executing state is reached, an eventless transition is encountered, which requires an external event and thus allows for cycles. To proceed, we must eliminate the cycle by collapsing the Pickup Executor into two states with no eventless transitions:

$$\Box(((Brain.state == searching) \vee (Brain.state == starving)) \wedge$$
$$((PickupExecutor.state == executing) \wedge (XPickupExecutor.state == idle) \rightarrow$$
$$\diamond ((PickupActuator.state == picking\_up) \vee (moveFailedFlag))))$$

This specification was successful. The move failed flag is a necessary inclusion, since the collapsed PickUp Executor makes that same transition from executing to idle on a failed move. This specification thus captures the scenario where a hungry squirrel moves to an acorn, it will try to pick it up, or it could not reach it. Since verifying this also includes verification of every path where the move succeeds, success here means that *vii* holds.

Squirrel Verification Results

The entire set of formal Promela specifications is given in Table 8–3. It follows standard programming language notation, where 'and', 'or', and 'not' are

represented as `&&`, `||`, and `!`. The LTL terms are `[]` for global, `<>` for eventually, `U` for until, and `X` is the unary operator for next. Other operators were not used. The states referenced refer to the Promela encoding of the statecharts. The complete listing of the Promela statecharts is given in full in Appendix B, and can be used as a cross-reference for the state numbering.

The most notable flag that was added was the quiescent flag. At the start of an external event generation the quiescent flag was set to false. It was only set to true when the AI queue became empty. It was used in several specifications, usually in the form `(!quiescent U state)`. This means that quiescence cannot be reached without `state` having become true.

The resources used in verifying each specification are given in Table 8–4. All specifications required the same search depth of 1121574 states. This is logical, since all specifications are global and thus demand a full exploration of the statespace. The varying time and memory usage results from the different amount of backtracking and path exploration required to evaluate the variety of temporal conditions. Repeated verification of a single specifications used identical amounts of memory, while the time only changed by at most 0.01s. Because of that stability, the numbers listed are the exact values in the output from Spin for a single representative run of each specification.

Table 8–3: Squirrel Verification: Promela Specifications

(i)	`[]( (Threat_AnalyzerState==3) -> (<>(Squirrel_BrainState==5)))`
(ii)	`[]((((Squirrel_BrainState!=4) && (Squirrel_BrainState!=5)) && (X(Squirrel_BrainState==5))) -> ((X(Squirrel_BrainState==5)) U (Move_ActuatorState==1)))`
(iii)	`[]((((Squirrel_BrainState==4) && (Threat_AnalyzerState==1)) && (X(Threat_AnalyzerState==2))) -> ((Squirrel_BrainState==4) U (quiescent)))`
(iv)	`[](((Squirrel_BrainState==4) && (Eat_AnalyzerState==2) && (!pickUpFlag) && (X(pickUpFlag))) -> (<>(eatFlag)))`
(v)	`[](((Squirrel_BrainState==2) && (Eat_DeciderState==0) && (Energy_SensorState==0) && (X(Energy_SensorState==1))) -> (<>(Eat_DeciderState==2)))`
(vi)	`[](((Energy_SensorState!=0) && (Eat_DeciderState==2) && (!foodSpottedFlag) && (X(foodSpottedFlag))) -> ((!(X(quiescent)))U((Pickup_ActuatorState==1) || (Pickup_ExecutorState==1))))`
(vii)	`[]((((Squirrel_BrainState==3) || (Squirrel_BrainState==4)) && ((Pickup_ExecutorState==1) && (X(Pickup_ExecutorState==0)))) -> (<>((Pickup_ActuatorState==1) || moveFailedFlag)))`
$(viii)$	`[](((  Energy_SensorState!=2) && (X(Energy_SensorState==2)) && (Eat_AnalyzerState==1)) -> (<>(eatFlag)))`
(ix)	`[](((Energy_SensorState==0) && (Threat_AnalyzerState==1)) -> ((!quiescent) U (Wander_ExecutorState!=1)))`
(x)	`[]((pickUpFlag && (Threat_AnalyzerState==1) && (Energy_SensorState==1)) ->(<>(Wander_ExecutorState!=1)))`
(xi)	`[](((Wander_ExecutorState==0) && (X(Wander_ExecutorState!=0))) -> ((!quiescent && (X(Wander_ExecutorState!=0))) U (Move_ActuatorState!=0)))`
(xii)	`[](((Flee_DeciderState<4) && (X(Flee_DeciderState>=4))) -> ((!quiescent && (X(Flee_DeciderState>=4))) U (Move_ActuatorState!=0)))`
$(xiii)$	`[]((((Eat_AnalyzerState==1) || (Eat_AnalyzerState==2)) && (X(Eat_AnalyzerState==0))) -> ((!quiescent)U(eatFlag)))`
(xiv)	`[](((Pickup_ExecutorState==0) && (Eat_DeciderState!=3) && (X(Eat_DeciderState==3))) -> (<>((Move_ActuatorState==1) || (Pickup_ActuatorState==1))))`

Table 8–4: Squirrel Verification: Resource Usage

Specification	Time (s)	Memory (MB)
(i)	5.67	820.067
(ii)	7.91	1115.087
(iii)	6.05	914.305
(iv)	5.80	889.696
(v)	5.27	824.559
(vi)	5.56	851.219
(vii)	5.60	865.087
$(viii)$	6.32	951.317
(ix)	5.11	793.504
(x)	5.10	793.504
(xi)	16.60	1274.462
(xii)	12.20	1416.258
$(xiii)$	8.31	1215.965
(xiv)	6.06	919.383

8.4.3 Verifying the Halo AI

In this subsection we address verification of the more complex Halo AI. While the investigation follows the same pattern as the squirrel AI, the correctness of the Halo AI is not known *a priori*, and thus verification takes on a more exploratory nature. Since the Halo AI is a large AI, similar in size to an industrial AI, verification tractability is of particular relevance.

Reachability

The first query tested was that the AI could move, specified as:

$$\Box!(Run\_Actuator.state == moving)$$

The verification took 0.1s, an order of magnitude greater than the squirrel, but still fast enough to be practical. This behaviour is in fact the initial behaviour,

meaning that this behaviour is reached solely through the on-entry events of default states. Since this occurs before a single external event is generated, the search depth is limited.

To perform a more extensive test, the reachability of the reposition state in the Ranged Combat Executor was verified. This state was chosen due to the fact that reaching it requires the generation of several different external events in a specific order. This forces Spin to perform many external event generations, necessitating a much deeper search. The verification took 48.9s and required 7476 MB of memory. Spin found no violations (i.e., the state was not reached), but came up against the maximum search depth.

Technically speaking, this result is inconclusive. Hitting the maximum search depth means that there exist deeper paths that Spin could not evaluate due to memory constraints, and thus the state space was only partially explored. It is possible that a violation could occur in the unexplored region. Spin can compress state data; doing so here increased the search depth and reduced memory usage to 1616 MB, but increased search time by an order of magnitude. While that verification also found no violations, Spin again hit the search depth bound of 1×10^7 states. Attempts to further increase the search depth failed due to the lack of additional physical memory even with compression active.

There are two possibilities at this point: either the state is in fact unreachable and there is an error in the statechart logic, or any paths that reach the state are below the depth limit. A simple examination of the statecharts tells us that the state should be reachable with transitions numbering in the tens, rather than

hundreds or thousands. It is reasonable to expect that with a search depth of 1×10^7 states, the state should have been reached if the AI logic was correct. Thus, we continue based on the assumption that the state is unreachable due to an error in the statecharts.

Our next step was to backtrack to earlier states in the Ranged Combat Executor. These states also produced inconclusive results. The Combat Decider should have been triggering transitions that would move the Ranged Combat Executor to the reposition state. Working through ranged combat states in the Combat Decider yielded similar inconclusive results. Backtracking again, we examined the Weapon Sensor. It turned out that our assumption was correct, and an error was preventing the occurrence of ranged combat behaviours. While this statechart produces the `ev_RangedWeaponEquipped` event as an output (thus satisfying syntactic correctness), an omitted transition prevented it from actually being generated. Since no ranged weapons were known to be equipped, all ranged combat behaviours were prevented. The transition was added, allowing for generation of the event.

With the error corrected, reachability verification was resumed. The Combat Executor was now able to reached the ranged combat state, which was verified in just 0.004s. Next, the reposition state in the Ranged Combat executor was verified as reachable. However, due to the fact that it requires a very specific chain of external events to enact the behaviour, Spin took 36.5s to establish the violation of our negative specification with a search depth of 6247272 states.

Based on memory requirements, finding this violation was near the limit of what is achievable on the test computer. In such resource-limited environments, this test case is an interesting exploration of how even inconclusive results can be used to infer the presence of errors. It also demonstrates the limits: states that are only reached when the Ranged Combat Executor is repositioning would not be verifiable as reachable due to memory limits.

Behavioural Verifications

Similar to the Squirrel AI, we wish to demonstrate how behavioural verifications might be performed on the Halo AI. One such verification would be that when the Brain decides to engage with an enemy, it always results in a melee or ranged attack. This was expressed as:

$$\Box((\neg(Brain.state = combat) \wedge (XBrain.state = combat)) \rightarrow$$
$$((Brain.state = combat)U(\diamond(MeleeActuator.state = swinging)$$
$$\vee \diamond(RangedCombatExecutor.R0.state = firing))))$$

In general, verifying liveness specifications is a more resource intensive process. Attempts to validate this specification failed due to the search depth limit being reached. Spin offers memory compression; activating this allowed us to increase the search depth to 1.5×10^7, but dramatically increased the already expensive computation time. Spin was able to verify the specification up to the depth limit, using only 2GB of RAM due to the compression. Unfortunately, this process took a full 3620s (just over 1 hour). Because the depth limit was reached, we can conclude that any violations of the claim must be deeper than

the depth limit, and that no shallow violations are possible. However, the overall specification is intractable given the resource limitations.

Establishing the overall behavioural correctness of the Halo AI might require dozens or hundreds of such verifications, which would demand large amounts of time and effort. This suggests that it would be more worthwhile to focus on critical behaviours. For example, if a computer with a large amount of memory (e.g., 16 or 32GB of RAM) could fully evaluate claims within an hour, then it would be possible to validate several critical behaviours as an overnight process. Unfortunately, it is impossible to predict what the search depth is for a given AI, and therefore the memory requirements are also unknown. Spin must actively evaluate them and explore them in order to build the search space, and so the memory required can only be discovered in an environment that already has sufficient memory.

8.5 Conclusions

Syntactic correctness is a simple and vital form of verification for layered statechart-based AI. The simplicity of the approach, coupled with the speed and usefulness of the results, make it easy to incorporate into a tool. Indeed, it has already been added to Scythe AI. It yielded immediate results, and helped in the creation of both the squirrel and the Halo AIs. We speculate that as reuse increases, syntactic correctness validation will increase in value. A larger scale evaluation of modular reuse methodology would provide an opportunity to verify this claim.

Establishing semantic correctness is a vital and valuable part of the development process for layered statechart-based AI. Here, we present the beginnings of an approach to systematic evaluation of semantic correctness using model-checking. By providing an automated tool to export a Promela representation of an AI built in Scythe AI, we demonstrate a practical workflow for verification.

Unfortunately, establishing semantic correctness is a complicated proposition. Developing correct specifications is a difficult task, and the results of any validation are only as useful as the correctness of the provided specification. Indeed, making a good specification requires an intimate knowledge of the AI being tested, as well as the formalism itself. Raw error traces from Spin can be thousands of lines long, and since the search is depth first, frequently include irrelevant operations that have no bearing on the violation discovered. Identifying the source of a violation is thus a challenge in itself.

To evaluate the verification process, the squirrel AI was fully verified against the design requirements of the NPC. Here, all states in the statechart-based AI were found to be reachable, and all specifications were verified to hold. Reachability took essentially no time to verify, while the specifications for semantic correctness took a mean of 7.3s and required 974.6 MB of memory. This shows that for small AIs, verification tractability is manageable. Verification attempts on the Halo AI, however, make it clear that combinatorial explosion is a concern for larger AIs.

While the research here has a tantalizing promise, much more work is required to fully explore the process. As next steps in this area of research, we propose the following:

- Tractability: This issue is of primary importance. Further research could find exactly where the cutoff occurs between intractable and tractable, given the system resources. This would help in both the design and development stage (e.g., if an AI exceeds that size, it will no longer be verifiable.)
- Specification Coverage: In the case of the squirrel AI, we manually created the specifications. Of course, a process such as this allows for the introduction of errors due to overlooked subtleties, inadvertent duplication, and so forth. Further research could explore how exactly one might systematically generate a complete set of specifications.
- Complexity Management: A fully workable process should provide techniques to work with large AIs. One possibility that could be explored would be creating functional groups and treating them as black boxes. This could dramatically reduce the state space. The functional group itself could then be easily verified for internal correctness, so long as it was not much larger than the squirrel AI.

Even considering the difficulties of the verification process, we still believe that model-checking is an appropriate solution for this problem. We have demonstrated here that it can be used to verify the correctness of AI for simple NPCs. To make it practical for game developers, there needs to be an automated way to generate and verify basic query types, such as reachability and intent-action

behaviours demonstrated here. Error traces need to be heavily processed such that only relevant information is presented.

Looking forward, we believe that specifications such as these can be used as unit tests for AI logic. The workflow could be streamlined by encoding only the most important specifications in Promela, and then running these on a regular basis over the development process as unit tests. This provides a method to ensure that working behaviours are not accidentally broken, without having to extensively test the AI manually. A cost/benefit analysis of this technique could be performed by the quality assurance team of a game development studio, where it could be extensively evaluated versus their prior knowledge of the effort involved in validating AI logic.

CHAPTER 9
Generating NPC Populations

Crafting well-designed AI for NPCs in a digital game can take significant development effort. Because of this, games requiring large numbers of NPCs put AI developers in a difficult position. Having a large number of interesting and varied characters increases realism and thus immersion [50], yet producing such a population is cost-prohibitive. For this reason, game developers often reuse the same AI with small changes in timings or probabilities to provide some variation without incurring high development costs.

We seek to provide developers with a third option, by giving them the ability to create a variety of high quality game AI without significant development overhead. To this end, we have developed a procedural content generation approach that creates highly individualized NPCs. Working from our layered-Statechart representation of an AI agent, our approach varies AI at three different levels. Simple modifications are done with parameter-based variation, compositional modification adds and remove behaviours, and more complex structure-based approaches alter behaviours using rule-based model transformations. These techniques take advantage of the formalism's inherent modularity, allowing for easy perturbation of AI designs at different levels of abstraction. With a concrete set of transformations, generation is quick, allowing for rapid prototyping and deployment.

In this chapter, we explain how a layered statechart-based AI can be varied at the parameter level, the module composition level, and the statechart structure level. The three variation techniques are then combined to form an overall content generation strategy. We apply this to the squirrel AI and statistically validate the generated variations.

The chapter concludes with an investigation into the generation of populations that satisfy certain design metrics. Specifically, we look at game difficulty. Early attempts to increase difficulty through NPC variation were limited to simple parameter modification: a faster NPC is harder to shoot, and an NPC with more hit points is harder to kill. Modern games improve on this by giving new actions or abilities to harder versions, yet the decision making process of the AI is relatively unchanged. By allowing changes at all levels of behaviour, our approach allows for changes that are simple or complex, allowing developers to create a mix of NPCs that meets the requirements of their game.

9.1 Content Generation

The content generation process requires a seed AI to act as the basis for generation. As in previous chapters, we will use the Squirrel AI described in Appendix A. In Mammoth, the squirrel is a background character, giving us considerable freedom to modify how behaviours are expressed. Additionally, the squirrel employs a number of separate behaviours with modifiable parameters, providing an excellent test case for our variation techniques.

9.1.1 Varying Parameter Values

Non-player characters have many properties that are set at instantiation, such as maximum health, movement speed, and so on. As a part of a larger content generation system, parameter modification is a simple yet valuable tool, since it can create differentiated instances of an NPC. While the approach is conceptually straightforward, modification of parameters of a finely tuned AI can lead to incorrect behaviour, and thus several guidelines need to be followed to ensure correctness of the resulting NPCs.

Assigning a new value to numeric parameters requires a range of acceptable values appropriate to the game context. This range derives from the expected capabilities of the NPC. For instance the movement speed of a squirrel must always be positive and should be faster than the average movement speed of a human. Selecting a value from within the chosen range can be done randomly through the use of a distribution function such as a Gaussian.

Parameters can be interdependent, meaning that a logical relationship exists between parameters. An example would be run and walk speed, where the run speed logically should be greater than the walk speed. A subset of these are *critical dependencies*, which are logical dependencies between parameters where violation of the logical dependency will result in behavioural errors. In the squirrel AI, there are separate values for the radius of low and high threat zones, with high threat taking precedence over low threat. If the high threat radius is greater than the low threat, behaviours resulting from low threats would effectively be removed. To prevent violations of critical dependencies, we employ *dynamic ranges*, where

the range of one parameter is based upon the value chosen for another parameter. Applied to our threat radii, if the range for the high threat radius is $[1, 5]$, the range for low threat could be $[1 + highThreshold, 10]$. By definition, non-critical dependencies can be ignored, though respecting such a relationship could give better results.

Dynamic ranges introduce an ordering problem in parameter selection. The generation of a value with a dynamic range must be preceded by the generation of the values upon which it relies. In the case of a cyclical dependency, generation should fail with a warning. Additionally, there exists the possibility of an impossible range, where $[x, y]$ is a dynamic range, and $x > y$. This too should cause generation to fail. The entire process is given in our parameter modification strategy, presented in Fig. 9–1.

9.1.2 Varying Module Configurations

The layered Statechart formalism is inherently modular, allowing AI modification by adding, removing, or swapping modules. Respectively, these changes allow the process to introduce new behaviours, excise unwanted behaviours, and replace existing behaviours.

To automate the process, modules part of the base AI can be tagged as removable or swappable, while available modules not part of the base AI can be tagged as addable. These tags are defined by the user when a variation is configured.

```
I. For each parameter, determine if it has any dependencies.
   if no dependencies:
   i. Define a range for the parameter.
   ii. Decide on the probability distribution best suited
       for that parameter.
II. For parameter found to have a dependency:
   i. Determine if the dependency is critical.
   ii. If critical, assign dynamic ranges to resolve
       problem.
   iii. If non-critical, resolve if desired, perhaps using
       dynamic ranges.
III. Generate values.
   i. If error due to cyclical dependency or invalid range,
    fail with a warning. User should resolve the problem
    manually and restart.
   ii. (Optional) Regenerate values for statistical
      outliers.
```

Figure 9–1: Parameter Modification Strategy

Removing Modules

Removing behaviours is a straightforward approach to generating a variation of an NPCs. This approach has been effectively employed in notable industrial titles, such as behaviour masks in the game Halo 3 [16], where "brave" enemies were made by removing the ability to flee.

Removal in the layered statechart approach consists of removing a subset of AI modules that are tagged as removable. In general, candidates for removal should be selected from analyzers, coordinators, 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 clumsy

or jittery. Removing deciders would take away the ability for an AI to perform one of its high-level goals. For example, removing the *FleeDecider* from a squirrel would generate a "brave" squirrel with no fear of humans. Removing an executor results in an AI that has difficulty in carrying out a tactical plan, though it may be possible to partially complete a task. Removing sensors or actuators limits how the NPC can sense and interact with the game, and could create interesting variations, such as a solipsistic AI with no sensors.

The strategizers are the most essential components and should never be removed. Most AIs have only one *Brain* component, the removal of which would sever the connection between the input and the output of the AI and yield an NPC that performs no actions whatsoever.

Replacing Modules

An effective way to generate new configurations from an existing AI is to replace components by other equivalent components. A new module is considered a match to an existing module so long as their AI module interface uses the same events and synchronous calls. A typical use-case would be to create an alternate version of a key component, such as a *BrainAlternate* module that uses a different high-level strategy. If the original Brain is tagged as swappable, then generation would then select between the original and alternate when producing a member of the population. This result is similar to branch replacement in behaviour trees [45].

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 a developer to specify that specific events of specific components are renamed, which makes it possible to integrate the *ExplorationPlanner* into the squirrel AI as a replacement for the *WanderPlanner*.

Adding Modules

New AI behaviours can be added by introducing new components to an existing configuration. Adding a new sensor would augment the ways the AI can perceive the game state. For example, *Ears* could allow a squirrel to detect approaching game entities even in the dark. By examining the interfaces of higher level Statecharts, the new sensor can create appropriately named events that would be received. New analyzers can help the AI in detecting high-level events based upon correlated occurrences of low-level events.

Modules belonging to any category between the strategic deciders and the actuators are more difficult to add. Adding a new actuator component, for example, is pointless without also adding or swapping for an executor that generates input events for the new actuator.

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. This allows event flow to be diverted to another statechart, which then executes and returns the originally expected event, making the addition invisible

to the originally connected statecharts. The *StutterExecutor* component shown in Fig. 9–2 is an example of an executor component that, when asked to *stutterMove* to a given position, moves towards the destination for a moment, waits some time, then continues moving, then waits again, and so on. 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 modules that produce *moveTo* events: the *WanderExecutor*, the *PickupExecutor* and the *FleeDecider*. Event interception makes it possible to transform the *moveTo* events generated by the *WanderExecutor* to *stutterMove* events. As a result, our squirrel moves intermittently while exploring, but does not stutter when picking up an acorn or when fleeing from a threat.

9.1.3 Varying Statechart Models

The most extreme variations come from arbitrary structural modifications to the statecharts forming the AI. This has the effect of modifying the AI logic itself, since the statechart structure implements behavioural logic. We explicitly model these modifications 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* [41], the transformation unit is a rule, which uses model patterns as pre- and post-conditions. The pre-condition pattern determines the applicability of a rule, here described with a Left-Hand Side (LHS) and optional Negative Application Conditions (NACs). The LHS defines the

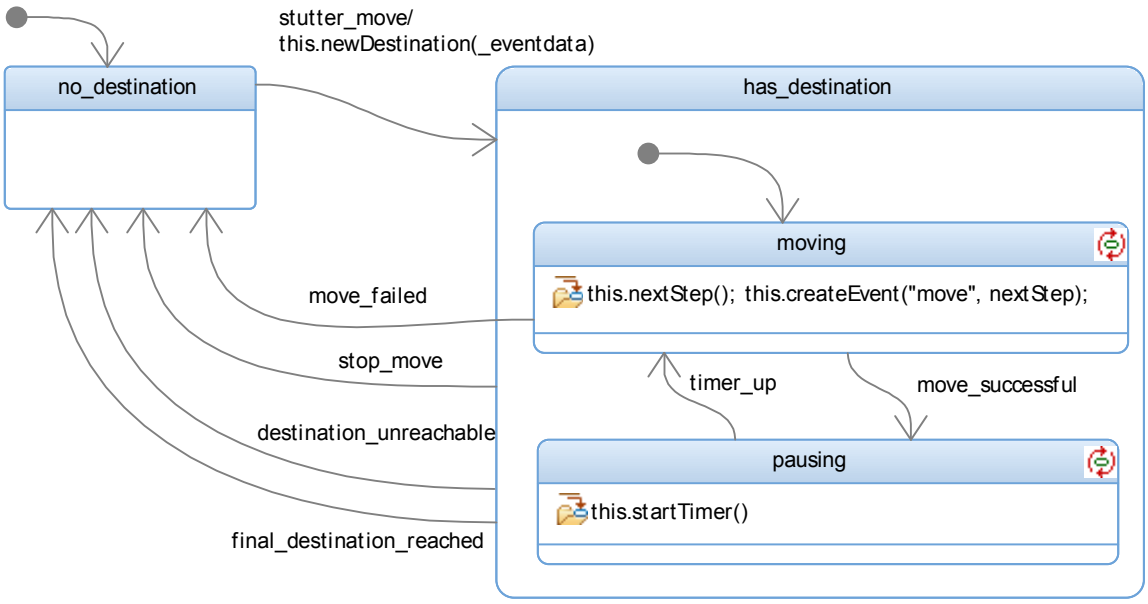


Figure 9–2: Stutter Executor for the Squirrel

pattern that must be found in the input model to apply the rule while the NAC defines a pattern that shall not be present. The Right-Hand Side (RHS) imposes the post-condition pattern to be found after the rule was applied, *i.e.*, the effect.

A key 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.

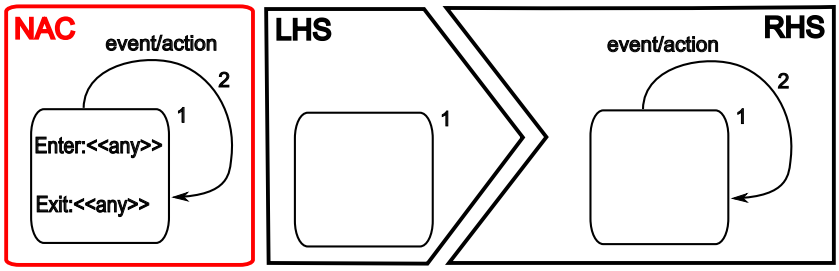
Orthogonal Behaviours

Safe variations are ensured by introducing only *orthogonal* behaviours. This type of behaviour is one that has no effect on the core functionality of the

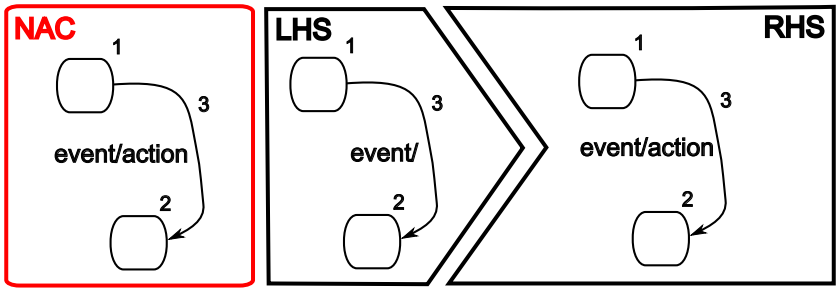
NPC, and thus, orthogonality is a highly game-dependent property. Playing an animation may be an orthogonal property in one game, while another may use animation-aware collision detection. Having the NPC make a noise could possibly attract enemies, or in another game sound could have no game play impact. Wearing a hat may be orthogonal in one game context, while another game may have magic helmets that affect the game state. Necessarily, defining a behaviour as orthogonal is highly game specific and requires detailed knowledge of the game context.

