DYNAMIC BEHAVIOR-BASED CONTROL AND WORLD-EMBEDDED KNOWLEDGE FOR INTERACTIVE ARTIFICIAL INTELLIGENCE

by

Frederick William Poe Heckel

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Information Technology

Charlotte

2011

Approved by:

Dr. G. Michael Youngblood

Dr. Tiffany Barnes

Dr. Richard Souvenir

Dr. Jing Xiao

Dr. Paula Goolkasian

ABSTRACT

FREDERICK WILLIAM POE HECKEL. Dynamic behavior-based control and world-embedded knowledge for interactive artificial intelligence. (Under the direction of DR. G. MICHAEL YOUNGBLOOD)

Video game designers depend on artificial intelligence to drive player experience in modern games. Therefore it is critical that AI not only be fast and computationally inexpensive, but also easy to incorporate with the design process. We address the problem of building computationally inexpensive AI that eases the game design process and provides strategic and tactical behavior comparable with current industry-standard techniques.

Our central hypothesis is that behavior-based characters in games can exhibit effective strategy and coordinate in teams through the use of knowledge embedded in the world and a new dynamic approach to behavior-based control that enables characters to transfer behavioral knowledge. We use dynamic extensions for behavior-based subsumption and world-embedded knowledge to simplify and enhance game character intelligence. We find that the use of extended affordances to embed knowledge in the world can greatly reduce the effort required to build characters and AI engines while increasing the effectiveness of the behavior controllers. In addition, we find that the technique of multi-character affordances can provide a simple mechanism for enabling team coordination. We also show that reactive teaming, enabled by dynamic extensions to the subsumption architecture, is effective in creating large adaptable teams of characters. Finally, we show that the command policy for reactive teaming can be used to improve performance of reactive teams for tactical situations.

ACKNOWLEDGEMENTS

Thanks to all the characters for their