Figures 9–3(a) and 9–3(b) illustrate how rule-based model transformations can be used to add orthogonal behaviours to existing state charts. The rule in Fig. 9–3(a) causes an event-triggered action to be added to an arbitrary state. For any event e generated by any Statechart in the AI, a self-loop e/a can be added. The NAC prevents application of the rule to states with on-entry or on-exit actions, thereby ensuring that no pre-existing actions are repeated when the new transition is taken. Since the underlying behaviour is not altered, this transformation can be used to add an orthogonal behaviour in a safe manner.

Another example is found in Fig. 9–3(b), which shows how an action a can be added to a Statechart. The LHS presents an arbitrary transition while the RHS gives that same transition with the action added. The NAC prevents choosing an event that already has an action. This transform is also safe in the sense that the original behaviour encoded in the Statechart structure is unaffected, and thus pre-existing behaviour is preserved modulo the new action affecting it directly.



(a) The selfloop adding rule.



(b) The action adding rule.

Figure 9–3: Transformation rules.

With the goal of creating variations, rule application can be done randomly. One approach is to chose an orthogonal transformation, examine statecharts until one meeting the conditions is found, and then apply the rule. The process can be repeated an arbitrary number of times. The greater the number of rule applications, the more the statechart will be transformed. If only orthogonal transformations are used, the existing behaviour of the AI will be preserved. Interesting, the self-loop adding rule can be applied multiple times (once for each event) to a single state, the NAC in the action adding rule could only be at most once for each transition.

Non-orthogonal Transformations

A non-orthogonal transformation is one that deliberately alters the existing behaviour. An example "re-setting a component" rule is shown in Figure 9–4. The effect is to wrap a statechart in a single super-state with a self-loop transition. This transition is triggered by a "reset" event that returns the statechart back to its original state. The LHS identifies a statechart region, labelled "1", allowing one to track that entity in other parts of the rule. The NAC specifies that this region may not be a sub-state, restricting this rule to operate only on the top-level region of a statechart. The RHS subsequently wraps the top-level region inside a new state with a new transition with trigger "reset" looping on it. The result is a module that forgets state, restarting the behaviour from default. An executor would forget that it has been triggered, while an analyzer may lose its line of analysis. However, if the input event for a 'forgetful' statechart is infrequently generated, any behaviours relying on that statechart may be dormant for excessive periods of time. This suggests that such a transformation is best applied to statecharts that receive frequent triggers, preventing long periods of idleness.

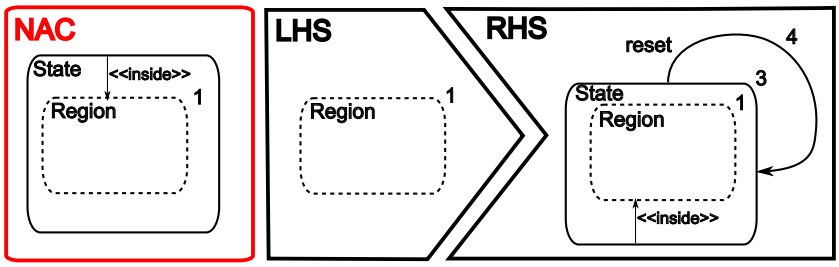


Figure 9–4: A reset transformation.

In the case of a one-time event, a reset transformation would cause the statechart to forget the one-time occurrence, thereby losing any behaviours associated with that event. This necessitates a form of verification to ensure that the generated population still meets behavioural requirements. We propose two primary methods to apply non-orthogonal transformations:

Limited Application: The transformation should be applied in a limited fashion, once or twice at most. This makes it possible to verify the resulting AIs directly, by means similar to the verification of the seed AI. Typically this would consist of testing (such as unit or play testing) to ensure that the AIs still function correctly. With a limited number of variations due to these transformations, the scope of testing is reasonably restricted. If that specific variation passes testing, then it is an acceptable variation and can be applied. Usage of this approach allows for the generation of variations that the designer may not have considered, and thus acts as an aide to the creative design process.

Generation and Culling: As with non-orthogonal transformations, one may apply the rule an arbitrary number of times. This may dramatically affect the population and render some population members unable to fulfil their game role. Like genetic algorithms, we can evaluate each population member with a fitness function and determine if they can meet their role. By creating a large enough population each time, the desired number of correct AIs will eventually be produced. The success of this approach would be based on the ability to effectively distinguish between correct and incorrect NPCs, a task which may be difficult for rare or subtle behaviours.

9.1.4 Generation Procedure

With parameter modification, compositional changes, and rule-based transformations, we are now ready to assemble the overall content generation procedure. Although we described them in the order of parameter modification, module composition changes, and Statechart transformation, our procedure applies these approaches in the *reverse* order. To create a population, the generation procedure is repeated once for each member of the population. As input, we take the set of modules from the seed AI given in the layered-Statechart format (with the interface defined), a set of rules-based transformations to employ, and any additional modules that are available for adding or swapping. At several points, the user will have to enter additional settings; these are noted in the description of the process. Once this is complete, the process generates populations nearly instantly, making it highly practical for rapid prototyping, iteration, and general testing.

First, the Statecharts from the seed AI and the additional modules are matched against the provided transformations, with each application of a rule outputting a new Statechart. The user defines how many times each rule should be matched. Typically, this number should be small (e.g., 1 or 2). In the case of orthogonal transformation, we found that one or two matches was sufficient to introduce the orthogonal behaviour. For non-orthogonal rules, a small number is prudent, as a greater number of matches increases the chance of breaking the underlying behaviour.

Rather than treat the transformation outputs as special cases, we instead treat them as candidates for swapping. The fact that an existing module was

used in its creation is irrelevant—what matters is that the module interface of the transformed module closely matches an existing module within the system. Insertion into a new AI is then done at the module composition stage of generation by swapping the newly transformed Statechart for the source Statechart.

From the set of modules in the source AI, any that can be removed are manually flagged as such, and new modules that can be added are also flagged. In addition, any modules that can be swapped are flagged with their potential alternates (including transformation outputs) - this step could be automated since it relies on matching interfaces. The user can influence the final result by assigning a percentage chance for the occurrence of each available composition modification, including the swaps resulting from rules-based transformations.

Any module that has the potential to be used must be finalized by assigning ranges to their parameters, as per Fig. 9–1. Assigning numeric values and distributions is another step in the process that the user must perform. In this case, the designer can use their judgment to select appropriate values.

With composition flags set and parameter ranges finalized, individual population members can now be generated. Initially, the new AI contains all modules from the base AI. From this, the percentage chance for each composition modification is tested and possibly applied. When the final set of modules has been selected, parameters are assigned values from their defined ranges. This completes the generation process for a single population member. A complete population is generated by repeating this entire process for each member.

9.2 Validation

To prove the validity of this content generation approach, we will demonstrate and experimentally test a number of concrete variations from each of the three levels of abstraction. The squirrel AI, presented in Chap. 4, will act as the generation seed, and the resulting populations will be tested in the Mammoth game world. Increased variance in the behaviour across a population will demonstrate statistically that content generation was able to create a varied population.

9.2.1 Experimental Setup

Testing took place on a map designed to elicit the complete range of core squirrel behaviours, which we define as exploration, gathering and eating food, along with fleeing from potential threats. As squirrels move, a limited pool of energy is drained. Only eating can recharge it. Acorns were placed throughout the map in quantities such that the amount of food available for the squirrels was essentially unlimited.

For each squirrel, we measured the total distance moved, acorns eaten, and energy gained. We expect that as variations in the squirrel population increase, these measurements will vary in two primary ways. Variations that increase randomness should increase the standard deviation, while variations that add or suppress behaviours should shift the mean. Based on the modification we are making, a shift in the variance or mean would indicate that our technique has successfully increased the variability of squirrel behaviour.

Using the squirrel AI as the base, populations of squirrels were generated. The actual generated artifact is a `role.xml` file for Mammoth. In the case of rule-based

Table 9–1: Baseline Parameter Values

Parameters	
Name	**Value**
Wander x-Range	1.5m
Wander y-Range	3.5m
Wander RestTimeMin	1000ms
Wander RestTimeRange	3300ms
Low Threat Range	1.0m
High Threat Range	0.5m
Low Energy	16000units
Critical Energy	5000units

transformations, new SCXML files were created from existing ones, and added to the folder with the already existing SCXML files. Each test used 48 generated squirrel AIs operating on the same map for a period of 4 minutes, with the tests being repeated 3 times for each population. Repeating the test multiple times helps to average out the effects of statistical outliers.

Squirrels make random choices when searching for food, and non-squirrel NPCs wander the map to elicit fleeing behaviours in the squirrels. This is a typical source of randomness for an NPC as it simulates the interactive nature of many digital games. To account for this base variance, the seed AI was applied to all squirrel NPCs on the map. Since the only variation is as a result of game randomness, this test gives us a baseline for variance in the game. The values used for the baseline are given in Table 9–1.

Results of the baseline tests are listed as "Identical Squirrels (Control)" scenario in Table 9–3. Here, the '±' results are of central importance. The indicated value is one standard deviation, giving us a direct measure of the

Table 9–2: Parameter Modification Settings

Parameters		
Name	**Min**	**Max**
Wander x-Range	1.5m	3.5m
Wander y-Range	1.5m	3.5m
Wander RestTimeMin	1000ms	2000ms
Wander RestTimeRange	1000ms	5000ms
Low Threat Range	High + 0.5m	High + 1.0m
High Threat Range	0.5m	1.0m
Low Energy	Critical + 500units	Critical + 10000 units
Critical Energy	2500units	7500units

variance. If later populations are more varied, we expect to see that reflected by an increase in the standard deviation. Changes in the mean indicate that introduced variations are shifting behaviours predominantly in one direction.

9.2.2 Parameter Modification

Modifiable parameters in the squirrel AI were related to wandering, energy levels in the Energy Sensor, and threat radius in the Threat Analyzer. There was a dependency between the low and high threat radius and a critical dependency between the low and critical energy levels. Dynamic ranges were used in both cases. The actual ranges used are listed in Table 9–2.

Since the statechart composition is not being modified, the behaviour of the squirrel remains the same (e.g., the squirrel will still flee from players, collect acorns, etc.). However, variability in the expression of those behaviours should increase as squirrels wandered different distances, ate at different energy levels, and fled from more distant threats. The "Parameter Modified" results in Table 9–3 confirms this, demonstrating that parameter modification is successful in

Table 9–3: Variation Test Results

Test Scenario	**Acorns Eaten**	**Distance Moved**	**Energy Gained**
Identical Squirrels (Control)	5.6 ± 2.0	37.8 ± 10.8	29.7 ± 10.8
Parameter Modified	5.6 ± 4.5	59.0 ± 22.9	47.6 ± 23.3
Parameter Modified w. Stuttering	3.7 ± 2.7	35.7 ± 9.9	36.5 ± 9.8
Parameter Modified w. Alternate	4.4 ± 1.8	72.8 ± 26.1	60.2 ± 25.8

increasing variation of simple quantifiable aspects of NPC behaviour. Qualitatively, the increase in distance moved appeared to be due to more frequent and longer range fleeing, as the ranges for fleeing were mostly greater than the baseline. More movement meant more energy drained, increasing the value of eaten acorns and increase the mean energy gained. While varying a basic parameter such as move distance may not visibly impact the game for a player, variations in properties such as damage dealt, health, or speed will be extremely noticeable as these are fundamental to player survival in many games.

9.2.3 Module Modifications

There are three fundamental modifications available at this level of abstraction: adding modules, removing modules, and swapping modules. Since we are now changing behaviour, generation will produce variations that are more observable to players.

Our first compositional change was to add a Stutter Executor to 50% of squirrels. This caused affected squirrels to move in several small segments with a short pause between each. In generating this group, we again employed the parameter ranges found in Table 9–2.

One would expect squirrels to move less distance overall when they stutter move, due to the frequent pauses. The data reflects this, appearing as "Parameter Modified w. Stuttering" in Table 9–3. Movement returned to baseline mean with a similar variance. However the number of acorns eaten was much lower, largely because those that stutter use much less energy overall and so require less food.

A second modification came in the form of a 50% chance of swapping out the Squirrel Brain for a new Squirrel Brain Alternate, shown in Fig. 9–5. The default brain prioritizes wandering over gathering food, while the alternate brain prioritizes gathering food, and only wanders when the squirrel is carrying an acorn. As a base, we again used the parameter ranges from Table 9–2 along with 50% wander incidence. Results are listed in Table 9–3 as "Parameter Modified w. Alternate". Interestingly, the number of acorns eaten by this group is quite low while the distance travelled and energy gained is quite high. The greater efficiency of the second group allows for more active squirrels in general, creating a livelier environment.

9.2.4 Rule-Based Transformations

Since rule-based transforms have the ability to wildly alter behaviour, validation focused on demonstrating that our orthogonal transforms are in fact safe and can alter behaviour in a clear and consistent manner.

Mammoth has a chat system that is available to NPCs, but the basic squirrel lacks the ability to communicate. Through random application of the rule in Fig. 9–3(b), an action was added that created an event triggering a chitter (a call made by actual squirrels). Three Statecharts meeting the LHS of the rule were

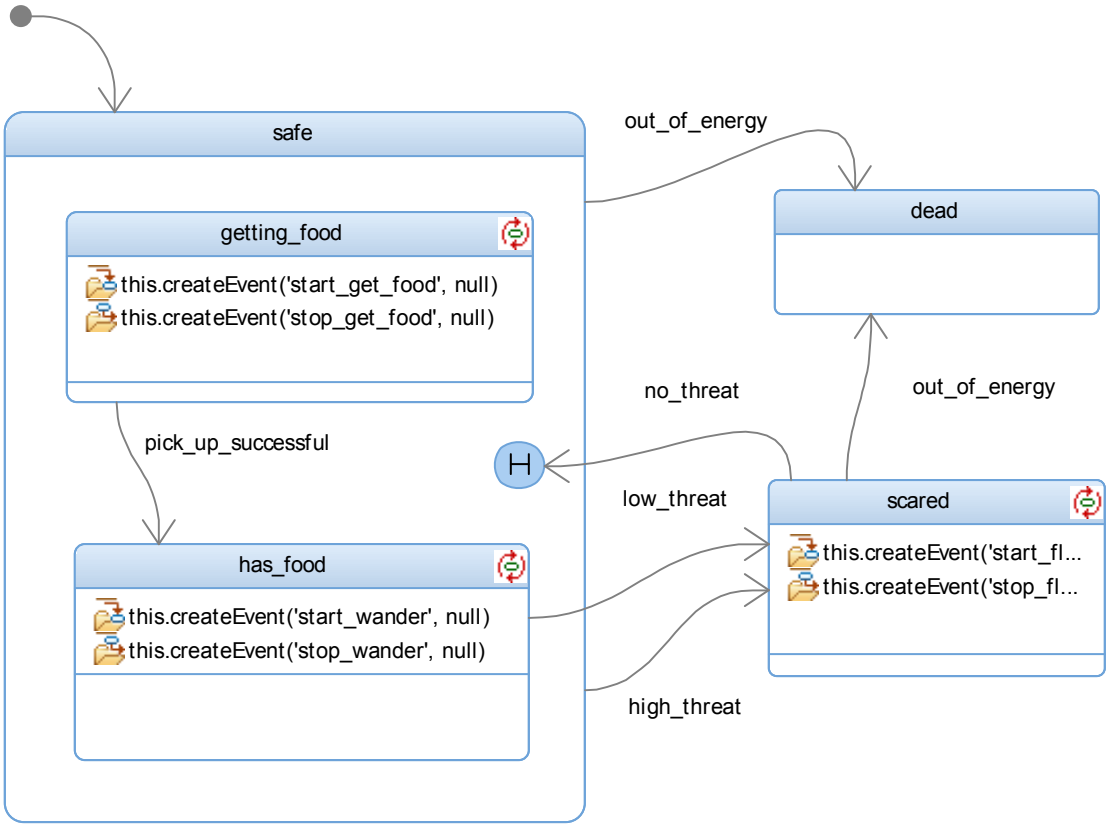


Figure 9–5: Alternate Squirrel Brain

selected at random, then transformed to add the chitter action. Testing was then augmented to record the number of chitters made by squirrels. Depending on the frequency of the event attached to the chosen transition, chittering would occur quite frequently, or quite rarely. One third of the squirrels chittered only once or twice during the test, while the rest would chitter anywhere between 4 and 30 times. The mean was 4.9 chitters with a relatively large standard deviation of $\sigma = 6.1$. This creates interesting variability in the occurrence of our new orthogonal behaviour.

Finally, the rule-based transformation in Fig. 9–3(a) was applied to introduce standing behaviour, causing a squirrel NPC to occasionally rear up on its hind legs. Since standing is non-orthogonal with respect to moving, this transformation necessitated the addition of a Stand Coordinator that only permits standing when the AI is not moving. In essence, this altered standing to become a safe orthogonal action, since standing at an unacceptable time would be ignored by the coordinator. This demonstrates how complex behaviours can be added through the modification of the AI at multiple levels. Testing was augmented to record the number of stands executed by the squirrel. The results show several clear groups, with 42% of the squirrels standing 1-5 times, and 40% standing more than 15 times. The mean for the entire standing population is 19.3 stands with $\sigma = 28.1$. Depending once again on the frequency of the event trigger and the frequency with which the modified state is entered, the resulting behaviour possesses considerable variation.

9.3 Directed Generation

While creating a varied population may be appropriate in some cases, the designer may want to instead generate populations of AI that vary along a specific design metric. Repeated generations may increase along this metric, such as waves of enemies that present an increasing challenge. Instead of choosing variations randomly, directed generation would apply them such that the population exhibits certain traits more frequently, or suppresses other traits. This results in a population that varies along a specific metric, generating populations to satisfy the design goal.

This section describes how our base variation approach was modified to instead vary along a specific design metric. This is implemented and tested in Mammoth by creating a squirrel catching mini-game with squirrels that vary in their ability to successfully evade the player.

9.3.1 Difficulty Classification

To be able to generate a population that varies along a metric, we must be able to accurately predict the effect an individual variation will have. Game difficulty specifically is a widely researched topic, and many techniques exist to predict difficulty. For instance, Zook *et. al.* [73] present a tensor factorization that analyzes game play metrics to predict player success when encountering new situations. Jaffe *et. al.* [33] explore competitive balance assessment through restricted game play, determining the impact of certain abilities. We see our generation approach as being complementary to these approaches; they tell us what changes should be made, our process enacts these changes in a varied manner. With this in mind, we seek to show that our process generates populations that vary along a desired metric, and do not intend to introduce any novel technique to assess or predict difficulty.

Basic changes using parameter modification can increase difficulty quite easily. A squirrel that runs faster is obviously more difficult to catch. These types of changes are quite trivial and can be performed using simpler approaches, however, our approach offers more interesting variation strategies. We seek to change NPCs at the behavioural level by exploring different flee techniques, namely, running

away in a straight line, fleeing in an arc, and fleeing by climbing trees and disappearing for a short period. When coupled with basic parameter modification, our technique provides a rich means for generating new and interesting combinations, thereby improving immersiveness [50] of the experience while meeting design goals.

9.3.2 Mammoth Implementation

A mini-game was added to Mammoth wherein the goal was for a player to catch as many squirrels as possible in 5 minutes. The player could only catch a squirrel by getting very close to it, while the squirrels would actively evade the players and flee. Using our generation technique, we created populations of squirrels that varied in their evasion effectiveness, allowing for the management of game-play difficulty through AI generation.

The difficulty of a population was evaluated using an AI-controlled NPC squirrel catcher. The catcher's behaviour was to run in a straight line at the nearest squirrel, automatically catching it when the catcher was close to the squirrel. The game lasted five minutes, and the number of successfully caught squirrels was tracked. For each of the three flee strategies, a population of 48 squirrels was generated that used only a single strategy. Parameter modifications were used, and some were given the alternate brain, but flee proximity and move speed were not varied. Each group was generated and tested three times. The results are summarized in fig. 9–6, showing that fleeing in a straight line presents the least challenge, circular fleeing offers a moderate challenge, and the tree fleeing strategy is the most challenging. The results are distinct with no overlapping error bars, giving us three base difficulty levels.

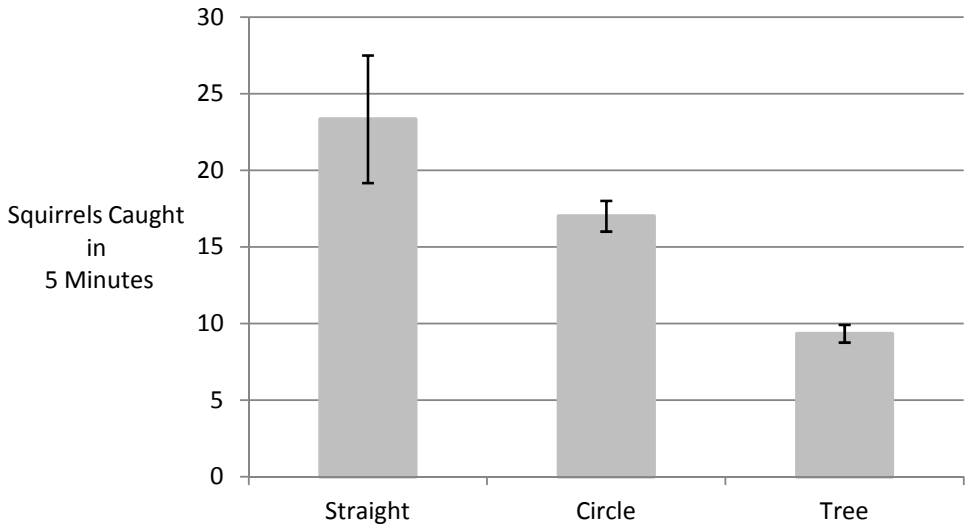


Figure 9–6: Baseline difficulty for flee-homogenous populations

9.3.3 Generation with Difficulty Targets

The next step is to show that non-homogenous populations will allow us to create populations of different difficulty. We do this by generating half-and-half split populations from our three flee types, with the expectation that mixed populations will create an averaging of the two difficulties. Results are shown in fig. 9–7 as the normal proximity group. All three combinations give results that fall between the values for the flee-homogenous groups, but the error is much higher. This is explained by variability in squirrels encountered by the NPC in a given trial run. The averaging effect here is not a simple mean. The Circle+Tree combination has a complex interaction: when a squirrel flees in a circular pattern, the catcher follows in a circle, remaining in one small area. Coincidentally, this is a very good strategy to catch any nearby squirrels that have climbed a tree. If the

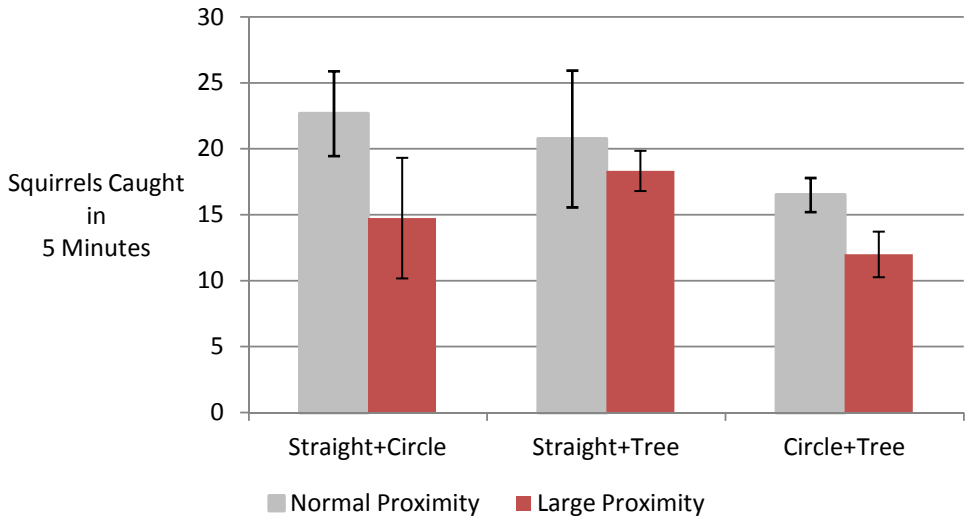


Figure 9–7: Baseline difficulty for mixed populations

catcher forces a squirrel up a tree, but remains in the area due to following of a circle-fleeing squirrel, the catcher will immediately snare the treed squirrel when it descends. Because of this, the catcher performs better than expected for this combination.

To better illustrate the power of our approach, we performed a final set of tests that combines the mixed population with targeted parameter modification. For this set, the range proximity at which a squirrel would flee was increased by 50%, meaning the squirrel exhibits flee behaviour much farther from the Catcher. The results are shown in fig. 9–7, which clearly shows the scaling effect this modification has on all squirrel populations. The scaling factor is not consistent across strategies, which again demonstrates the complexity of behavioural interactions.

What we have demonstrated is that our content generation approach is suitable for creating populations that vary along a defined metric. Our squirrel catcher caught fewer squirrels when given a more difficult population. By performing a proper difficulty analysis (such as those by Zook *et. al.* [73] or Jaffe *et. al.* [33]), this approach can effectively perform directed content generation of NPC AI.

9.4 Expanding the Approach

Our approach to procedurally generated NPC AI provides a novel tool for game development. Generating new behaviours for AI opens up exciting new possibilities, as designers now have a new design space to explore.

Directed generation provides a method to meet design goals through variation within the AI population. Combining our generation approach with a sophisticated methodology for gathering game metrics will allow for generation of highly specialized populations of NPCs that meet a specific goal within the design context. We look forward to seeing how such approaches can be applied, and how procedurally generated NPCs can become a vital ingredient in future games.

In the future, 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 limited to pre-defined variations, 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.

CHAPTER 10
Related Work

This work touches upon many different aspects of digital games research and software engineering. As such, there is a large body of research that inspires, impacts, or competes with these results. To aid in evaluation of the impact of this research, we will describe and compare existing related work.

Discussion of related work will be divided into four separate conversations relating to the four principle components of this work. First, we will discuss works relating to the layered statechart approach itself. Next, we will discuss reuse of AI, along with works relating to Scythe AI. The next area will focus of validation of game AI, while the final section will cover AI variation and generation.

10.1 Layered Statechart-Based AI

We advocate the use of layered statechart-based AI, as it is a principled, model-driven approach with clear benefits from a software engineering perspective. In this section, we will weigh the benefits and drawbacks of this approach with respect to commonly employed formalisms, including behaviour trees, goal-based approaches, and simple FSMs and HFSMs.

10.1.1 Finite State Machines

Finite state machines are the oldest formalism used in modelling game AI, and are still commonly employed. Here, states represent behaviours and transitions are triggered to change the behaviour exhibited [20, 23]. Hierarchical FSMs (HFSMs)

incorporate aspects of statecharts [25] by allowing states to contain substates with internal transitions.

Because of the lack of clear formalisms describing either FSMs or HFSMs as used in the field of Game AI, they act as a poor basis for a model-driven approach. For example, one implementation may allow guarded transitions, while another may not, or one may allow entry and exit actions, while another does not. These differences mean that any model describing only one approach has a high chance of failure when combined with another. Since statecharts generalize FSM and HFSM approaches, they offer a clear advantage: a model-driven approach for statechart-based AI can be used for FSM, HFSMs, and statecharts.

The primary drawback of typical state-based approaches is the explosion in complexity as the number of states and transitions increase. However, work by Heckel et. al. has shown that when combined, hierarchical subsumption architectures have low representational complexity when compared against behaviour trees, FSMs and HFSMs, or pure subsumption architectures [27]. As the layered statechart approach employs both hierarchy and subsumption, Heckel's work suggests that the layered statechart-based AI presented in this thesis possesses a low representational complexity when compared with other common AI approaches. Further research could verify the degree to which our formalism corresponds with Heckel's conclusions. That being said, the small size of the statecharts in our Halo AI implementation in Chap. 5 does appear to be in line with Heckel's theoretical result.

10.1.2 Behaviour Trees

Behaviour trees are an emerging formalism in game AI. Highly successful industrial games such as *Halo 2* [32] and *Spore* [28] have brought this approach to the forefront. In 2012, the industry focused Vienna Game/AI Conference 2012 featured an entire workshop devoted to behaviour tree implementation.

As introduced in §2.1.2, behaviour trees use leaf nodes to read and write to the game state. Non-leaf nodes control the ordering in which leaves are accessed. However, it is unclear how trees should be structured when behaviours should be repeated or skipped, and also unclear how behaviours should be stopped or terminated when the AI must react to new information. This highlights the primary failing of behaviour trees, which is a lack of ability to react to events. As a result, implementers of behaviour trees end up adding parallelism to attempt to provide reactivity, with mutual exclusion enforced only by construction. Recent advances, such as event-driven and data-driven BTs improve efficiency [7], but sidestep reactivity issues.

10.1.3 Other Architectures

Goal-based planning approaches, including hierarchical task networks, have been successfully employed in industrial games such as F.E.A.R. [54], or the academic language ABL [48]. In this approach, a library of actions with preconditions is created, defining the capabilities of the AI. Behavioural goals establish preconditions for a task, then a search finds and executes an action that satisfies preconditions and accomplishes the goal. Dynamic awareness and reactivity are compromised by this model. In practice, the planner must be

repeatedly run to ensure that the selected action is still the appropriate one, which is both inefficient, and can lead to jittering in edge cases.

Learning approaches are typically avoided in game AI. One notable attempt was Lionhead Studio's *Black and White* in 2001, where a player controlled a creature based on the Belief-Desire-Intention model [22], with a perceptron for reinforcement learning. While the novelty and high quality of the game made it a commercial success, the nature of learning combined with the variation in play styles resulted in a game that was very easy for some players, and impossible for others. In general, learning adds unnecessary complexity to controlling the level of difficulty posed to the player, and complicates testing since the behaviour of the AI is an emergent property of the learning process. Few if any AAA titles since have employed a learning approach.

10.2 AI Reuse

Reuse of AI is a highly important technique that is growing in importance, and has been applied in the context of most of the formalisms introduced here. Kevin Dill, a major proponent of modular AI, proposed a pattern based approach to modular AI [14]. This introduced the concept of abstracting partial logic making up the AI into patterns that could then be matched when building new AIs. Later, he furthered the concept by showing specific types of patterns that could be used, and how they could be expressed in C++ and XML code [15]. While this work lacks the advantages of a model-driven approach, it offers a highly practical and industrially relevant approach to AI reuse.

In the context of reuse, superstates in HFSMs can be treated as modules and exported to new AIs [37]. This approach is valuable, but omits important details such as code portability, and has no provision for interaction with internal states. The strict hierarchical nature of HFSMs can be limiting as it places restrictions on how transitions between states can be modelled. Since HFSMs form a proper subset of statecharts, our reuse model generalizes this approach.

Behaviour trees offer a reuse model based on pruning and reusing branches [45]. In a practical sense, however, the extent of reuse is limited as individual tree nodes are often highly game or AI-specific—code actions and abstract, high-level behaviour are intimately entwined in behaviour tree models. Although behaviour trees delineate clearly how the system chooses behaviours, they suffer from a lack of modal states encapsulating different behavioural groupings. Reuse in practice has been demonstrated primarily in terms of modifying an existing AI [16] with incremental improvements rather than porting AI logic to a fundamentally new context.

Goal-based approaches offer a reuse model based upon the addition of new goals, or new actions that can be used to change the AI state. In theory this is a quite straightforward path to reuse. In practice, actions and goals must be highly specific, and basic knowledge about the game world must be encoded into the process of action selection. Thus, adding new behaviours requires substantial effort in rebalancing existing behaviours such that all behaviours are expressed correctly.

10.2.1 AI Reuse Tools

In the niche of AI Reuse, Scythe AI may indeed be the only tool for that specific purpose. However, there are many tools and game middlewares for producing game AI in general.

The Unreal Engine 3 [70] 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. There is no explicit support for reuse in the Kismet tool.

Among scripting tools, a noteworthy tool is *ScriptEase* [53], which is 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. Instead of modelling a decision making structure, scripted AI provides simple rules that cause AI actions, such as "when the player enters the room, this NPC should attack".

The paid version of the game engine Unity provides basic support for basic AI through A* pathfinding modules. Authoring modules for behaviour trees and FSMs are available through the Unity Asset Store. These allow a user to build a behaviour tree, or draw an FSM, but do not provide higher level functionality. Scythe AI, in comparison, does not provide authoring tools, but does address reuse and verification of existing AI modules.

10.3 Verification of Game AI

Verification of game AI, like the rest of game development, is highly focused on testing. While this can include formalized elements such as unit or regression testing, the process is largely manual. Games are typically tested by designers during development and eventually by dedicated testers. During this process the entire game is tested, and typically the AI itself receives no specialized testing. As such, improvements to AI verification tend to focus on improvements to the testing process itself, such as increasing code-coverage [49]. Even the well-regarded text by Millington and Funge [52] has nothing to say on the subject of testing. Our approach is radically different, both because it is a formalized approach, and because it provides a strong guarantee of correctness with respect to the verified specifications.

There do exist several formal approaches that have been employed to verify statecharts. Primarily this has been accomplished through the use of model-checking to exhaustively explore the state-space. Methods include transforming the statechart into a Promela representation and then using the Spin model-checker to verify LTL conditions [51, 44, 60], and by using intermediate Kripke structures to verify CTL conditions. [72]. Fundamentally, these two approaches yield the same result, expressibility differences between CTL and LTL notwithstanding.

Our approach is differentiated by the use of cooperating statecharts. This necessitates the explicit modelling of event distribution, ensuring that all statecharts advance one event at a time in accordance with the Rhapsody semantics of statecharts [25]. This in turn required modelling the external event queue, passing

external events as steps, and modelling micro steps arising from statechart transitions as micro steps using the AI event queue. As well, we explicitly represent the environment with our external event generator, and help control state-space explosion by using guards to disable events.

10.4 AI Variation

Research into software product lines [10] suggests a starting point for model-based generation, giving a process that builds upon inter-product commonalities to efficiently produce similar outputs. Krueger phrases variation as modifying not the product line, but the production line itself [38]. However, the complex temporal interactions that exist in variation management do not arise in our work. Our goal of variation within a game AI does not give a strong set of requirements on the output, and it is enough that the output is different than the source, giving our work a different focus than typical product line research.

Smith and Mateas [63] formally describe the design space of a generation technique, then use answer-set programming to create generators covering that design space, but do not discuss specifically how this could impact NPC design. Mateas *et. al.* have done considerable research into generating NPC responses in their game *Facade*[66], in which an NPC generates a narrative discourse that corresponds with player intentions, but the behavioural capabilities of the NPC are not altered.

Non agent-based AI is sometimes used to model group behaviour in crowd simulations [67, 40]. It avoids the need to manage a large number of individual agents by placing behavioural triggers in the environment that act upon members

of the crowd. Thus, individual NPCs no longer need a unique agent to power them, yielding a large gain in computational efficiency. However, this is counter to the goal that NPCs possess behavioural uniqueness. This can be somewhat mitigated by individualizing NPCs to the current game context [12], or by increasing diversity [68].

Modern gaming platforms are powerful enough to manage agent-based AI crowds, as demonstrated in the game 'Hitman: Absolution' [34], which has crowds consisting of dozens up to hundreds of NPCs. This is convincing support for the applicability of our approach, as it shows that agent based approaches can successfully scale to a crowd of NPCs. That being said, our work differs by focussing on creating identifiable individuals, rather than large crowd simulation.

CHAPTER 11
Conclusions

Academic research into computer game development fills a curious niche. As a largely industry-driven field, much of the progress comes from developers themselves, searching for solutions to the daily problems they face. However, in the rush to meet deadlines and ship products, good software engineering practice often becomes the first casualty. Academia, unconstrained by the rush to market, is in a unique position to deeply explore this often neglected aspect of game development. As the game industry has matured, and genres stabilized, the value of good software engineering practice has become increasingly important.

In this work, we have presented a comprehensive approach to developing computer game AI through the use of model-driven design techniques. By treating it as a software engineering problem, and applying the principles found in software engineering, we were able to successfully describe a new approach to developing game AI using layered statechart-based AI. As demonstrated by the work on our Halo AI in Chapter 5, this approach is capable of representing modern AIs used in AAA gaming titles. The true advance, and that which is most useful to industry and academia, is the demonstration that developing game AI using a principled approach based on a well-defined formalism yields clear benefits. These include reusable AI, verification of desired AI behaviour, and content generation. Industrial developers will benefit from having access to these approaches, and since

there is a clear formalism, academia can continue to explore this research topic and further clarify the approaches contained herein.

What this work both envisions and enables is for AI developers to build AIs from standard reusable components, sharing and distributing various AI modules to build new AIs. This uncouples generalized AI logic from a specific game or a series, and allows AI to become an asset much like the 3D models used in graphics development. Combined with the verification techniques described in Chapter 8, developers can focus on perfecting novel behaviours, rather than reimplementing basic tasks for each new game.

From a theoretical point of view, this work has delivered a complete and thorough exploration of layered statechart-based AI, and shown how this formalism can be used to represent both simple and complex AI logic for NPCs. We described how reuse is enabled by the fundamentally modular approach, and presented a tool designed for the modular reuse of layered statechart-based AI. Verification of modular AIs was addressed, and we demonstrated a method to ensure syntactic correctness. As well, we made significant progress towards verification of behavioural correctness at the level of AI logic, an approach which is novel in the field of AI development for games. Already, it has shown itself to be capable of verifying smaller AIs, and with additional work on optimizing verification of larger AIs, it has the potential to become a beneficial tool for game developers.

This work also has broader implications for the field of computer science. By creating modular AI, we make it possible for researchers in other fields to more

easily package and share their work. This includes some of the more powerful techniques in the AI field, such as neural networks or generalized learning. If a module could be designed such that it can accept input and provide output in the form of events, then the mechanism within (be it a statechart, a neural network, or other decision maker) would pose little barrier to integration with a statechart-based AI. This creates the potential for new and exciting combinations and applications of AI techniques between AI practitioners.

11.1 Future Work

The diverse nature of this work provides ample opportunity for future work. Some of this involves minor improvements to the application of the approaches described herein, while others are more theoretical questions suitable for academic exploration. We divide our discussion of future work into two broad categories: extensions, and further research.

11.1.1 Extensions

In all research, there are obvious extensions to the principle work. While these are often tangential to the main thrust of the research, they represent interesting and potentially useful avenues to explore.

When creating statechart modules, we noted the existence of several statechart patterns, such as the quantizing sensor. While we have presented several archetypes, it is highly likely that many more patterns exist. A thorough survey should be undertaken to get better coverage over what could be a large pattern space, making a pattern-based approach more useful.

Modular reuse shows promise as a highly practical improvement to the software development process, but amount of benefit has not been measured in any meaningful way. The logical next step in the research is to begin a round of user studies to validate the claim that it improves the development process. Crafting such a study is challenging in its own right, and would represent a significant addition to the research.

Verification, especially that of semantic correctness, offers a powerful tool to AI developers, yet the utility is constrained by the difficulty in creating correct and meaningful specifications. It would be tremendously helpful to augment Scythe AI such that it creates specifications automatically, including reachability, and some basic behavioural requirements. Similarly, it would be valuable to automatically filter Spin output to simplify error trace analysis. The usability of the process must be improved for it to become practical, and these extensions are excellent first steps.

Our variation process allows developers to create new AIs from a base AI. One of the most powerful techniques available comes from rule-based variation of the statecharts. Adding more transformations would result in more dramatic variations, increasing the power and usefulness of the approach, but faces challenges in ensuring correctness, either by design or through verification.

Finally, the Scythe AI tool can be extended to support more of the work in this thesis. In its current state, it is capable of facilitating modular reuse. The experience could be improved by adding support for functional groups, and push button creation of statechart pattern modules. As well, support for additional

features such as variation and verification could be added. This would round out the tool and make it usable by industry and academia alike.

11.1.2 Further Research

Our entire model is based upon the usage of statecharts. As noted in the background section, this is just one approach that is currently used in industry. Interoperability could be enhanced greatly by investigating how other formalisms, especially behaviour trees could be integrated. What type of interface would a reusable behaviour tree have, and how could it be linked to a statechart to build hybrid solutions?

While SCXML was used to implement statecharts, running an SCXML interpreter at run time is inefficient and would not be considered in an industrial situation. The missing piece is to develop a SCXML compiler that outputs an executable statechart in efficient C++ or Java code, or even C# for Unity 3D. Creating a compiler that is correct, performs optimizations, and output highly efficient code forms the basis of an excellent research challenge. Work by Wasowski [71] and Samek [59] provides an excellent starting point for such an effort.

The largest problem in verification is also the most fundamental. As the AI grows in size, model-checking becomes intractable due to state-space explosion. However, squirrel AI behaviours took very little time to verify, and error traces where generated quickly for the much larger Halo AI. Because of this, we hypothesize that further optimizations may be enough to allow a more complete verification of larger AIs. The processes themselves could be optimized, for example, by abstracting away the contents of functional groups. Views could be created,

where only a subset of statecharts are represented. The view could be analyzed independently, with the actions of excluded statecharts being modelled as external events. If views can be generated that are similar in size to the squirrel AI, then rapid verification becomes possible. In the Promela representation, any optimization in the amount or usage of guards would reduce the number of states seen by Spin, and would further improve the verification process. Successful optimizations might move the needle enough that intractable specifications become tractable, and slow verifications become fast.

Our work on variation is an interesting diversion from the main thrust of this work, and bore interesting results. While our method of AI variation generates correct AIs, there also exists a number of fundamentally different approaches that would be interesting to explore. Non-orthogonal rule-based transformations can create wildly variant output, many of which will not be able to perform the role of the NPC correctly. By using a genetic approach, whereby we cull bad variations, and create new variations using the good ones as seeds, it would be possible to create wildly divergent, yet still correct variations. With an appropriate fitness function, these offspring could be optimized allowing for directed generation. Exploring the feasibility of such a method, and comparing the results with ours would make for an interesting research topic.

Through the conceptualization of the AI as a model, we were able to achieve interesting results. But with a well defined-model, it follows that there is a higher level definition for the model itself – a meta-model. A meta-model could be used as a basis to explore variations in the formalism, or to more accurately express

concepts within the formalism. For instance, patterns can be represented with a defined syntax, avoiding the need to express the entire statechart for each pattern. Common transition patterns could also be represented using syntactic sugar. Work of this nature is the precursor to a domain specific model, or domain specific language, either of which would be a valuable shorthand for future practitioners.

Appendix A: The Halo AI as Layered Statecharts

Given in this appendix is the full Halo AI recreated as a layered statechart-based AI. The full description of the AI, module listing and description, and key features are outlined in Chap. 5. Verification and corrections from Chap. 8 have been performed and are shown here.

A.1 Sensors

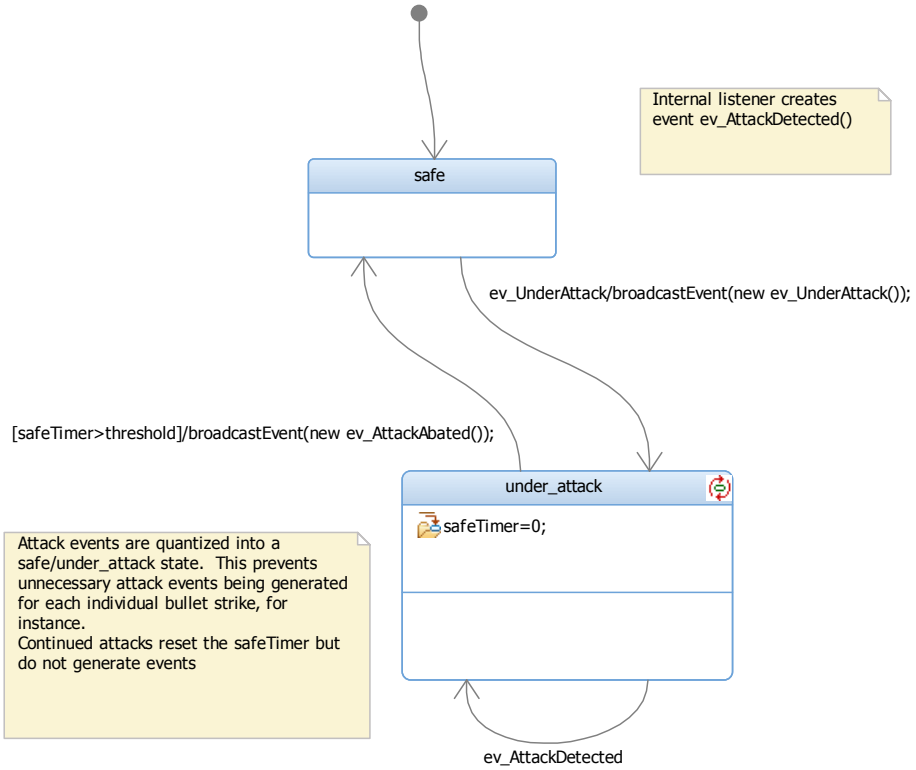


Figure A–1: The *AttackSensor*.

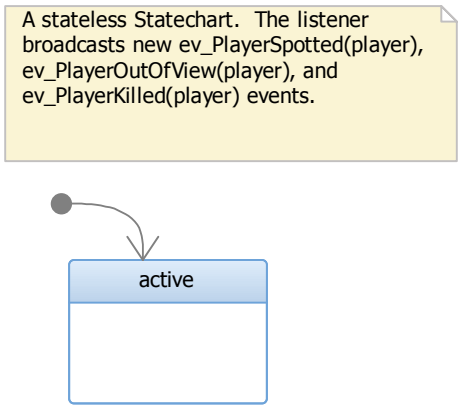


Figure A–2: The *CharacterSensor*.

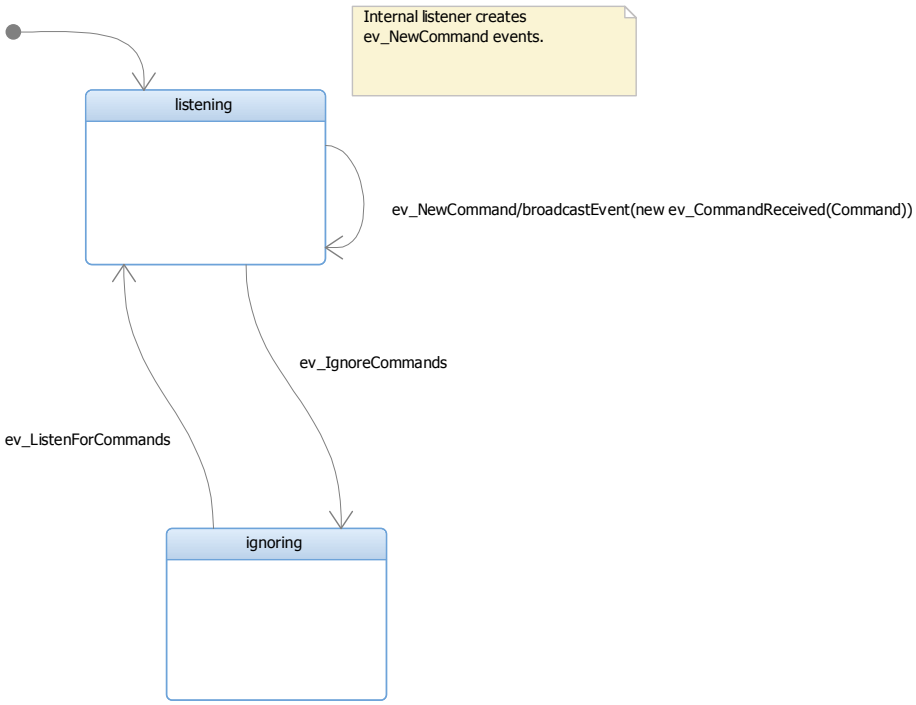


Figure A–3: The *CommandSensor*.

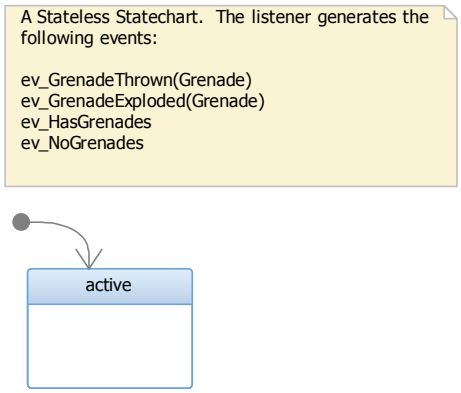


Figure A–4: The *GrenadeSensor*.

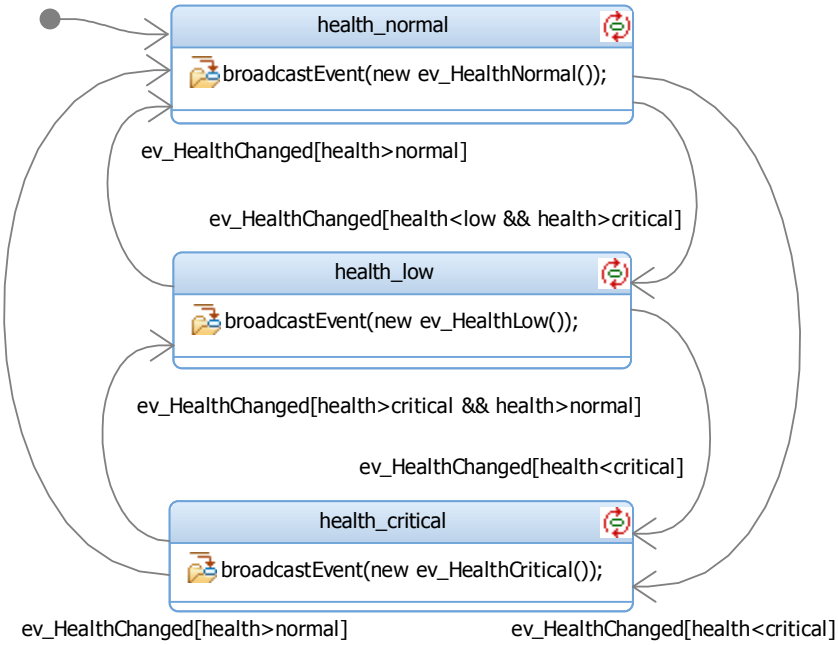


Figure A–5: The *HealthSensor*.

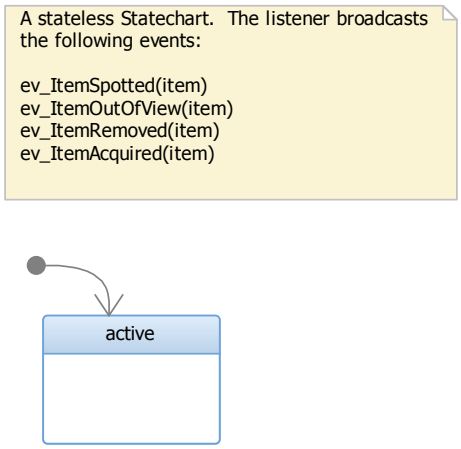


Figure A–6: The *ItemSensor*.

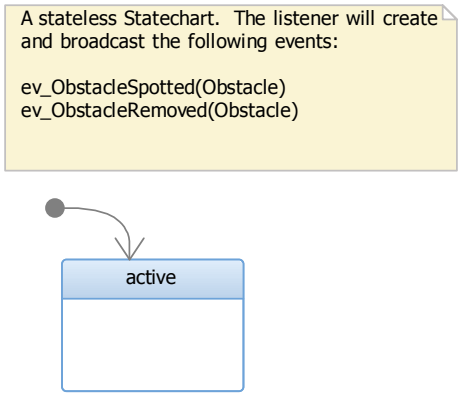


Figure A–7: The *ObstacleSensor*.

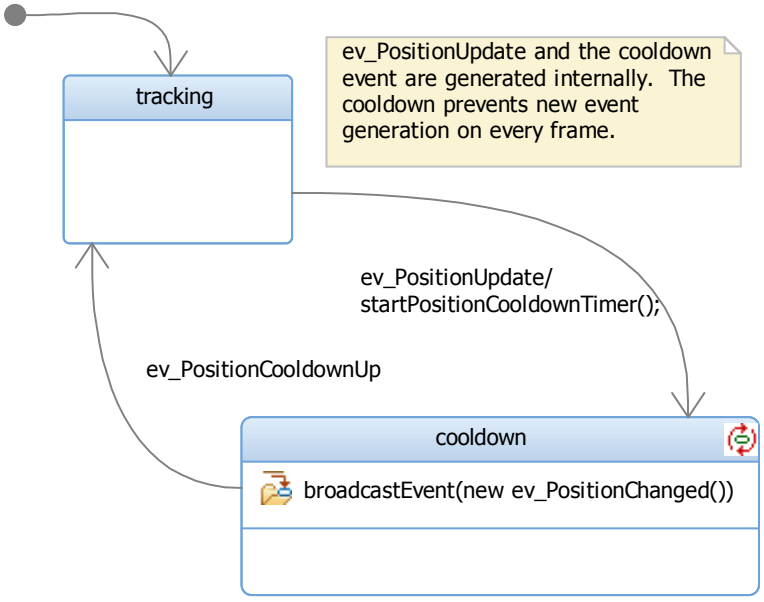


Figure A–8: The *PositionSensor*.

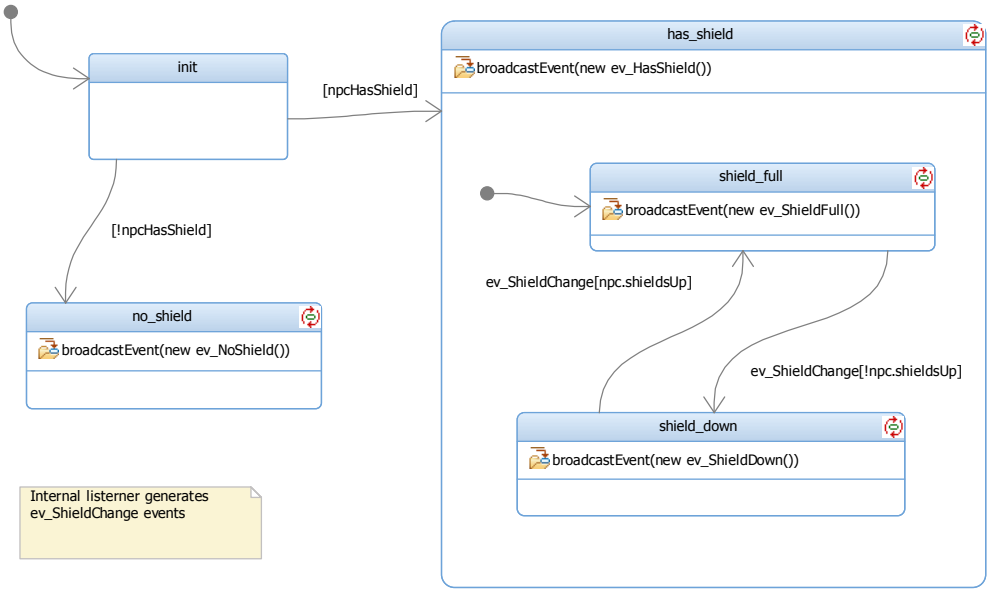


Figure A–9: The *ShieldSensor*.

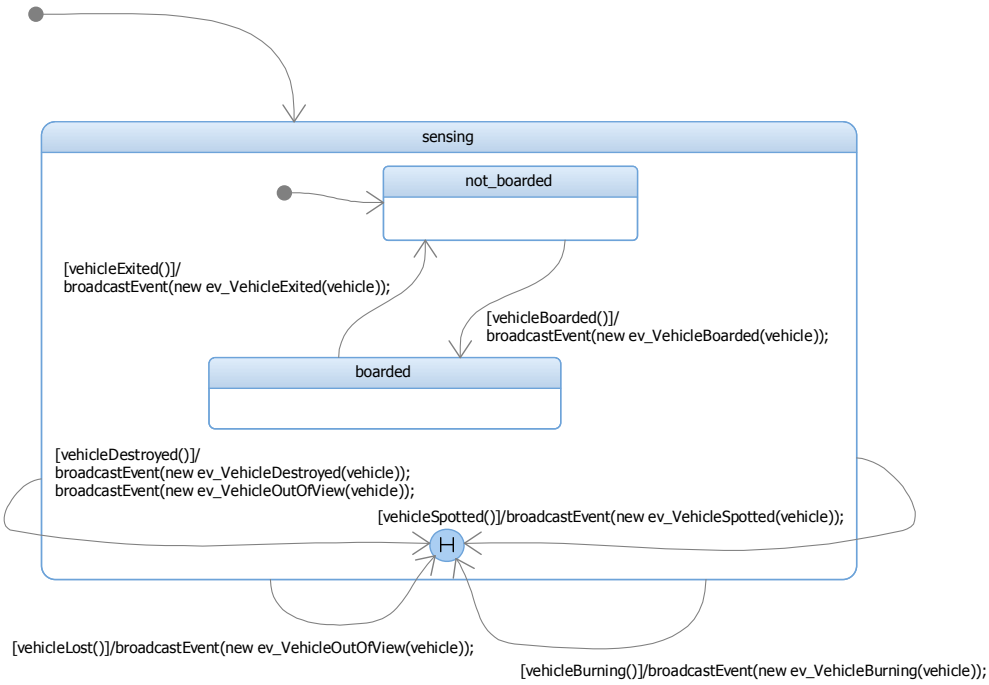


Figure A–10: The *VehicleSensor*.

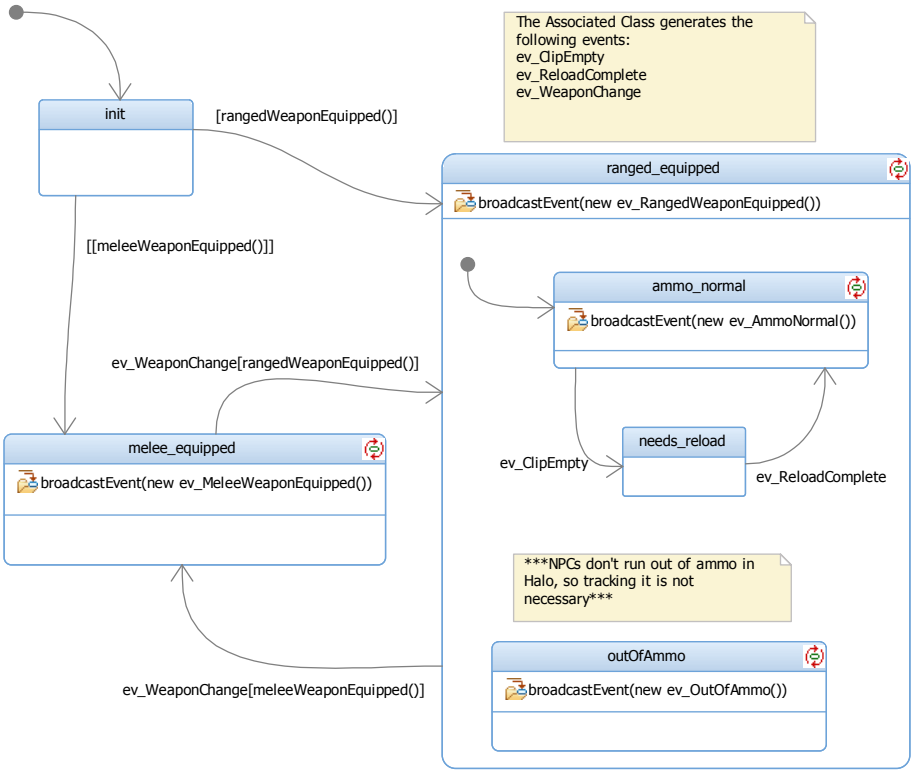


Figure A–11: The *WeaponSensor*.

A.2 Analyzers

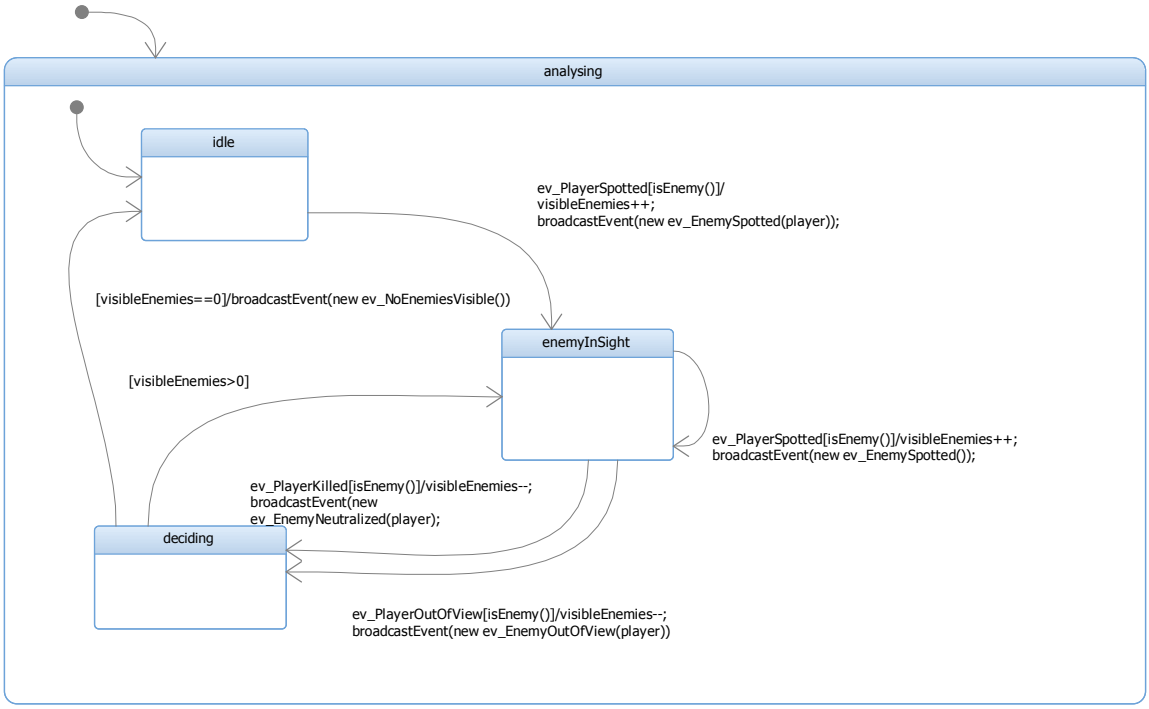


Figure A–12: The *EnemyAnalyzer*.

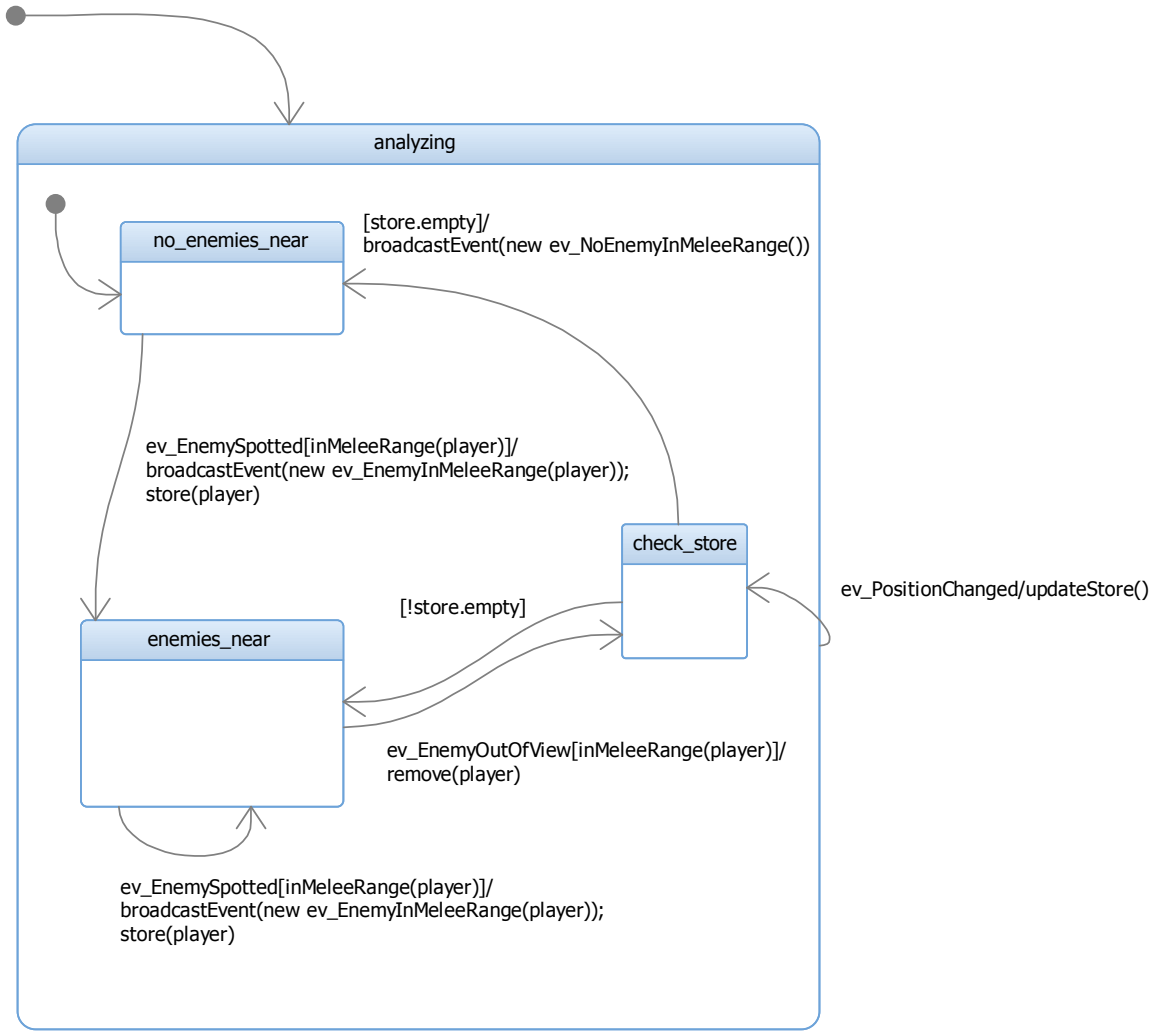


Figure A–13: The *EnemyProximityAnalyzer*.

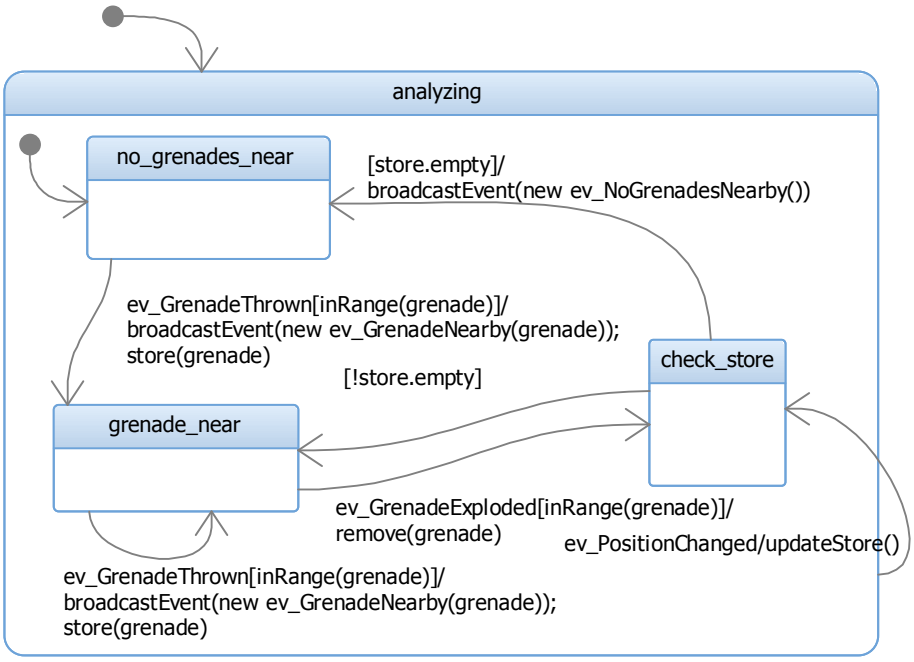


Figure A–14: The *GrenadeProximityAnalyzer*.

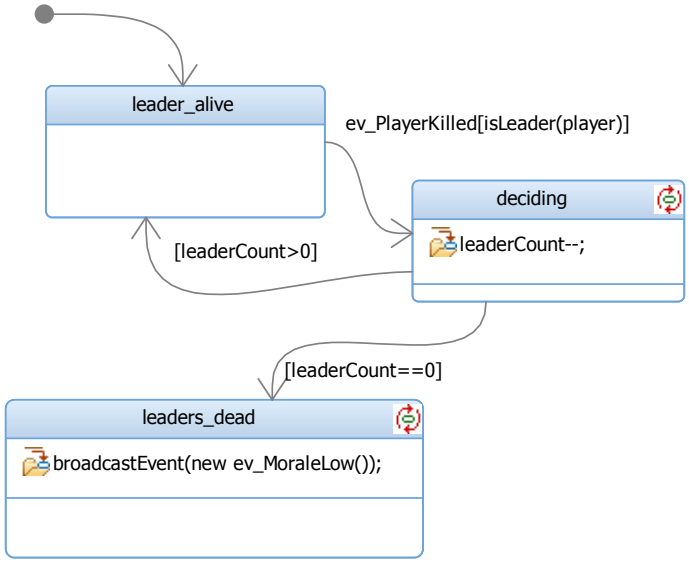


Figure A–15: The *LowMoraleAnalyzer*.

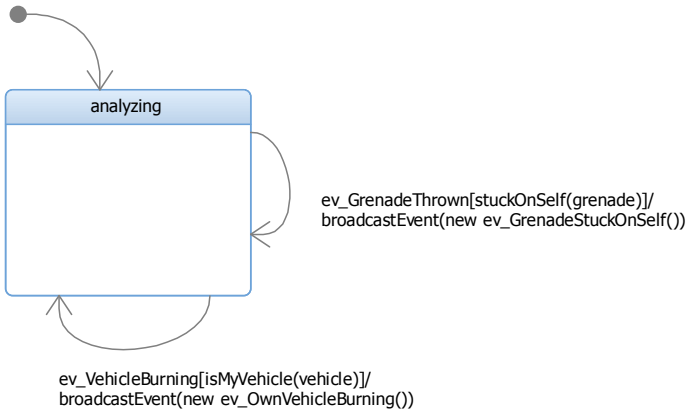


Figure A–16: The *SpecialEventAnalyzer*.

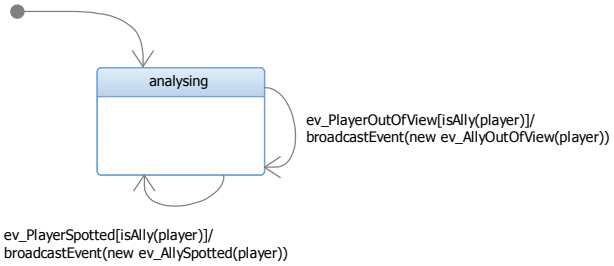


Figure A–17: The *SquadAnalyzer*.

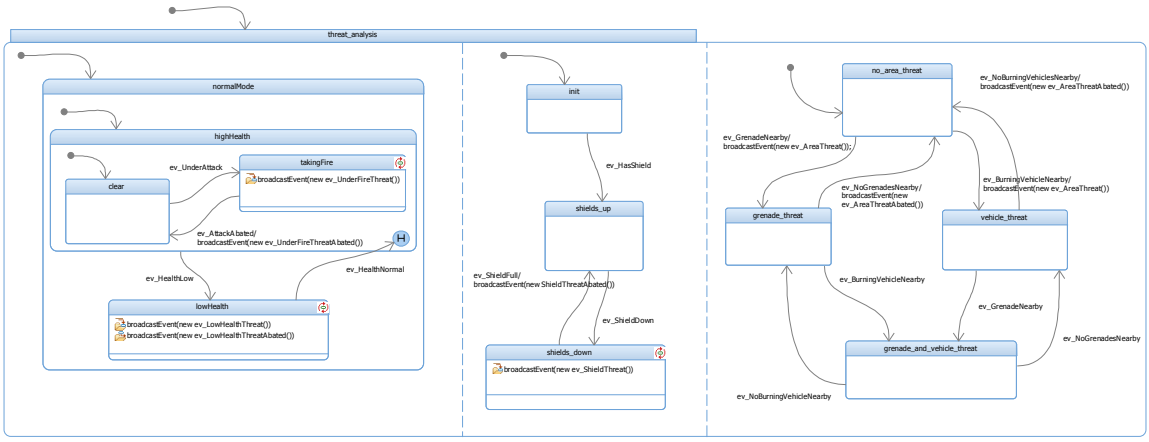


Figure A–18: The *ThreatAnalyzer*.

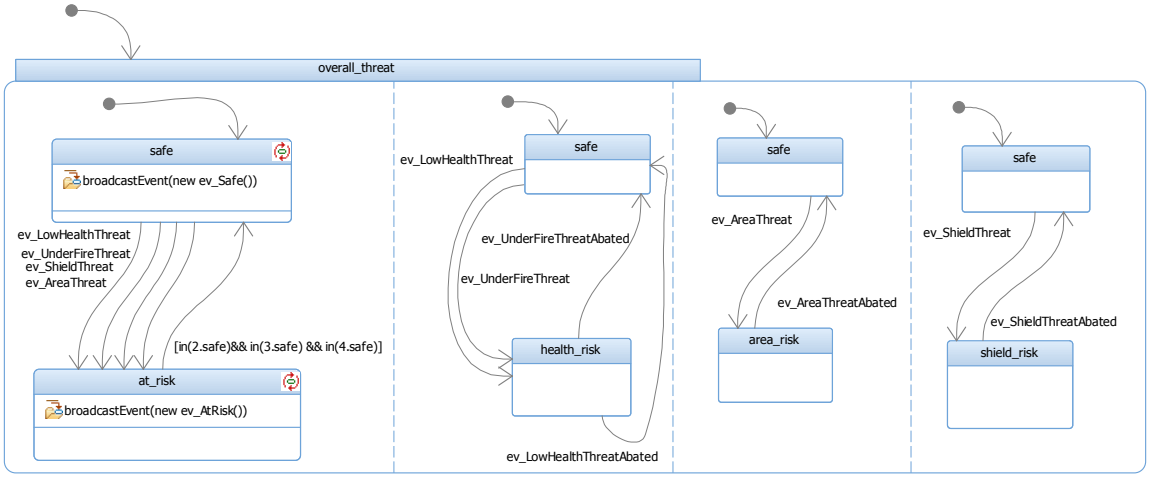


Figure A–19: The *ThreatCompilerAnalyzer*.

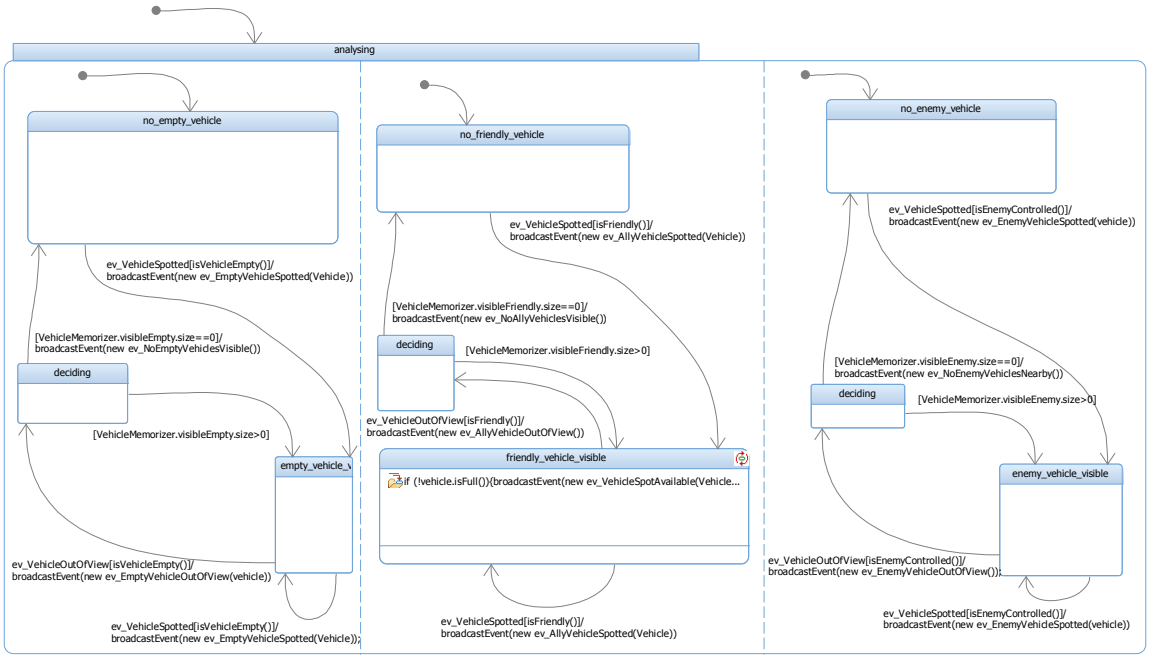


Figure A–20: The *VehicleAnalyzer*.

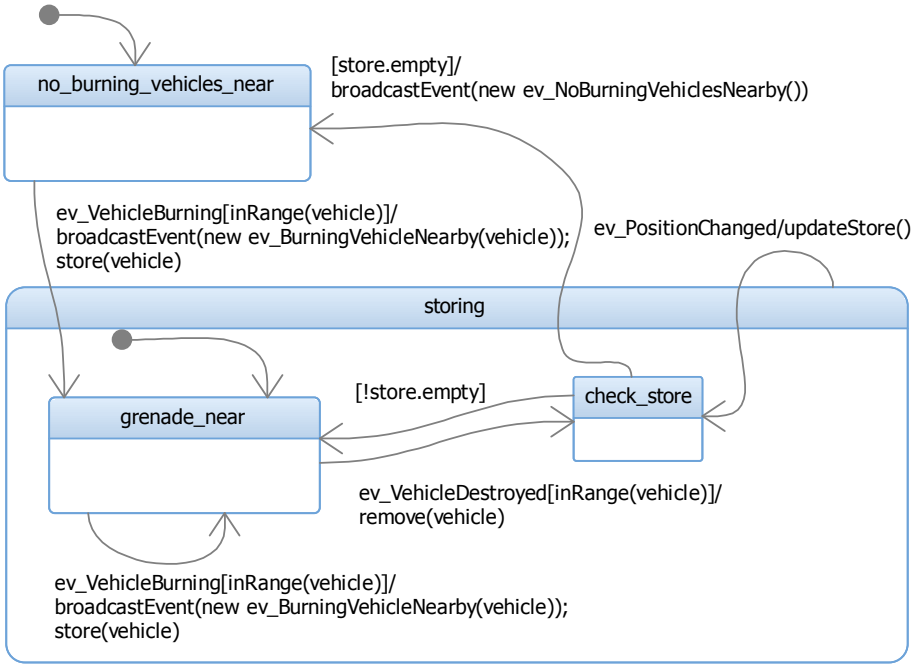


Figure A–21: The *VehicleProximityAnalyzer*.

A.3 Memorizers

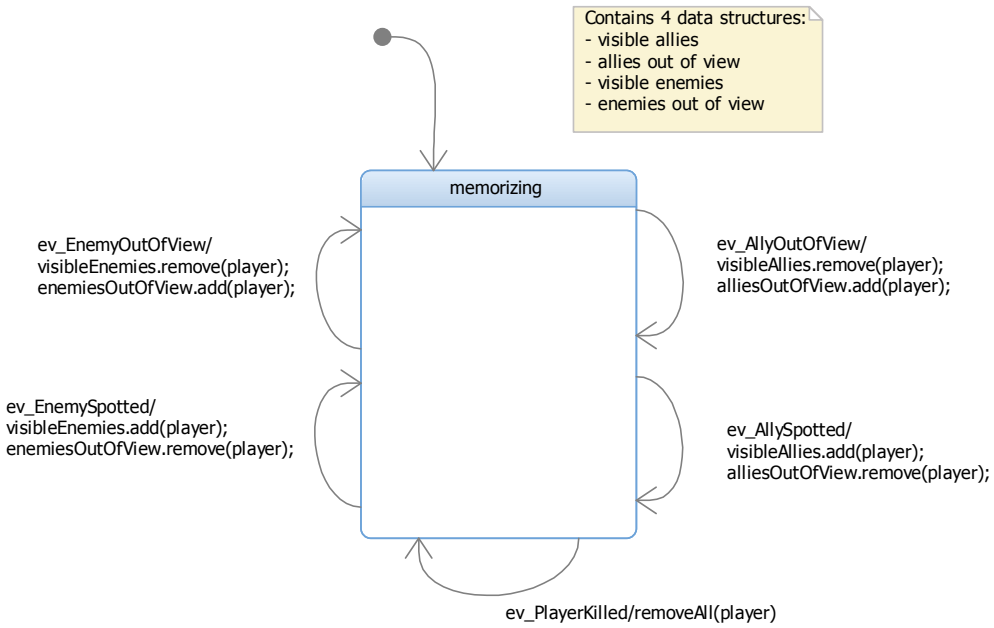


Figure A–22: The *CharacterMemorizer*.

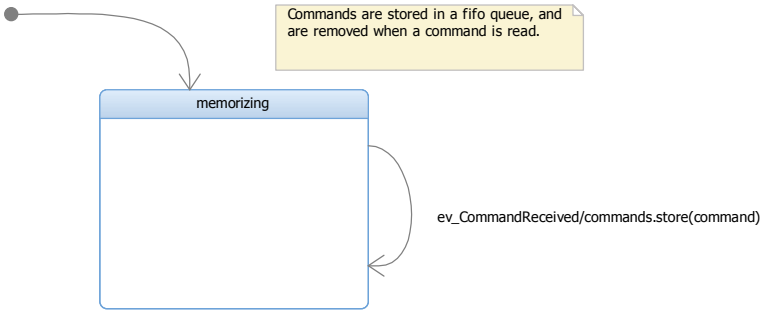


Figure A–23: The *CommandMemorizer*.

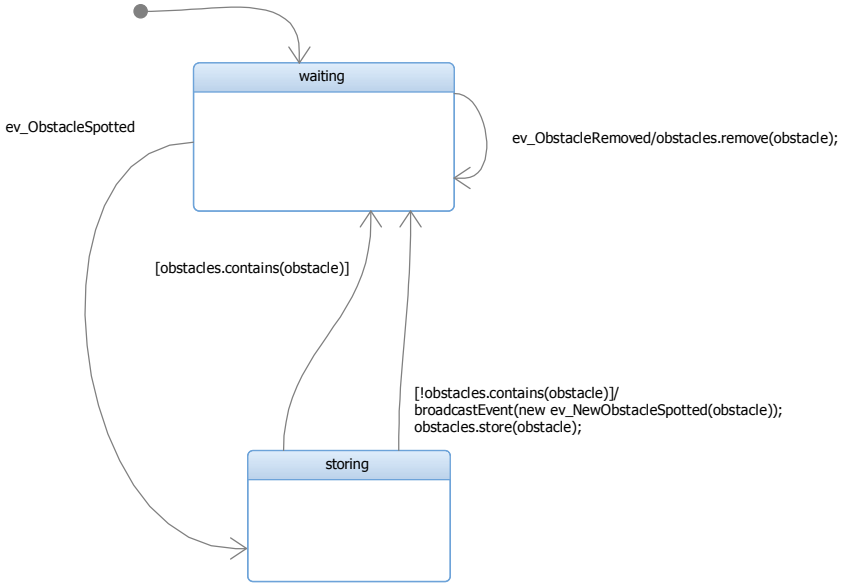


Figure A–24: The *ObstacleMemorizer*.

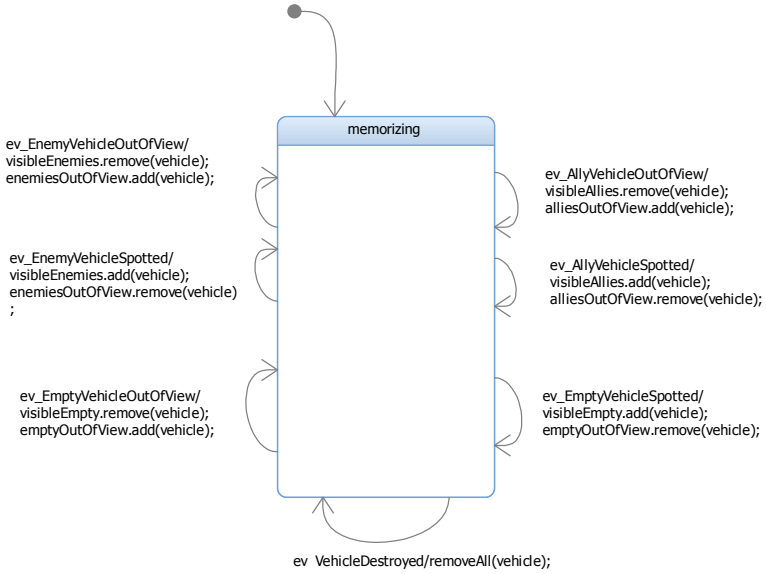


Figure A–25: The *VehicleMemorizer*.

A.4 Strategizer

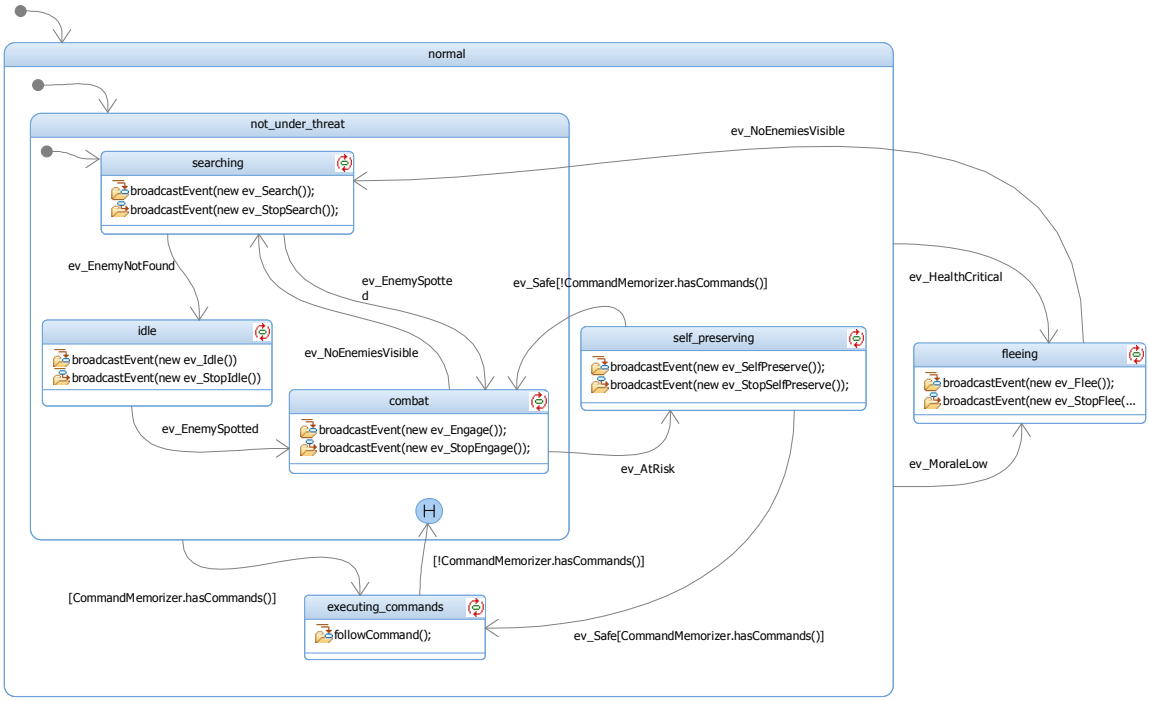


Figure A–26: The *Brain*.

A.5 Deciders

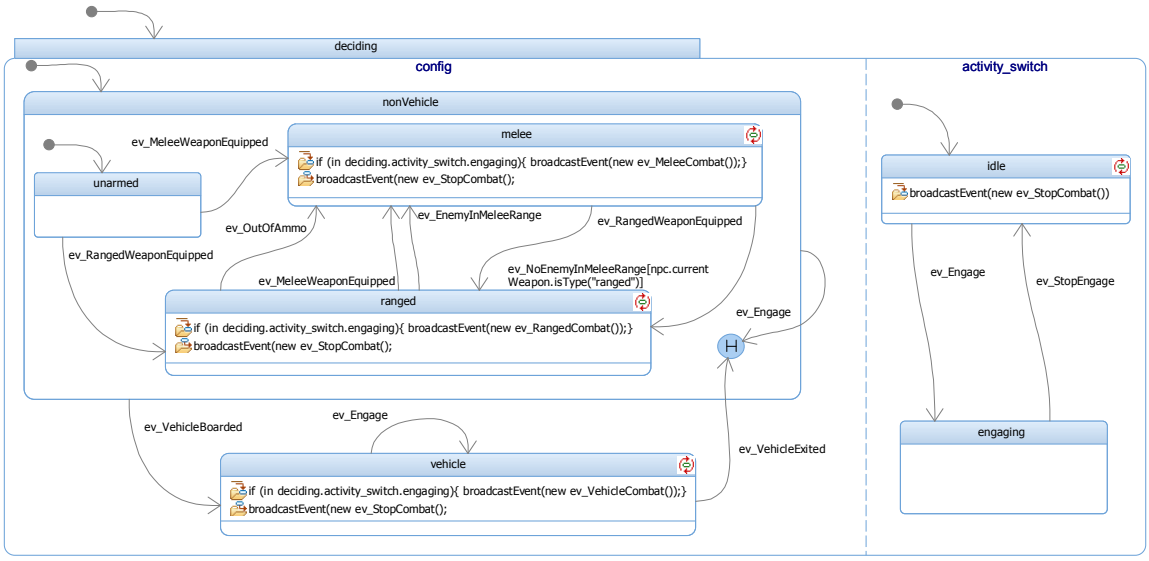


Figure A–27: The *CombatDecider*.

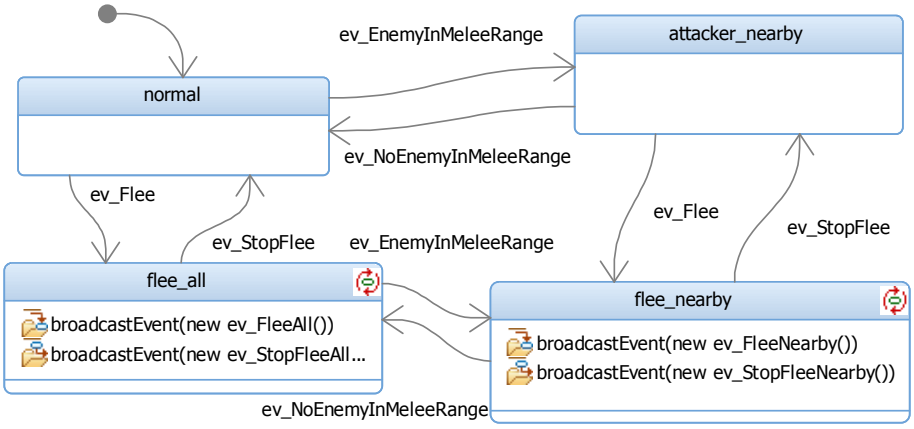


Figure A–28: The *FleeDecider*.

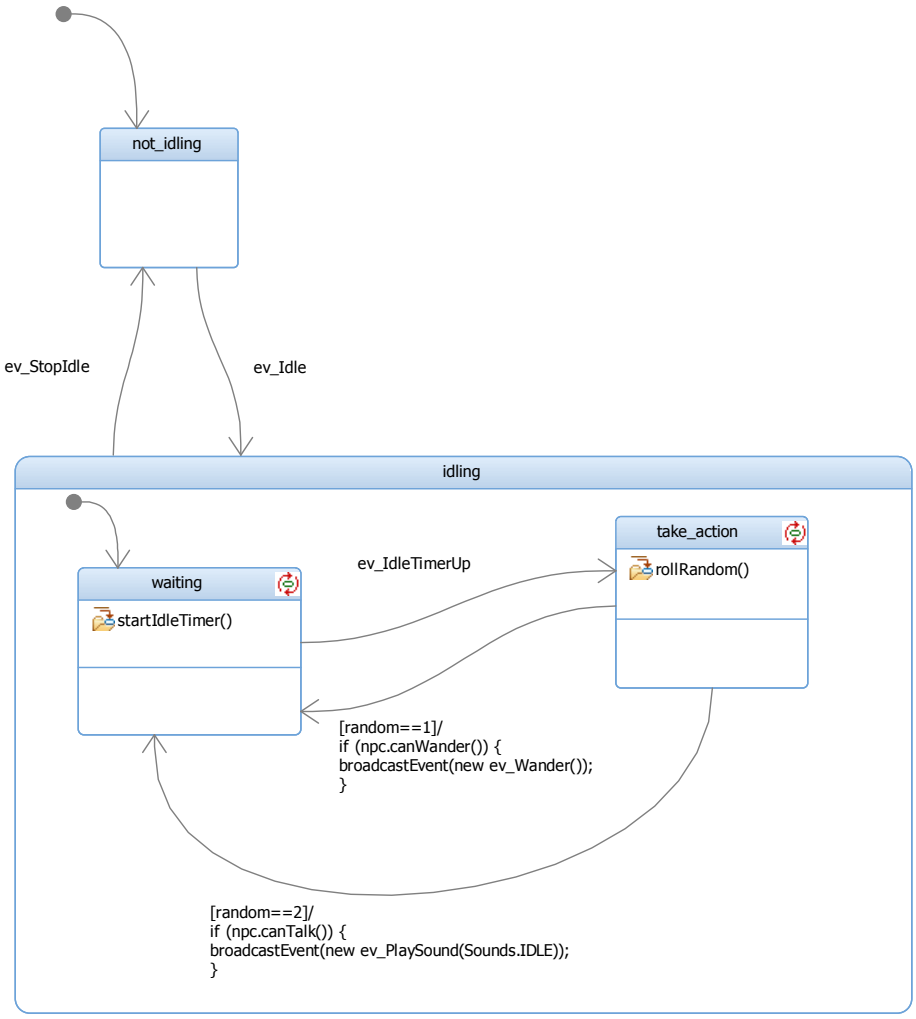


Figure A–29: The *IdleDecider*.

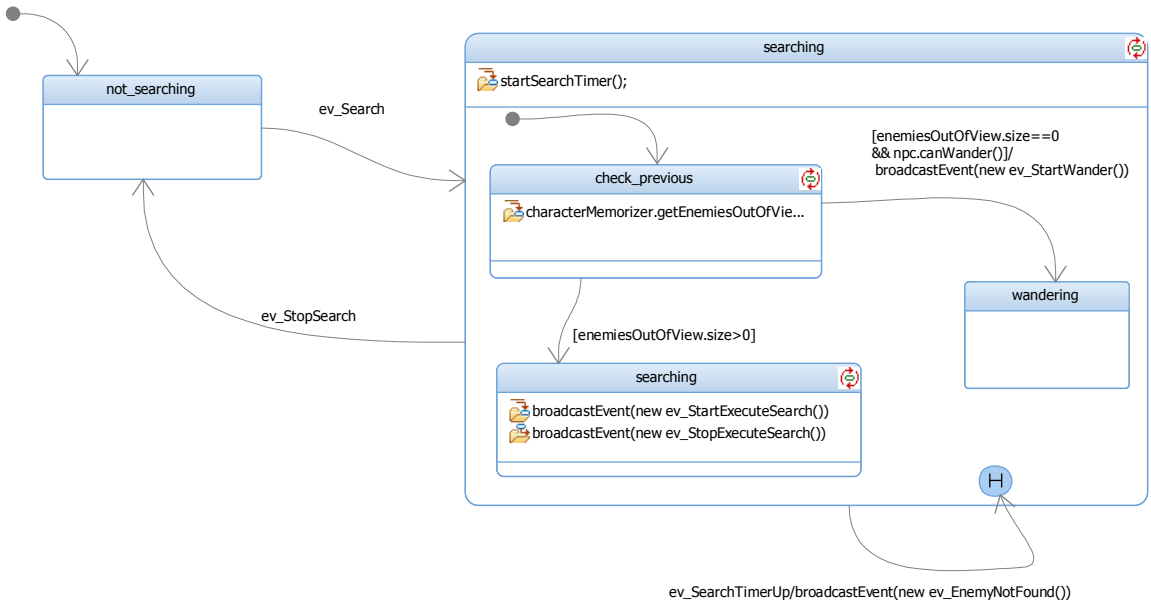


Figure A–30: The *SearchDecider*.

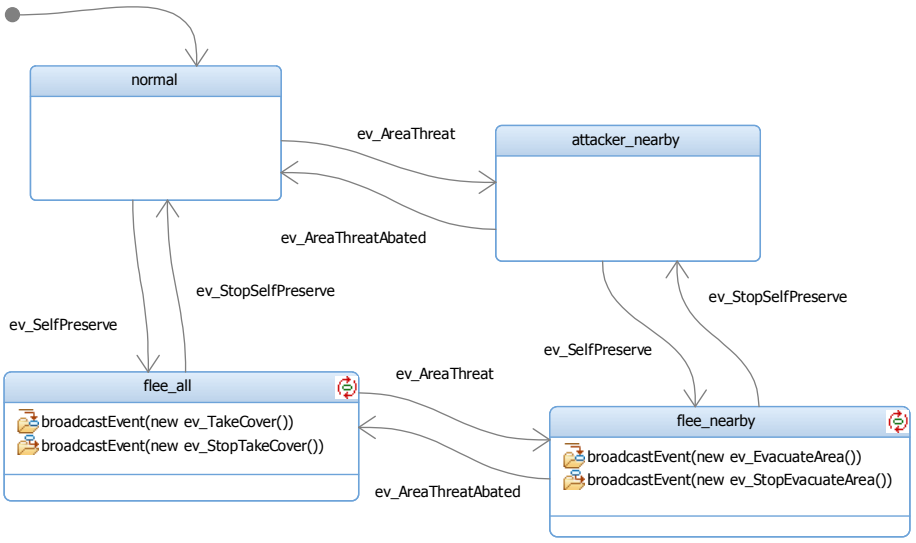


Figure A–31: The *SelfPreservationDecider*.

A.6 Executors

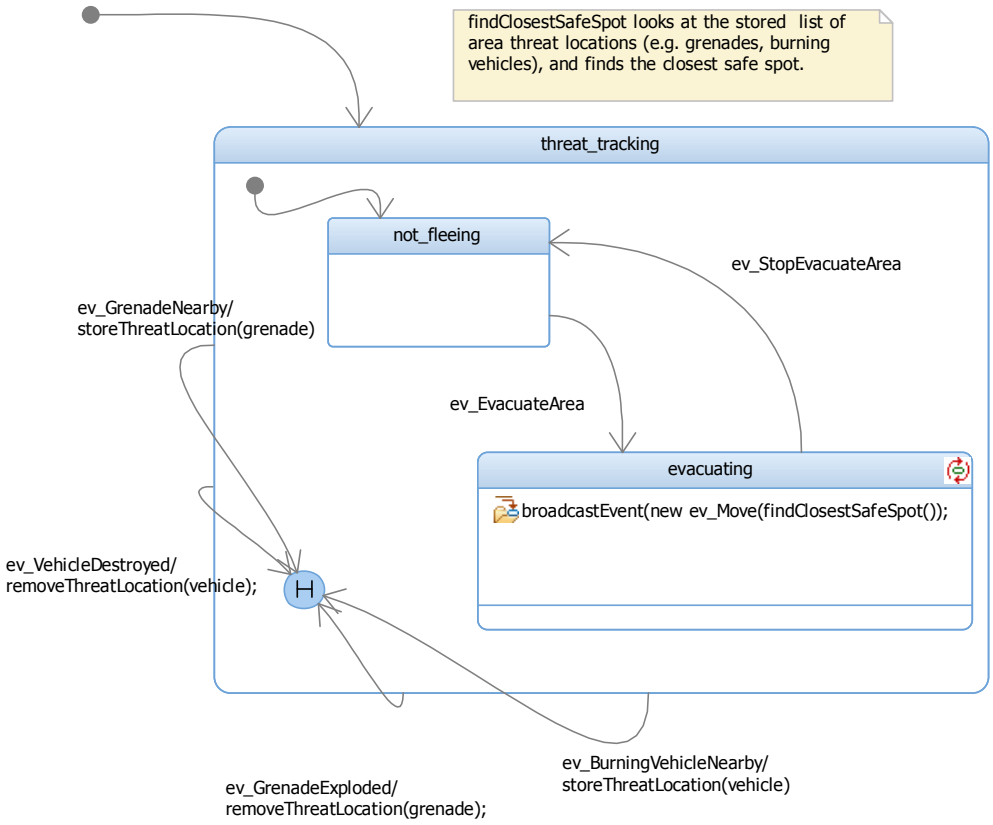


Figure A–32: The *ClearAreaExecutor*.

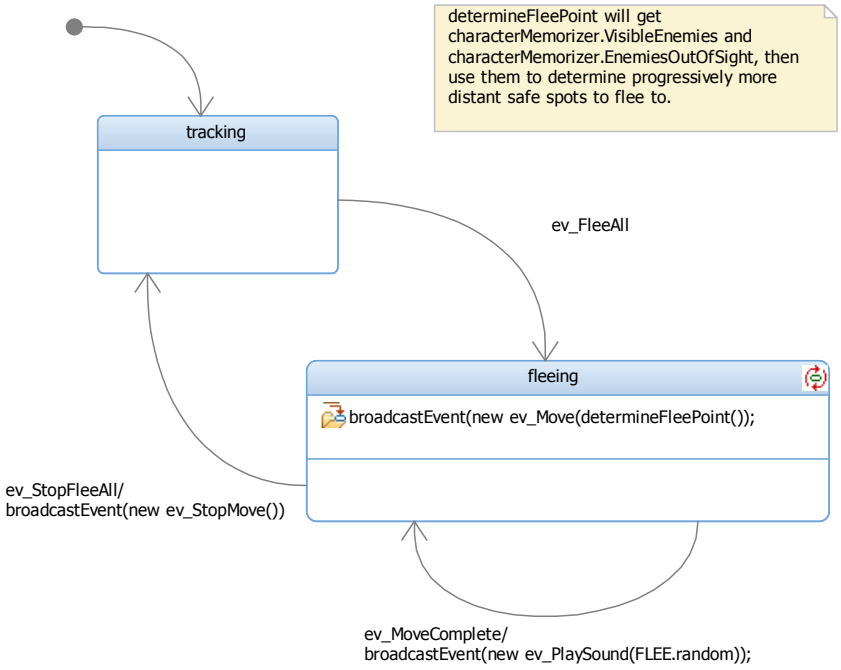


Figure A–33: The *FleeAllExecutor*.

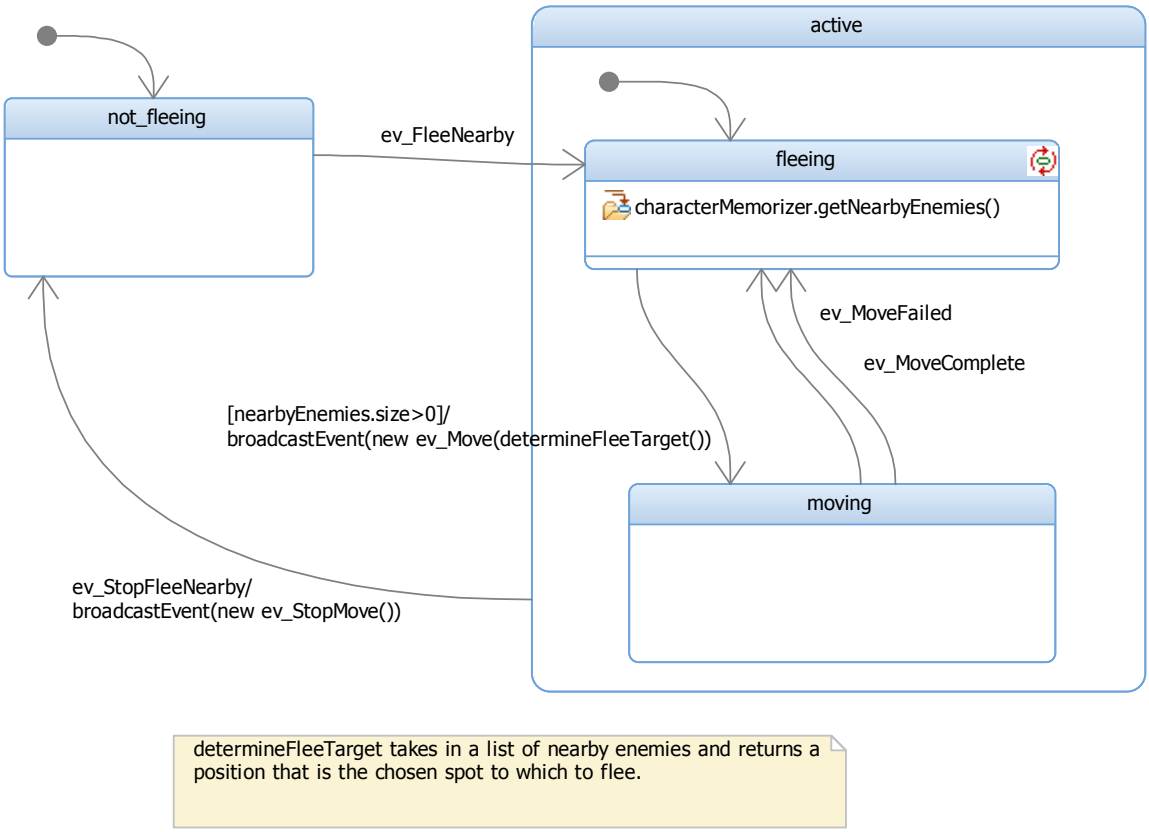


Figure A–34: The *FleeNearbyExecutor*.

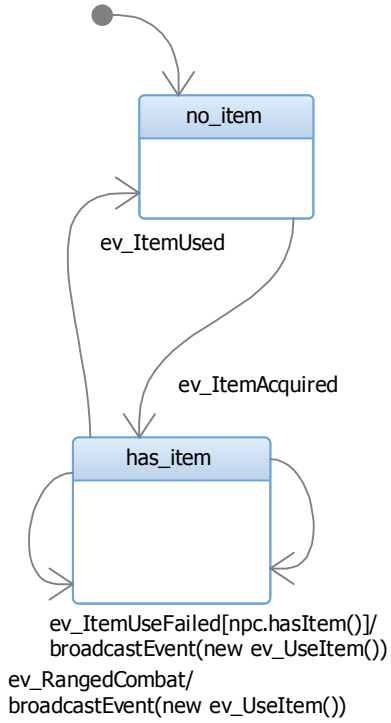


Figure A–35: The *ItemExecutor*.

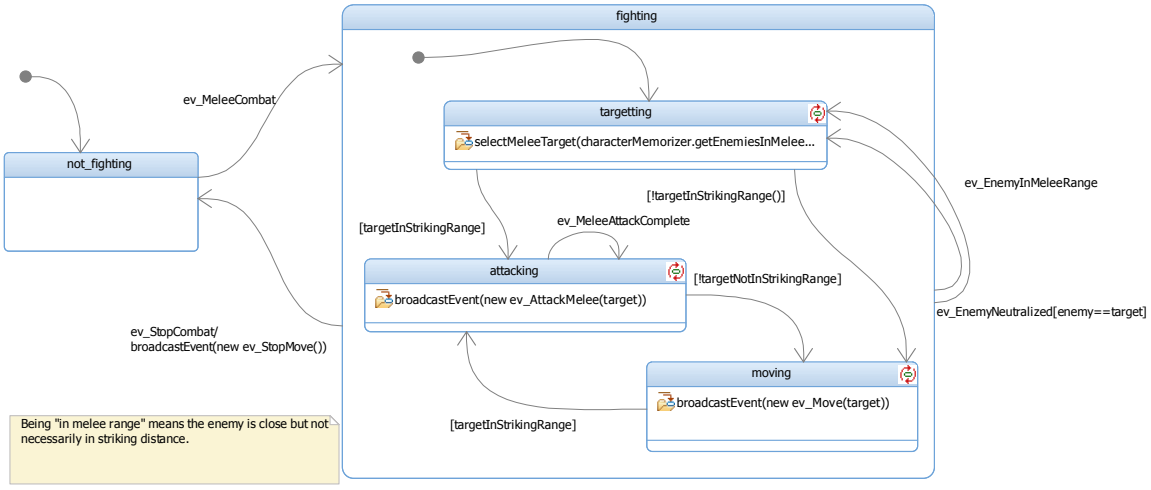


Figure A–36: The *MeleeCombatExecutor*.

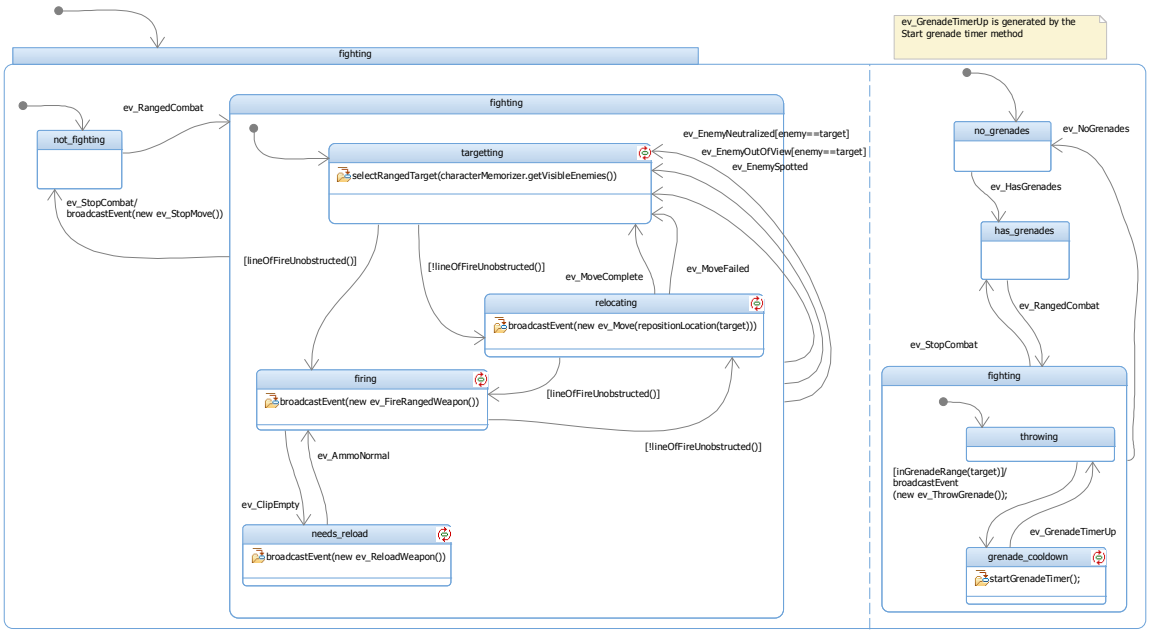


Figure A–37: The *RangedCombatExecutor*.

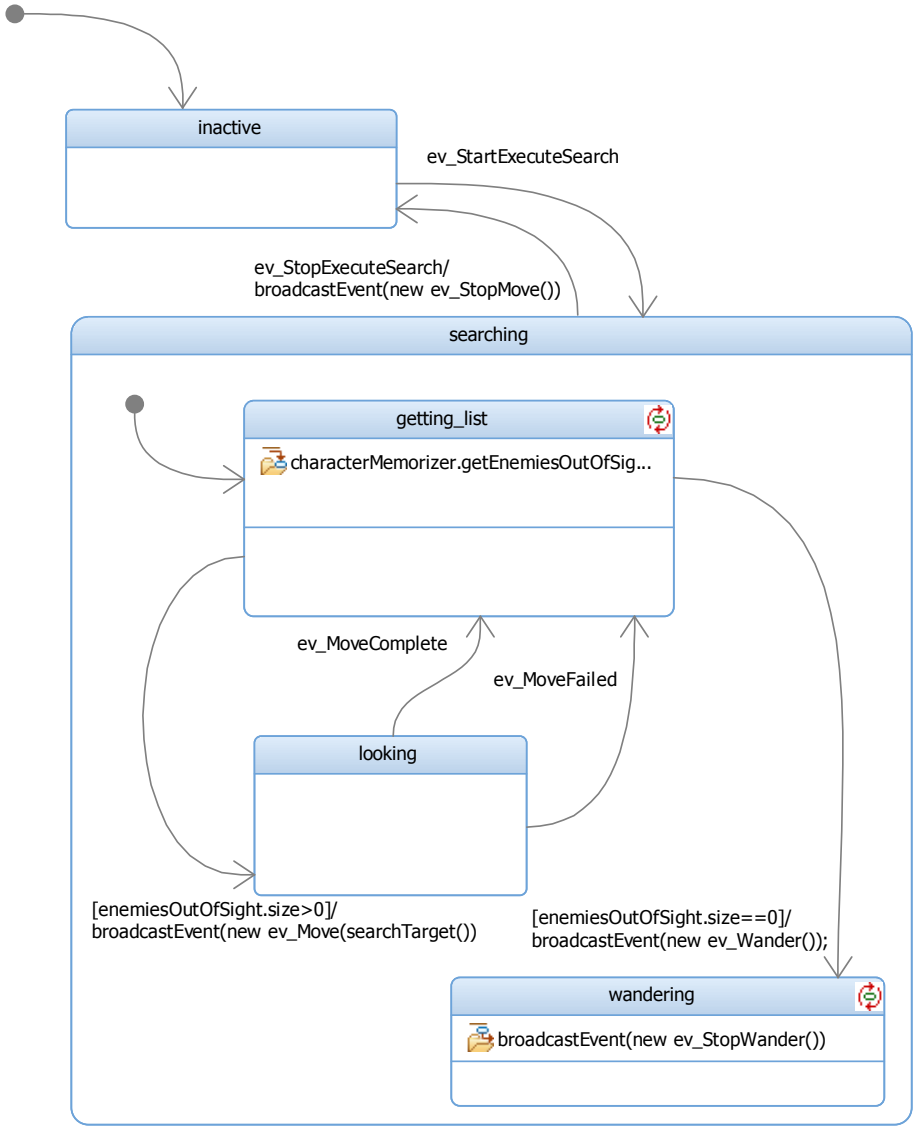


Figure A–38: The *SearchExecutor*.

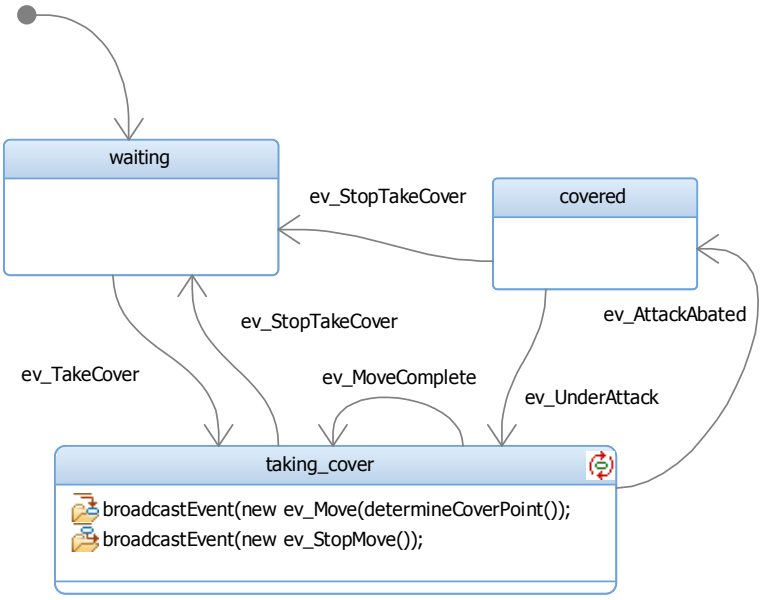


Figure A–39: The *TakeCoverExecutor*.

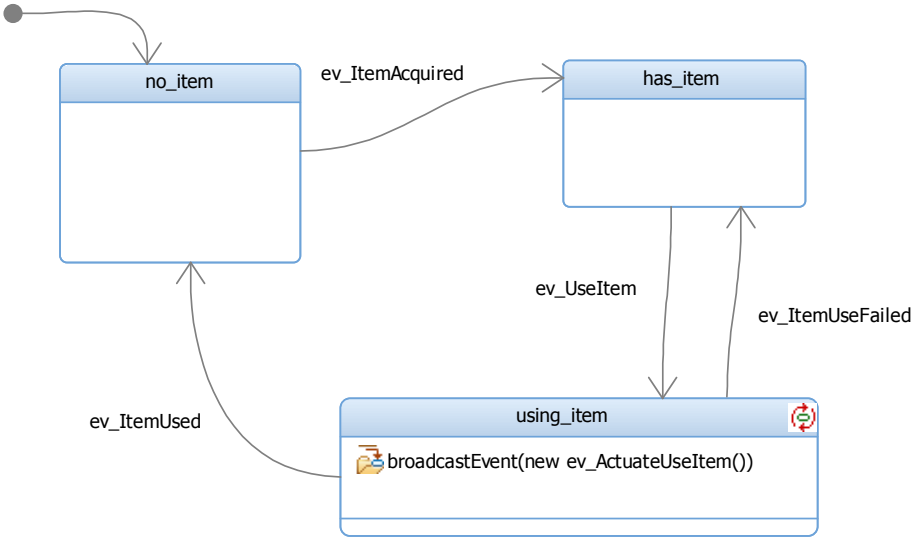


Figure A–40: The *UseItemExecutor*.

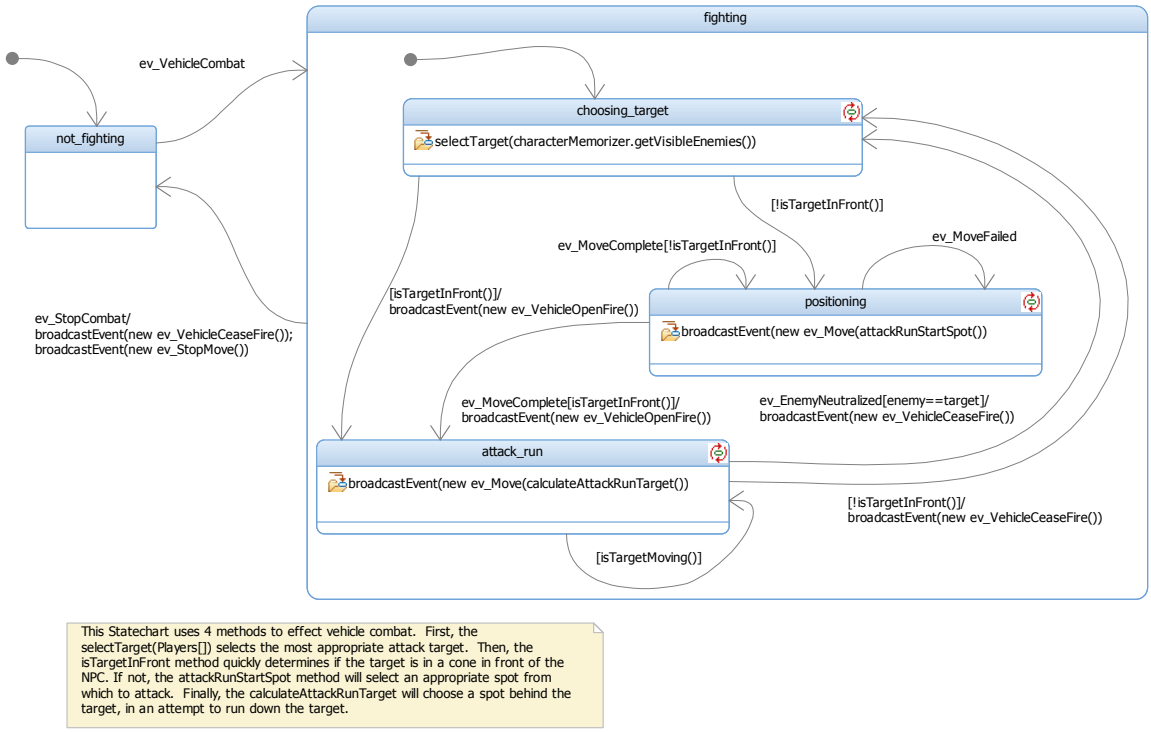


Figure A–41: The *VehicleCombatExecutor*.

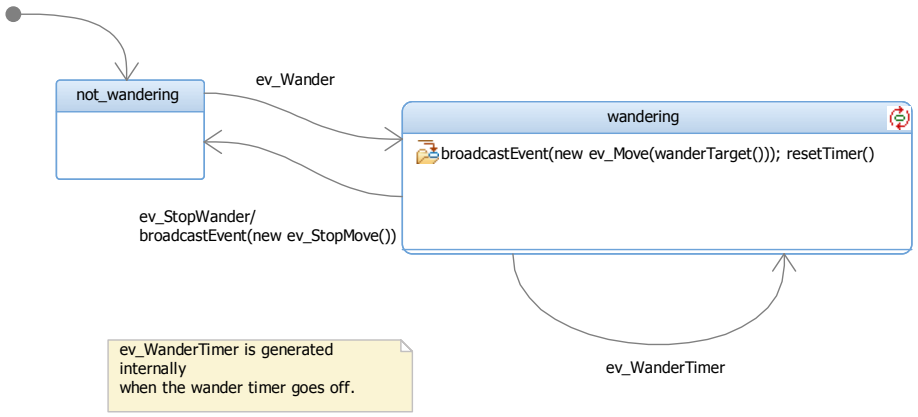


Figure A–42: The *WanderExecutor*.

A.7 Coordinators

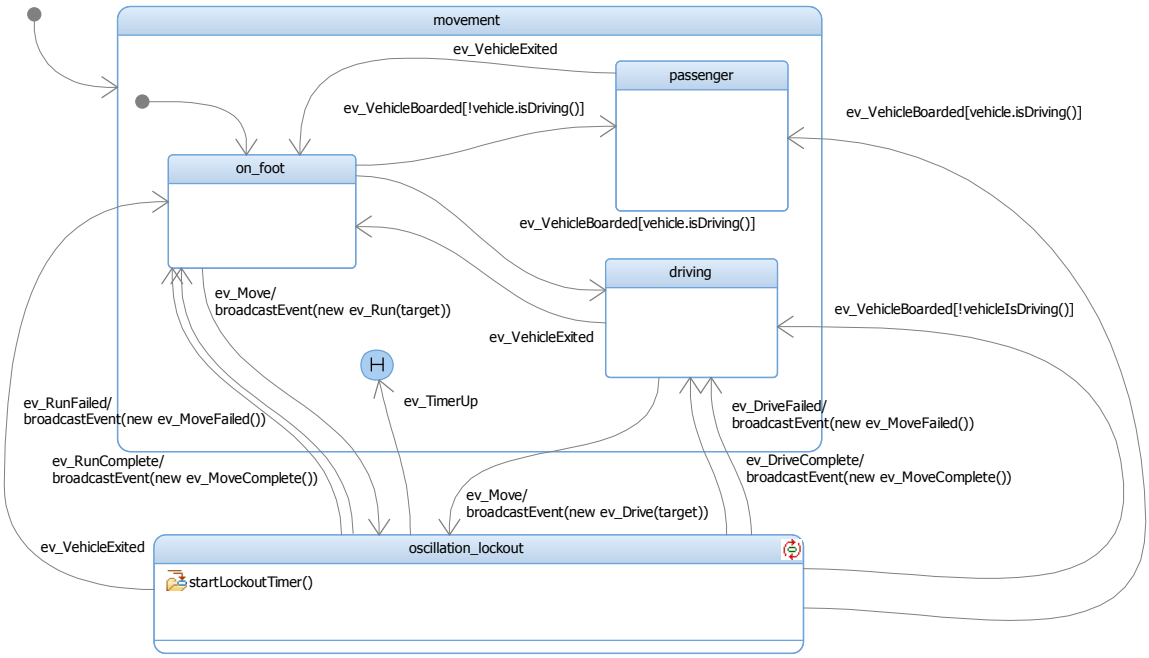


Figure A–43: The *MovementCoordinator*.

A.8 Actuators

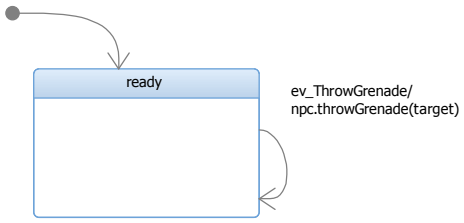


Figure A–44: The *GrenadeActuator*.

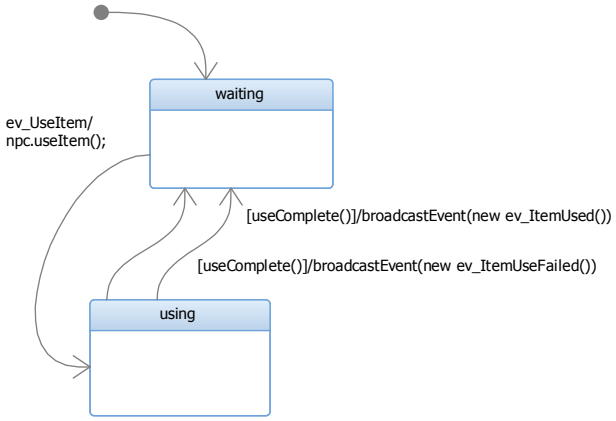


Figure A–45: The *ItemActuator*.

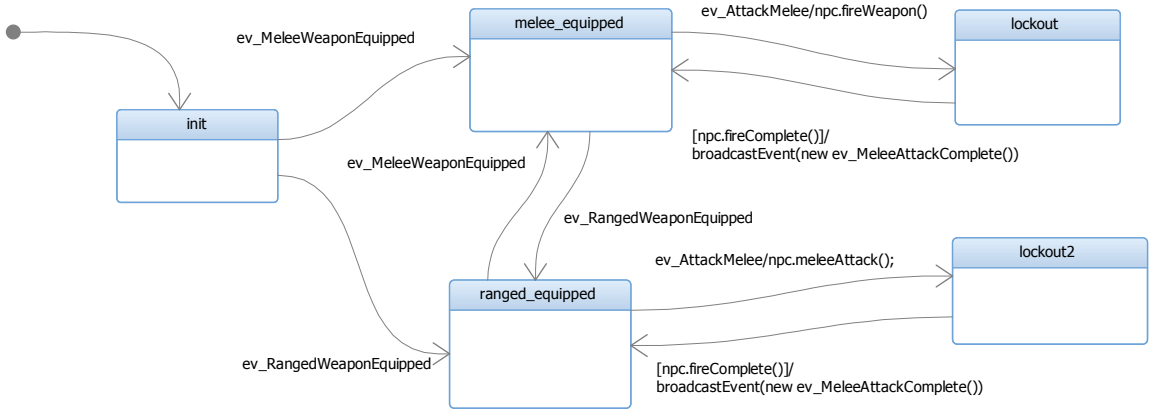


Figure A–46: The *MeleeActuator*.

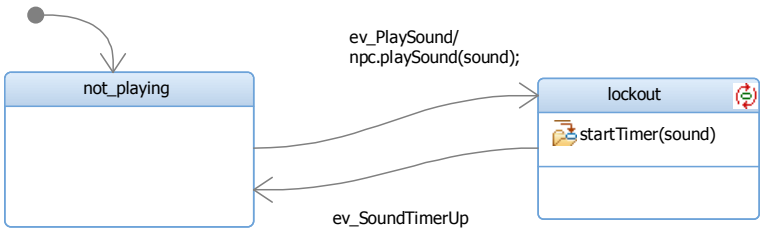


Figure A–47: The *SoundActuator*.

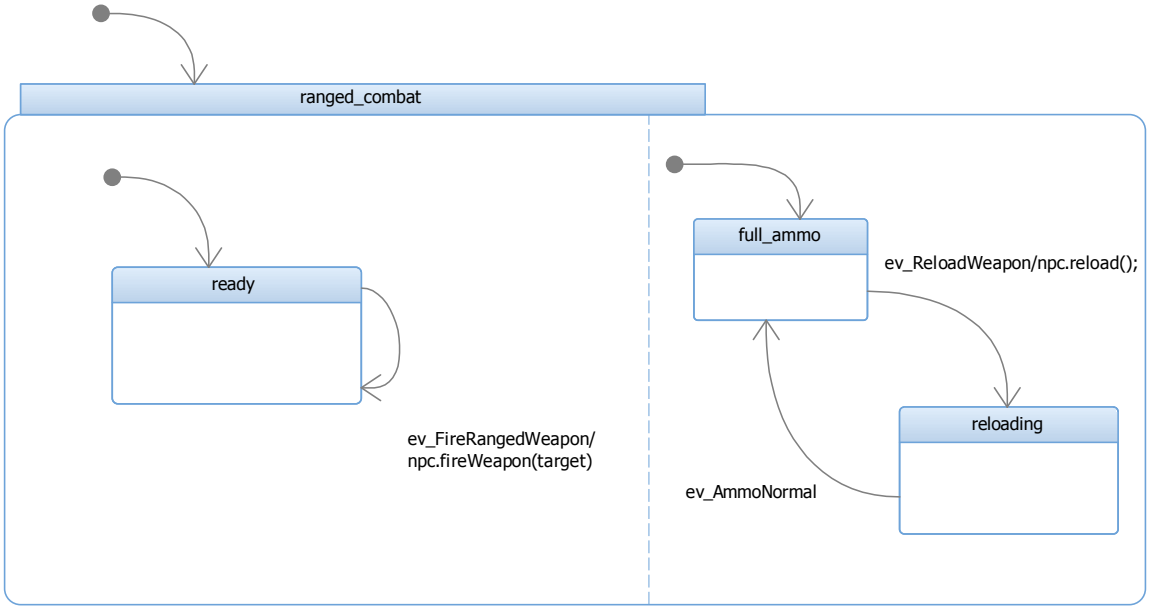


Figure A–48: The *RangedCombatActuator*.

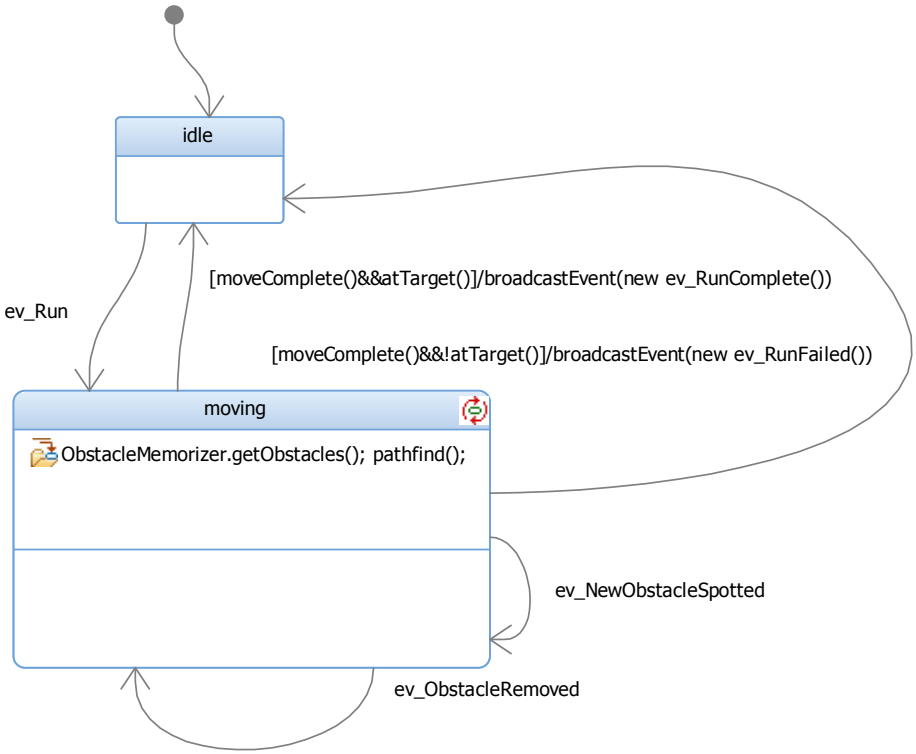


Figure A–49: The *RunActuator*.

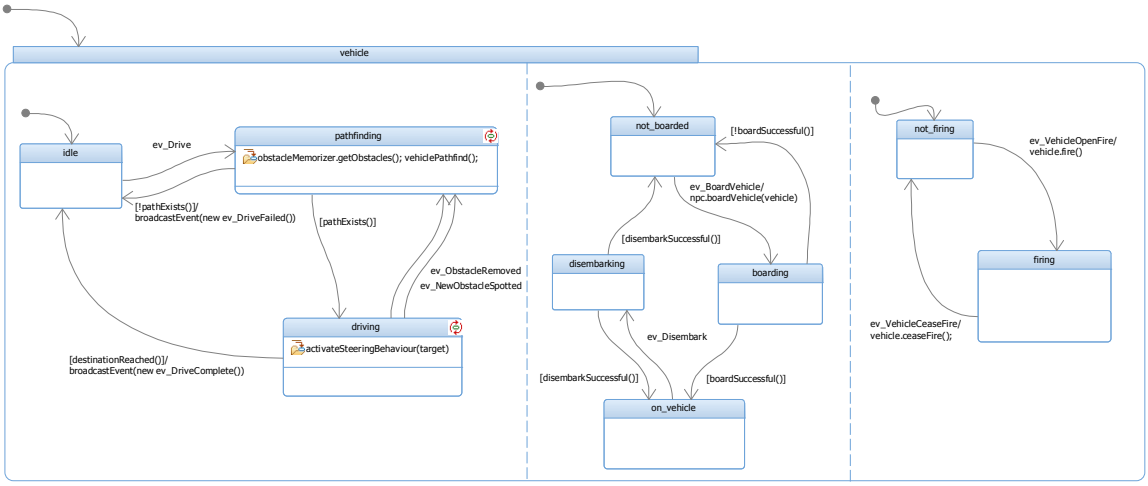


Figure A–50: The *VehicleActuator*.

Appendix B: Promela Representation of the Squirrel AI

The following code is the Promela representation of the Squirrel AI presented in Chap. 4. It was automatically generated by Scythe AI and has not been modified in any way. This was used as the starting point for verification of semantic correctness of the squirrel AI; changes made during verification are not reflected here.

Listing B.1: "SquirrelAI.pml"

```
1 mtype = {pick_up_item, high_energy, no_key_item_visible, start_get_food
      , start_flee, pick_up_successful, energy_changed,
      destination_unreachable, time, i_see_player, i_see_item,
      stop_get_food, execute_stand, high_threat, low_threat,
      i_dont_see_item, threat_changed, move_successful,
      pick_up_item_request, move, stop_move, pick_up_failed, collided,
      no_threat, stop_flee, key_item_visible, path_move, i_dont_see_player
      , stop_wander, very_low_energy, low_energy, eat, move_failed,
      start_wander, destination_reached, null, INITIAL};
2
3 chan AIEventQueue = [255] of {mtype};
4 chan ExternalEventQueue = [1] of {mtype};
5
6 bool externalEventFire=false;
7 bool quiescent=false;
8
9 /* External event guards */
```

```
10 bool exGuard_Move_Actuator_destination_reached = false;
11 bool exGuard_Move_Actuator_destination_unreachable = false;
12 bool exGuard_Move_Actuator_collided = false;
13 bool exGuard_Pickup_Actuator_pick_up_successful = false;
14 bool exGuard_Pickup_Actuator_pick_up_failed = false;
15 bool exGuard_Threat_Analyzer_i_see_player = false;
16 bool exGuard_Threat_Analyzer_i_dont_see_player = false;
17 bool exGuard_Threat_Analyzer_threat_changed = true;
18 bool exGuard_Wander_Executor_time = false;
19 bool exGuard_Eat_Decider_key_item_visible = true;
20 bool exGuard_Eat_Decider_no_key_item_visible = false;
21 bool exGuard_Energy_Sensor_energy_changed = true;
22 bool exGuard_Item_Memorizer_i_see_item = true;
23 bool exGuard_Item_Memorizer_i_dont_see_item = true;
24 bool exGuard_null = true;
25
26 init {
27   /* Launch the Statechart processes */
28   run Move_Actuator();
29   run Eat_Actuator();
30   run Pickup_Actuator();
31   run Pickup_Executor();
32   run Threat_Analyzer();
33   run Squirrel_Brain();
34   run Wander_Executor();
35   run Eat_Decider();
36   run Flee_Decider();
37   run Eat_Analyzer();
```

```
38   run Energy_Sensor();
39   run Item_Memorizer();
40
41   run processEvents();
42
43   /* load up initial Events with any on-entry events from default
       states */
44   AIEventQueue!no_threat;
45   AIEventQueue!start_wander;
46   AIEventQueue!high_energy;
47   /* fake an external event to get AIEventQueue processing*/
48   ExternalEventQueue!INITIAL;
49   externalEventFire=true;
50   quiescent;
51
52   /* now we are in the proper default state. */
53   run externalGenerator();
54 }
55
56
57 proctype externalGenerator() {
58   do
59   :: atomic {
60       quiescent = false;
61       if
62         :: (exGuard_Move_Actuator_destination_reached) ->
     ExternalEventQueue!destination_reached;
```

```
63          :: (exGuard_Move_Actuator_destination_unreachable) ->
    ExternalEventQueue!destination_unreachable;
64          :: (exGuard_Move_Actuator_collided) -> ExternalEventQueue!
    collided;
65          :: (exGuard_Pickup_Actuator_pick_up_successful) ->
    ExternalEventQueue!pick_up_successful;
66          :: (exGuard_Pickup_Actuator_pick_up_failed) ->
    ExternalEventQueue!pick_up_failed;
67          :: (exGuard_Threat_Analyzer_i_see_player) -> ExternalEventQueue
    !i_see_player;
68          :: (exGuard_Threat_Analyzer_i_dont_see_player) ->
    ExternalEventQueue!i_dont_see_player;
69          :: (exGuard_Threat_Analyzer_threat_changed) ->
    ExternalEventQueue!threat_changed;
70          :: (exGuard_Wander_Executor_time) -> ExternalEventQueue!time;
71          :: (exGuard_Eat_Decider_key_item_visible) -> ExternalEventQueue
    !key_item_visible;
72          :: (exGuard_Eat_Decider_no_key_item_visible) ->
    ExternalEventQueue!no_key_item_visible;
73          :: (exGuard_Energy_Sensor_energy_changed) -> ExternalEventQueue
    !energy_changed;
74          :: (exGuard_Item_Memorizer_i_see_item) -> ExternalEventQueue!
    i_see_item;
75          :: (exGuard_Item_Memorizer_i_dont_see_item) ->
    ExternalEventQueue!i_dont_see_item;
76          :: (exGuard_null) -> ExternalEventQueue!null;
77        fi;
78        externalEventFire = true;
```

```
79      }
80      quiescent;
81    od;
82  }
83
84  /****** Statechart processes ******/
85
86  byte Move_ActuatorState = 0;
87  bool Move_ActuatorStart = false;
88  bool Move_ActuatorCallback = false;
89  proctype Move_Actuator() {
90    mtype currentEvent;
91    do
92      :: Move_ActuatorStart -> atomic {
93          AIEventQueue?<currentEvent>;
94          if
95            :: (Move_ActuatorState == 0) ->
96              if
97                :: (currentEvent == move) -> Move_ActuatorState = 1;
      exGuard_Move_Actuator_destination_reached = true;
      exGuard_Move_Actuator_destination_unreachable = true;
      exGuard_Move_Actuator_collided = true;
98                :: (currentEvent == path_move) -> Move_ActuatorState = 1;
       exGuard_Move_Actuator_destination_reached = true;
      exGuard_Move_Actuator_destination_unreachable = true;
      exGuard_Move_Actuator_collided = true;
99                :: (currentEvent == execute_stand) -> Move_ActuatorState
     = 0;
```

```
100             :: !( currentEvent == move || currentEvent == path_move
    || currentEvent == execute_stand ) -> skip;
101           fi;
102         :: (Move_ActuatorState == 1) ->
103           if
104             :: (currentEvent == destination_reached) ->
    Move_ActuatorState = 0; exGuard_Move_Actuator_destination_reached =
    false; exGuard_Move_Actuator_destination_unreachable = false;
    exGuard_Move_Actuator_collided = false;
105             :: (currentEvent == stop_move) -> Move_ActuatorState = 0;
     exGuard_Move_Actuator_destination_reached = false;
    exGuard_Move_Actuator_destination_unreachable = false;
    exGuard_Move_Actuator_collided = false;
106             :: (currentEvent == destination_unreachable) ->
    Move_ActuatorState = 0; exGuard_Move_Actuator_destination_reached =
    false; exGuard_Move_Actuator_destination_unreachable = false;
    exGuard_Move_Actuator_collided = false;
107             :: (currentEvent == collided) -> Move_ActuatorState = 0;
    exGuard_Move_Actuator_destination_reached = false;
    exGuard_Move_Actuator_destination_unreachable = false;
    exGuard_Move_Actuator_collided = false;
108             :: !( currentEvent == destination_reached || currentEvent
     == stop_move || currentEvent == destination_unreachable ||
    currentEvent == collided ) -> skip;
109           fi;
110       fi;
111       Move_ActuatorStart = false;
112       Move_ActuatorCallback = true;
```

```
113       } /* end atomic */
114   od;
115 }
116
117 byte Eat_ActuatorState = 0;
118 bool Eat_ActuatorStart = false;
119 bool Eat_ActuatorCallback = false;
120 proctype Eat_Actuator() {
121   mtype currentEvent;
122   do
123     :: Eat_ActuatorStart -> atomic {
124         AIEventQueue?<currentEvent>;
125         if
126           :: (Eat_ActuatorState == 0) ->
127             if
128               :: (currentEvent == eat) -> Eat_ActuatorState = 0;
129               :: !( currentEvent == eat ) -> skip;
130             fi;
131         fi;
132         Eat_ActuatorStart = false;
133         Eat_ActuatorCallback = true;
134       } /* end atomic */
135   od;
136 }
137
138 byte Pickup_ActuatorState = 0;
139 bool Pickup_ActuatorStart = false;
140 bool Pickup_ActuatorCallback = false;
```

```
141 proctype Pickup_Actuator() {
142   mtype currentEvent;
143   do
144     :: Pickup_ActuatorStart -> atomic {
145         AIEventQueue?<currentEvent>;
146         if
147           :: (Pickup_ActuatorState == 0) ->
148             if
149               :: (currentEvent == pick_up_item) -> Pickup_ActuatorState
      = 1; exGuard_Pickup_Actuator_pick_up_successful = true;
      exGuard_Pickup_Actuator_pick_up_failed = true;
150               :: !( currentEvent == pick_up_item ) -> skip;
151             fi;
152           :: (Pickup_ActuatorState == 1) ->
153             if
154               :: (currentEvent == pick_up_successful) ->
      Pickup_ActuatorState = 0; exGuard_Pickup_Actuator_pick_up_successful
      = false; exGuard_Pickup_Actuator_pick_up_failed = false;
155               :: (currentEvent == pick_up_failed) ->
      Pickup_ActuatorState = 0; exGuard_Pickup_Actuator_pick_up_successful
      = false; exGuard_Pickup_Actuator_pick_up_failed = false;
156               :: !( currentEvent == pick_up_successful || currentEvent
      == pick_up_failed ) -> skip;
157             fi;
158         fi;
159         Pickup_ActuatorStart = false;
160         Pickup_ActuatorCallback = true;
161       } /* end atomic */
```

```
162   od;
163 }
164
165 byte Pickup_ExecutorState = 0;
166 bool Pickup_ExecutorStart = false;
167 bool Pickup_ExecutorCallback = false;
168 proctype Pickup_Executor() {
169   mtype currentEvent;
170   do
171     :: Pickup_ExecutorStart -> atomic {
172         AIEventQueue?<currentEvent>;
173         if
174           :: (Pickup_ExecutorState == 0) ->
175             if
176               :: (currentEvent == pick_up_item_request) ->
      Pickup_ExecutorState = 1;
177               :: !( currentEvent == pick_up_item_request ) -> skip;
178             fi;
179           :: (Pickup_ExecutorState == 1) ->
180             if
181               :: true -> Pickup_ExecutorState = 0; AIEventQueue!
      pick_up_item;
182               :: true -> Pickup_ExecutorState = 2; AIEventQueue!
      stop_move; AIEventQueue!move;
183             fi;
184           :: (Pickup_ExecutorState == 2) ->
185             if
```

```
186                 :: (currentEvent == move_successful) ->
      Pickup_ExecutorState = 0; AIEventQueue!pick_up_item;
187                 :: (currentEvent == move_failed) -> Pickup_ExecutorState
      = 0; AIEventQueue!pick_up_failed;
188                 :: !( currentEvent == move_successful || currentEvent ==
      move_failed ) -> skip;
189               fi;
190           fi;
191           Pickup_ExecutorStart = false;
192           Pickup_ExecutorCallback = true;
193         } /* end atomic */
194   od;
195 }
196
197 byte Threat_AnalyzerState = 1;
198 bool Threat_AnalyzerStart = false;
199 bool Threat_AnalyzerCallback = false;
200 proctype Threat_Analyzer() {
201   mtype currentEvent;
202   do
203     :: Threat_AnalyzerStart -> atomic {
204           AIEventQueue?<currentEvent>;
205           if
206             :: (Threat_AnalyzerState == 1) ->
207               if
```

```
208             :: (currentEvent == threat_changed) ->
    Threat_AnalyzerState = 2; AIEventQueue!low_threat;
    exGuard_Threat_Analyzer_threat_changed = false;
    exGuard_Threat_Analyzer_threat_changed = false;
    exGuard_Threat_Analyzer_threat_changed = true;
    exGuard_Threat_Analyzer_threat_changed = true;
209             :: (currentEvent == threat_changed) -> skip;
210             :: (currentEvent == threat_changed) ->
    Threat_AnalyzerState = 3; AIEventQueue!high_threat;
    exGuard_Threat_Analyzer_threat_changed = false;
    exGuard_Threat_Analyzer_threat_changed = false;
    exGuard_Threat_Analyzer_threat_changed = true;
    exGuard_Threat_Analyzer_threat_changed = true;
211             :: (currentEvent == threat_changed) -> skip;
212             :: !( currentEvent == threat_changed || currentEvent ==
    threat_changed ) -> skip;
213           fi;
214         :: (Threat_AnalyzerState == 2) ->
215           if
216             :: (currentEvent == threat_changed) ->
    Threat_AnalyzerState = 1; AIEventQueue!no_threat;
    exGuard_Threat_Analyzer_threat_changed = false;
    exGuard_Threat_Analyzer_threat_changed = false;
    exGuard_Threat_Analyzer_threat_changed = true;
    exGuard_Threat_Analyzer_threat_changed = true;
217             :: (currentEvent == threat_changed) -> skip;
```

```
218             :: (currentEvent == threat_changed) ->
    Threat_AnalyzerState = 3; AIEventQueue!high_threat;
    exGuard_Threat_Analyzer_threat_changed = false;
    exGuard_Threat_Analyzer_threat_changed = false;
    exGuard_Threat_Analyzer_threat_changed = true;
    exGuard_Threat_Analyzer_threat_changed = true;
219             :: (currentEvent == threat_changed) -> skip;
220             :: !( currentEvent == threat_changed || currentEvent ==
    threat_changed ) -> skip;
221           fi;
222         :: (Threat_AnalyzerState == 3) ->
223           if
224             :: (currentEvent == threat_changed) ->
    Threat_AnalyzerState = 1; AIEventQueue!no_threat;
    exGuard_Threat_Analyzer_threat_changed = false;
    exGuard_Threat_Analyzer_threat_changed = false;
    exGuard_Threat_Analyzer_threat_changed = true;
    exGuard_Threat_Analyzer_threat_changed = true;
225             :: (currentEvent == threat_changed) -> skip;
226             :: (currentEvent == threat_changed) ->
    Threat_AnalyzerState = 2; AIEventQueue!low_threat;
    exGuard_Threat_Analyzer_threat_changed = false;
    exGuard_Threat_Analyzer_threat_changed = false;
    exGuard_Threat_Analyzer_threat_changed = true;
    exGuard_Threat_Analyzer_threat_changed = true;
227             :: (currentEvent == threat_changed) -> skip;
228             :: !( currentEvent == threat_changed || currentEvent ==
    threat_changed ) -> skip;
```

```
229             fi;
230         fi;
231         Threat_AnalyzerStart = false;
232         Threat_AnalyzerCallback = true;
233       } /* end atomic */
234   od;
235 }
236
237 byte Squirrel_BrainState = 2;
238 bool Squirrel_BrainStart = false;
239 bool Squirrel_BrainCallback = false;
240 int Squirrel_BrainHistory = 0;
241 proctype Squirrel_Brain() {
242   mtype currentEvent;
243   do
244     :: Squirrel_BrainStart -> atomic {
245         AIEventQueue?<currentEvent>;
246         if
247           :: (Squirrel_BrainState == 0) ->
248             if
249               :: (currentEvent == high_threat) -> Squirrel_BrainState =
      5; AIEventQueue!start_flee;
250               :: !( currentEvent == high_threat ) -> skip;
251             fi;
252           :: (Squirrel_BrainState == 1) ->
253             if
254               :: (currentEvent == very_low_energy) ->
     Squirrel_BrainState = 4; AIEventQueue!start_get_food;
```

```
255             :: (currentEvent == low_threat) -> Squirrel_BrainState =
    5; AIEventQueue!start_flee;
256             :: (currentEvent == high_threat) -> Squirrel_BrainState =
     5; AIEventQueue!start_flee;
257             :: !( currentEvent == very_low_energy || currentEvent ==
    low_threat || currentEvent == high_threat ) -> skip;
258           fi;
259         :: (Squirrel_BrainState == 2) ->
260           if
261             :: (currentEvent == low_energy) -> Squirrel_BrainState =
    3; AIEventQueue!stop_wander; AIEventQueue!start_get_food;
    Squirrel_BrainHistory = 2;
262             :: (currentEvent == very_low_energy) ->
    Squirrel_BrainState = 4; AIEventQueue!start_get_food; AIEventQueue!
    stop_wander; Squirrel_BrainHistory = 2;
263             :: (currentEvent == low_threat) -> Squirrel_BrainState =
    5; AIEventQueue!start_flee; AIEventQueue!stop_wander;
    Squirrel_BrainHistory = 2;
264             :: (currentEvent == high_threat) -> Squirrel_BrainState =
     5; AIEventQueue!start_flee; AIEventQueue!stop_wander;
    Squirrel_BrainHistory = 2;
265             :: !( currentEvent == low_energy || currentEvent ==
    very_low_energy || currentEvent == low_threat || currentEvent ==
    high_threat ) -> skip;
266           fi;
267         :: (Squirrel_BrainState == 3) ->
268           if
```

```
269             :: (currentEvent == pick_up_successful) ->
    Squirrel_BrainState = 2; AIEventQueue!stop_get_food; AIEventQueue!
    start_wander; AIEventQueue!eat; Squirrel_BrainHistory = 3;
270             :: (currentEvent == very_low_energy) ->
    Squirrel_BrainState = 4; AIEventQueue!start_get_food; AIEventQueue!
    stop_get_food; Squirrel_BrainHistory = 3;
271             :: (currentEvent == low_threat) -> Squirrel_BrainState =
    5; AIEventQueue!start_flee; AIEventQueue!stop_get_food;
    Squirrel_BrainHistory = 3;
272             :: (currentEvent == high_threat) -> Squirrel_BrainState =
     5; AIEventQueue!start_flee; AIEventQueue!stop_get_food;
    Squirrel_BrainHistory = 3;
273             :: !( currentEvent == pick_up_successful || currentEvent
   == very_low_energy || currentEvent == low_threat || currentEvent ==
    high_threat ) -> skip;
274           fi;
275         :: (Squirrel_BrainState == 4) ->
276           if
277             :: (currentEvent == pick_up_successful) ->
    Squirrel_BrainState = 2; AIEventQueue!stop_get_food; AIEventQueue!
    start_wander; AIEventQueue!eat; Squirrel_BrainHistory = 4;
278             :: (currentEvent == high_threat) -> Squirrel_BrainState =
     5; AIEventQueue!start_flee; AIEventQueue!stop_get_food;
    Squirrel_BrainHistory = 4;
279             :: !( currentEvent == pick_up_successful || currentEvent
   == high_threat ) -> skip;
280           fi;
281         :: (Squirrel_BrainState == 5) ->
```

```
282             if
283               :: (currentEvent == no_threat &&Squirrel_BrainHistory ==
      2) -> Squirrel_BrainState = 2; AIEventQueue!stop_flee; AIEventQueue!
      start_wander;
284               :: (currentEvent == no_threat &&Squirrel_BrainHistory ==
      3) -> Squirrel_BrainState = 3; AIEventQueue!stop_flee; AIEventQueue!
      start_get_food;
285               :: (currentEvent == no_threat &&Squirrel_BrainHistory ==
      4) -> Squirrel_BrainState = 4; AIEventQueue!stop_flee; AIEventQueue!
      start_get_food;
286               :: !( currentEvent == no_threat || currentEvent ==
      no_threat || currentEvent == no_threat ) -> skip;
287             fi;
288         fi;
289         Squirrel_BrainStart = false;
290         Squirrel_BrainCallback = true;
291       } /* end atomic */
292   od;
293 }
294
295 byte Wander_ExecutorState = 0;
296 bool Wander_ExecutorStart = false;
297 bool Wander_ExecutorCallback = false;
298 proctype Wander_Executor() {
299   mtype currentEvent;
300   do
301     :: Wander_ExecutorStart -> atomic {
302         AIEventQueue?<currentEvent>;
```

```
303         if
304           :: (Wander_ExecutorState == 0) ->
305             if
306               :: (currentEvent == start_wander) -> Wander_ExecutorState
      = 2; AIEventQueue!move;
307               :: !( currentEvent == start_wander ) -> skip;
308             fi;
309           :: (Wander_ExecutorState == 1) ->
310             if
311               :: (currentEvent == stop_wander) -> Wander_ExecutorState
     = 0; AIEventQueue!stop_move;
312               :: !( currentEvent == stop_wander ) -> skip;
313             fi;
314           :: (Wander_ExecutorState == 2) ->
315             if
316               :: (currentEvent == null) -> Wander_ExecutorState = 3;
     AIEventQueue!stop_move;
317               :: (currentEvent == null) -> skip;
318               :: (currentEvent == stop_wander) -> Wander_ExecutorState
     = 0; AIEventQueue!stop_move;
319               :: !( currentEvent == stop_wander ) -> skip;
320             fi;
321           :: (Wander_ExecutorState == 3) ->
322             if
323               :: (currentEvent == null) -> Wander_ExecutorState = 2;
     AIEventQueue!move;
324               :: (currentEvent == null) -> skip;
```

```
325                 :: (currentEvent == stop_wander) -> Wander_ExecutorState
    = 0; AIEventQueue!stop_move;
326                 :: !( currentEvent == stop_wander ) -> skip;
327             fi;
328         fi;
329         Wander_ExecutorStart = false;
330         Wander_ExecutorCallback = true;
331       } /* end atomic */
332   od;
333 }
334
335 byte Eat_DeciderState = 0;
336 bool Eat_DeciderStart = false;
337 bool Eat_DeciderCallback = false;
338 proctype Eat_Decider() {
339   mtype currentEvent;
340   do
341     :: Eat_DeciderStart -> atomic {
342         AIEventQueue?<currentEvent>;
343         if
344           :: (Eat_DeciderState == 0) ->
345             if
346               :: (currentEvent == key_item_visible) -> Eat_DeciderState
    = 1; exGuard_Eat_Decider_key_item_visible = false;
    exGuard_Eat_Decider_no_key_item_visible = true;
347               :: (currentEvent == start_get_food) -> Eat_DeciderState =
    2; AIEventQueue!start_wander; exGuard_Eat_Decider_key_item_visible
    = false; exGuard_Eat_Decider_key_item_visible = true;
```

```
348             :: !( currentEvent == key_item_visible || currentEvent ==
    start_get_food ) -> skip;
349           fi;
350         :: (Eat_DeciderState == 1) ->
351           if
352             :: (currentEvent == no_key_item_visible) ->
    Eat_DeciderState = 0; exGuard_Eat_Decider_no_key_item_visible =
    false; exGuard_Eat_Decider_key_item_visible = true;
353             :: (currentEvent == start_get_food) -> Eat_DeciderState =
    3; AIEventQueue!pick_up_item_request;
    exGuard_Eat_Decider_no_key_item_visible = false;
    exGuard_Eat_Decider_no_key_item_visible = true;
354             :: !( currentEvent == no_key_item_visible || currentEvent
    == start_get_food ) -> skip;
355           fi;
356         :: (Eat_DeciderState == 2) ->
357           if
358             :: (currentEvent == key_item_visible) -> Eat_DeciderState
    = 3; AIEventQueue!stop_wander; AIEventQueue!pick_up_item_request;
    exGuard_Eat_Decider_key_item_visible = false;
    exGuard_Eat_Decider_no_key_item_visible = true;
359             :: (currentEvent == stop_get_food) -> Eat_DeciderState =
    0; AIEventQueue!stop_wander; exGuard_Eat_Decider_key_item_visible =
    false; exGuard_Eat_Decider_key_item_visible = true;
360             :: !( currentEvent == key_item_visible || currentEvent ==
    stop_get_food ) -> skip;
361           fi;
362         :: (Eat_DeciderState == 3) ->
```

```
363             if
364               :: (currentEvent == pick_up_failed) -> Eat_DeciderState =
     3; AIEventQueue!pick_up_item_request;
    exGuard_Eat_Decider_no_key_item_visible = false;
    exGuard_Eat_Decider_no_key_item_visible = true;
365               :: (currentEvent == pick_up_successful) ->
    Eat_DeciderState = 1; exGuard_Eat_Decider_no_key_item_visible =
    false; exGuard_Eat_Decider_no_key_item_visible = true;
366               :: (currentEvent == pick_up_successful) -> skip;
367               :: (currentEvent == pick_up_successful) ->
    Eat_DeciderState = 0; exGuard_Eat_Decider_no_key_item_visible =
    false; exGuard_Eat_Decider_key_item_visible = true;
368               :: (currentEvent == pick_up_successful) -> skip;
369               :: (currentEvent == no_key_item_visible) ->
    Eat_DeciderState = 2; AIEventQueue!start_wander;
    exGuard_Eat_Decider_no_key_item_visible = false;
    exGuard_Eat_Decider_key_item_visible = true;
370               :: (currentEvent == stop_get_food) -> Eat_DeciderState =
    1; exGuard_Eat_Decider_no_key_item_visible = false;
    exGuard_Eat_Decider_no_key_item_visible = true;
371               :: !( currentEvent == pick_up_failed || currentEvent ==
    pick_up_successful || currentEvent == pick_up_successful ||
    currentEvent == no_key_item_visible || currentEvent == stop_get_food
     ) -> skip;
372             fi;
373         fi;
374         Eat_DeciderStart = false;
375         Eat_DeciderCallback = true;
```

```
376       } /* end atomic */
377   od;
378 }
379
380 byte Flee_DeciderState = 0;
381 bool Flee_DeciderStart = false;
382 bool Flee_DeciderCallback = false;
383 proctype Flee_Decider() {
384   mtype currentEvent;
385   do
386     :: Flee_DeciderStart -> atomic {
387         AIEventQueue?<currentEvent>;
388         if
389           :: (Flee_DeciderState == 0) ->
390             if
391               :: (currentEvent == low_threat) -> Flee_DeciderState = 1;
392               :: (currentEvent == high_threat) -> Flee_DeciderState =
      2;
393               :: !( currentEvent == low_threat || currentEvent ==
      high_threat ) -> skip;
394             fi;
395           :: (Flee_DeciderState == 1) ->
396             if
397               :: (currentEvent == no_threat) -> Flee_DeciderState = 0;
398               :: (currentEvent == high_threat) -> Flee_DeciderState =
      2;
399               :: (currentEvent == start_flee) -> Flee_DeciderState = 4;
       AIEventQueue!stop_move; AIEventQueue!path_move;
```

```
400             :: !( currentEvent == no_threat || currentEvent ==
    high_threat || currentEvent == start_flee ) -> skip;
401           fi;
402         :: (Flee_DeciderState == 2) ->
403           if
404             :: (currentEvent == no_threat) -> Flee_DeciderState = 0;
405             :: (currentEvent == low_threat) -> Flee_DeciderState = 1;
406             :: (currentEvent == start_flee) -> Flee_DeciderState = 5;
     AIEventQueue!stop_move; AIEventQueue!path_move;
407             :: !( currentEvent == no_threat || currentEvent ==
    low_threat || currentEvent == start_flee ) -> skip;
408           fi;
409         :: (Flee_DeciderState == 3) ->
410           if
411             :: (currentEvent == stop_flee) -> Flee_DeciderState = 0;
    AIEventQueue!stop_move;
412             :: !( currentEvent == stop_flee ) -> skip;
413           fi;
414         :: (Flee_DeciderState == 4) ->
415           if
416             :: (currentEvent == high_threat) -> Flee_DeciderState =
    5; AIEventQueue!stop_move; AIEventQueue!path_move;
417             :: (currentEvent == move_successful) -> Flee_DeciderState
     = 4; AIEventQueue!stop_move; AIEventQueue!path_move;
418             :: (currentEvent == move_failed) -> Flee_DeciderState =
    4; AIEventQueue!stop_move; AIEventQueue!path_move;
419             :: (currentEvent == stop_flee) -> Flee_DeciderState = 0;
```

```
420                 :: !( currentEvent == high_threat || currentEvent ==
    move_successful || currentEvent == move_failed || currentEvent ==
    stop_flee ) -> skip;
421             fi;
422           :: (Flee_DeciderState == 5) ->
423             if
424               :: (currentEvent == low_threat) -> Flee_DeciderState = 4;
     AIEventQueue!stop_move; AIEventQueue!path_move;
425               :: (currentEvent == move_successful) -> Flee_DeciderState
    = 5; AIEventQueue!stop_move; AIEventQueue!path_move;
426               :: (currentEvent == move_failed) -> Flee_DeciderState =
    5; AIEventQueue!stop_move; AIEventQueue!path_move;
427               :: (currentEvent == stop_flee) -> Flee_DeciderState = 0;
    AIEventQueue!stop_move;
428               :: !( currentEvent == low_threat || currentEvent ==
    move_successful || currentEvent == move_failed || currentEvent ==
    stop_flee ) -> skip;
429             fi;
430         fi;
431         Flee_DeciderStart = false;
432         Flee_DeciderCallback = true;
433       } /* end atomic */
434   od;
435 }
436
437 byte Eat_AnalyzerState = 0;
438 bool Eat_AnalyzerStart = false;
439 bool Eat_AnalyzerCallback = false;
```

```
440 proctype Eat_Analyzer() {
441   mtype currentEvent;
442   do
443     :: Eat_AnalyzerStart -> atomic {
444         AIEventQueue?<currentEvent>;
445         if
446           :: (Eat_AnalyzerState == 0) ->
447             if
448               :: (currentEvent == pick_up_successful) ->
      Eat_AnalyzerState = 1;
449               :: (currentEvent == very_low_energy) -> Eat_AnalyzerState
      = 2;
450               :: !( currentEvent == pick_up_successful || currentEvent
     == very_low_energy ) -> skip;
451             fi;
452           :: (Eat_AnalyzerState == 1) ->
453             if
454               :: (currentEvent == very_low_energy) -> Eat_AnalyzerState
      = 0; AIEventQueue!eat;
455               :: !( currentEvent == very_low_energy ) -> skip;
456             fi;
457           :: (Eat_AnalyzerState == 2) ->
458             if
459               :: (currentEvent == pick_up_successful) ->
      Eat_AnalyzerState = 0; AIEventQueue!eat;
460               :: !( currentEvent == pick_up_successful ) -> skip;
461             fi;
462         fi;
```

```
463           Eat_AnalyzerStart = false;
464           Eat_AnalyzerCallback = true;
465         } /* end atomic */
466   od;
467 }
468
469 byte Energy_SensorState = 0;
470 bool Energy_SensorStart = false;
471 bool Energy_SensorCallback = false;
472 proctype Energy_Sensor() {
473   mtype currentEvent;
474   do
475     :: Energy_SensorStart -> atomic {
476          AIEventQueue?<currentEvent>;
477          if
478            :: (Energy_SensorState == 0) ->
479              if
480                :: (currentEvent == energy_changed) -> Energy_SensorState
       = 1; AIEventQueue!low_energy; exGuard_Energy_Sensor_energy_changed
      = false; exGuard_Energy_Sensor_energy_changed = false;
      exGuard_Energy_Sensor_energy_changed = true;
      exGuard_Energy_Sensor_energy_changed = true;
481                :: (currentEvent == energy_changed) -> skip;
```

```
482             :: (currentEvent == energy_changed) -> Energy_SensorState
    = 2; AIEventQueue!very_low_energy;
    exGuard_Energy_Sensor_energy_changed = false;
    exGuard_Energy_Sensor_energy_changed = false;
    exGuard_Energy_Sensor_energy_changed = true;
    exGuard_Energy_Sensor_energy_changed = true;
483             :: (currentEvent == energy_changed) -> skip;
484             :: !( currentEvent == energy_changed || currentEvent ==
    energy_changed ) -> skip;
485           fi;
486         :: (Energy_SensorState == 1) ->
487           if
488             :: (currentEvent == energy_changed) -> Energy_SensorState
    = 0; AIEventQueue!high_energy; exGuard_Energy_Sensor_energy_changed
    = false; exGuard_Energy_Sensor_energy_changed = false;
    exGuard_Energy_Sensor_energy_changed = true;
    exGuard_Energy_Sensor_energy_changed = true;
489             :: (currentEvent == energy_changed) -> skip;
490             :: (currentEvent == energy_changed) -> Energy_SensorState
    = 2; AIEventQueue!very_low_energy;
    exGuard_Energy_Sensor_energy_changed = false;
    exGuard_Energy_Sensor_energy_changed = false;
    exGuard_Energy_Sensor_energy_changed = true;
    exGuard_Energy_Sensor_energy_changed = true;
491             :: (currentEvent == energy_changed) -> skip;
492             :: !( currentEvent == energy_changed || currentEvent ==
    energy_changed ) -> skip;
493           fi;
```

```
494             :: (Energy_SensorState == 2) ->
495               if
496                 :: (currentEvent == energy_changed) -> Energy_SensorState
       = 0; AIEventQueue!high_energy; exGuard_Energy_Sensor_energy_changed
       = false; exGuard_Energy_Sensor_energy_changed = false;
       exGuard_Energy_Sensor_energy_changed = true;
       exGuard_Energy_Sensor_energy_changed = true;
497                 :: (currentEvent == energy_changed) -> skip;
498                 :: (currentEvent == energy_changed) -> Energy_SensorState
       = 1; AIEventQueue!low_energy; exGuard_Energy_Sensor_energy_changed
       = false; exGuard_Energy_Sensor_energy_changed = false;
       exGuard_Energy_Sensor_energy_changed = true;
       exGuard_Energy_Sensor_energy_changed = true;
499                 :: (currentEvent == energy_changed) -> skip;
500                 :: !( currentEvent == energy_changed || currentEvent ==
       energy_changed ) -> skip;
501               fi;
502           fi;
503           Energy_SensorStart = false;
504           Energy_SensorCallback = true;
505         } /* end atomic */
506     od;
507 }
508
509 byte Item_MemorizerState = 0;
510 bool Item_MemorizerStart = false;
511 bool Item_MemorizerCallback = false;
512 proctype Item_Memorizer() {
```

```
513   mtype currentEvent;
514   do
515     :: Item_MemorizerStart -> atomic {
516         AIEventQueue?<currentEvent>;
517         if
518           :: (Item_MemorizerState == 0) ->
519             if
520               :: (currentEvent == i_see_item) -> Item_MemorizerState =
      0; exGuard_Item_Memorizer_i_see_item = false;
      exGuard_Item_Memorizer_i_dont_see_item = false;
      exGuard_Item_Memorizer_i_see_item = true;
      exGuard_Item_Memorizer_i_dont_see_item = true;
521               :: (currentEvent == i_dont_see_item) ->
      Item_MemorizerState = 0; exGuard_Item_Memorizer_i_see_item = false;
      exGuard_Item_Memorizer_i_dont_see_item = false;
      exGuard_Item_Memorizer_i_see_item = true;
      exGuard_Item_Memorizer_i_dont_see_item = true;
522               :: !( currentEvent == i_see_item || currentEvent ==
      i_dont_see_item ) -> skip;
523             fi;
524         fi;
525         Item_MemorizerStart = false;
526         Item_MemorizerCallback = true;
527       } /* end atomic */
528   od;
529 }
530
531 /* Event Processor */
```

```
532 proctype processEvents() {
533   mtype newEvent;
534   mtype processedEvent;
535   do
536     :: externalEventFire;
537       /* get the external events and distribute them */
538       ExternalEventQueue?newEvent;
539       AIEventQueue!newEvent;
540       do
541         :: nempty(AIEventQueue) ->
542             /* activate statecharts in order*/
543             Move_ActuatorCallback = false; Move_ActuatorStart = true;
      Move_ActuatorCallback;
544             Eat_ActuatorCallback = false; Eat_ActuatorStart = true;
      Eat_ActuatorCallback;
545             Pickup_ActuatorCallback = false; Pickup_ActuatorStart =
      true; Pickup_ActuatorCallback;
546             Pickup_ExecutorCallback = false; Pickup_ExecutorStart =
      true; Pickup_ExecutorCallback;
547             Threat_AnalyzerCallback = false; Threat_AnalyzerStart =
      true; Threat_AnalyzerCallback;
548             Squirrel_BrainCallback = false; Squirrel_BrainStart = true;
       Squirrel_BrainCallback;
549             Wander_ExecutorCallback = false; Wander_ExecutorStart =
      true; Wander_ExecutorCallback;
550             Eat_DeciderCallback = false; Eat_DeciderStart = true;
      Eat_DeciderCallback;
```

```
551             Flee_DeciderCallback = false; Flee_DeciderStart = true;
    Flee_DeciderCallback;
552             Eat_AnalyzerCallback = false; Eat_AnalyzerStart = true;
    Eat_AnalyzerCallback;
553             Energy_SensorCallback = false; Energy_SensorStart = true;
    Energy_SensorCallback;
554             Item_MemorizerCallback = false; Item_MemorizerStart = true;
     Item_MemorizerCallback;
555
556           /* clear processed event*/
557           AIEventQueue?processedEvent;
558         :: empty(AIEventQueue) -> break;
559       od;
560       externalEventFire = false;
561       quiescent = true;
562   od;
563 }
```

References

[1] Apache Commons. Commons JEXL. http://commons.apache.org/jexl/, November 2010.

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

[3] Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, 1998.

[4] Jim Barnett, Rahul Akolkar, RJ Auburn, Michael Bodell, Daniel C. Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Mark Helbing, Rafah Hosn, T.V. Raman, Klaus Reifenrath, and No'am Rosenthal. State chart XML (SCXML): State machine notation for control abstraction. W3C working draft, W3C, May 2010.

[5] R. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14 – 23, March 1986.

[6] Joshua Brustein. Grand Theft Auto V is the most expensive game ever—and its almost obsolete. *BloombergBusinessweek*, 2013. http://www.businessweek.com/articles/2013-09-18/grand-theft-auto-v-is-the-most-expensive-game-ever-and-it-s-almost-obsolete.

[7] Alex Champanard. Understanding the second-generation of behavior trees. http://aigamedev.com/insider/tutorial/second-generation-bt/, 2012.

[8] Alex Champandard. 18 embarrassing game AI bugs caught on tape and fixed!, 2009. http://aigamedev.com/open/article/bugs-caught-on-tape/.

[9] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[10] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley, 2002.

[11] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of

fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[12] Christian J. Darken. Individualized NPC attitudes with social networks. In Steve Rabin, editor, *AI Game Programming Wisdom 4*, pages 571–578. Charles River Media, 2008.

[13] Dean Grandquist, Technical Director, Visceral Games. Keynote address at the 1st games and software engineering workshop, 2011.

[14] Kevin Dill. A pattern-based approach to modular AI for games. In Adam Lake, editor, *Game Programming Gems 8*, pages 232–243. Cengage Learning, 2010.

[15] Kevin Dill, Eugene Ray Pursel, Pat Garrity, and Gino Fragomeni. Achieving modular AI through conceptual abstractions. In *The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)*. NTSA, 2012.

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

[17] Max Dyckhoff. Decision making and knowledge representation in Halo 3. Presentation at the Game Developers Conference, 2008.

[18] Cliff Edwards. GTA5 sets opening-day record with US$ 800-million in sales. *Financial Post*, 2013. `http://business.financialpost.com/2013/09/19/grand-theft-auto-5-sets-opening-day-record-with-us800-million-in-sales/`.

[19] E. A. Emerson. *Handbook of Theoretical Computer Science*, chapter Temporal and Modal Logic. North-Holland Publishing Company, 1995.

[20] Daniel Fu and Ryan T. Houlette. Putting AI in entertainment: An AI authoring tool for simulation and games. *IEEE Intelligent Systems*, 17(4):81–84, 2002.

[21] Erann Gat et al. On three-layer architectures, 1998.

[22] Michael P. Georgeff, Barney Pell, Martha E. Pollack, Milind Tambe, and Michael Wooldridge. The belief-desire-intention model of agency. In *Proceedings of the 5th International Workshop on Intelligent Agents V, Agent*

Theories, Architectures, and Languages, ATAL '98, pages 1–10, London, UK, 1999. Springer-Verlag.

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

[24] David Harel. Statecharts: A visual formalism for complex systems. *Sci. of Comp. Programming*, 8:231–274, 1987.

[25] David Harel and Hillel Kugler. The Rhapsody semantics of Statecharts (or, on the executable core of the UML). *LNCS*, 3147:325 – 354, 2004.

[26] David Harel and Amnon Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[27] Frederick W. P. Heckel, G. Michael Youngblood, and Nikhil S. Ketkar. Representational complexity of reactive agents. In *2010 IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 257–264, 2010.

[28] Chris Hecker. My liner notes for Spore/Spore behavior tree docs. `http://chrishecker.com/My_liner_notes_for_spore/Spore_Behavior_Tree_Docs`, 2009.

[29] G Holzmann and SPIN Model Checker. *The Primer and Reference Manual*. Addison Wesley Professional, 2004.

[30] Gerard J Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[31] Damian Isla. Handling complexity in the Halo 2 AI. Presentation at the Game Developers Conference, 2005.

[32] Damian Isla. Managing complexity in the Halo 2 AI system. In *Proceedings of the Game Developers Conference*, 2005.

[33] A. Jaffe, A. Miller, E. Andersen, Y.E. Liu, A. Karlin, and Z. Popovic. Evaluating competitive game balance with restricted play. In *Proc. of AIIDE*, 2012.

[34] IO Interactive Kasper Fauerby. Scaling crowds in Hitman: Absolution. In *Vienna Game/AI Conference '12*, 2012.

[35] Jörg Kienzle, Alexandre Denault, and Hans Vangheluwe. Model-based design of computer-controlled game character behavior. In *MODELS*, volume 4735 of *LNCS*, pages 650–665. Springer, 2007.

[36] Jörg Kienzle, Clark Verbrugge, Bettina Kemme, Alexandre Denault, and Michael Hawker. Mammoth: A Massively Multiplayer Game Research Framework. In *4th International Conference on the Foundations of Digital Games (ICFDG)*, pages 308 – 315, New York, NY, USA, April 2009. ACM.

[37] John Krajewski. Creating all humans: A data-driven AI framework for open game worlds. `http://www.gamasutra.com/view/feature/1862/creating_all_humans_a_datadriven_.php`, 2 2009.

[38] Charles Krueger. Variation management for software production lines. *Software Product Lines*, pages 107–108, 2002.

[39] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[40] Paul Kruszewski. Real-time crowd simulation using AI.implant. In Steve Rabin, editor, *AI Game Programming Wisdom 3*, pages 233–248. Charles River Media, 2006.

[41] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Systematic Transformation Development. *Electronic Communications of the EASST*, 21, 2010.

[42] McGill Sable Lab. Soot, 2013. `http://www.sable.mcgill.ca/soot/`.

[43] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, 1989.

[44] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[45] Chong-U Lim, Robin Baumgarten, and Simon Colton. Evolving behaviour trees for the commercial game DEFCON. In *Applications of Evolutionary Computation*, volume 6024 of *LNCS*, pages 100–110. Springer, 2010.

[46] Donald W Loveland. *Automated theorem proving: A logical basis (Fundamental studies in computer science)*. Elsevier, 1978.

[47] M2 Research. THE BRIEF - 2009 Ups and Downs. `http://www.m2research.com/the-brief-2009-ups-and-downs.htm`, 2007.

[48] Michael Mateas and Andrew Stern. A behavior language: Joint action and behavioral idioms. In *Life-like Characters: Tools, Affective Functions and Applications*. Springer, 2004.

[49] Matthew Jack. Code coverage for QA: A practical approach to testing AI in games, 2010. `http://aigamedev.com/premium/article/code-coverage/`.

[50] Alison McMahan. Immersion, engagement and presence. *The video game theory reader*, pages 67–86, 2003.

[51] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J Holzmann. Implementing statecharts in PROMELA/SPIN. In *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on*, pages 90–101. IEEE, 1998.

[52] Ian Millington. *Artificial Intelligence for Games*. Morgan Kaufmann, 2006.

[53] 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*, pages 33 – 38. Eurosis, August 2005.

[54] J. Orkin. Three states and a plan: The AI of F.E.A.R. In *Proceedings of the Game Developer's Conference (GDC)*, 2006.

[55] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[56] D. L. Parnas. A technique for software module specification with examples. *Communications of the Association of Computing Machinery*, 15(5):330–336, May 1972.

[57] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Computer aided verification*, pages 377–390. Springer, 1994.

[58] Craig W Reynolds. Steering behaviors for autonomous characters. In *Game developers conference*, volume 1999, pages 763–782, 1999.

[59] Miro M. Samek. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Newnes, 2008.

[60] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):357–369, 2001.

[61] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39:41–47, 2006.

[62] Schwab, Brian and Mark, Dave and Dill, Kevin, and Lewis, Mike and Evans, Richard. GDC: Turing tantrums: AI developers rant. `http://www.gdcvault.com/play/1014586/Turing-Tantrums-AI-Developers-Rant`, 2011.

[63] A.M. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *TCIAIG*, 3(3):187–200, 2011.

[64] Open Source Software. JMonkeyEngine, 2013. `http://jmonkeyengine.org`.

[65] Open Source Software. OGRE, 2013. `http://www.ogre3d.org/`.

[66] C.R. Strong and M. Mateas. Talking with NPCs: Towards dynamic generation of discourse structures. In *Proc. of AIIDE*, 2008.

[67] Mankyu Sung, Michael Gleicher, and Stephen Chenney. Scalable behaviors for crowd simulation. *Computer Graphics Forum*, 23(3):519–528, 2004.

[68] I. Szita, M. Ponsen, and P. Spronck. Effective and diverse adaptive game AI. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):16–27, March 2009.

[69] Unity Technologies. Unity, 2013. `http://unity3d.com/`.

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

[71] Andrzej Wasowski. On efficient program synthesis from statecharts. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03, pages 163–170, New York, NY, USA, 2003. ACM.

[72] Qianchuan Zhao and Bruce H Krogh. Formal verification of statecharts using finite-state model checkers. *Control Systems Technology, IEEE Transactions on*, 14(5):943–950, 2006.

[73] A. Zook and M. Riedl. A temporal data-driven player model for dynamic difficulty adjustment. In *Proc. of AIIDE*, 2012.

KEY TO ABBREVIATIONS

AI: Artificial Intelligence

API: Application Programming Interface

CTL: Computational Tree Logic

FPS: First-Person Shooter

FSM: Finite State Machine

HFSM: Hierarchical Finite State Machine

IDE: Integrated Development Environment

LTL: Linear Temporal Logic

MDE: Model-Driven Engineering

MMO: Massively Multiplayer Online Game

NPC: Non-Player Character

PC: Player Character

SCXML: Statechart Extensible Markup Language

XMI: XML Metadata Interchange

XML: Extensible Markup Language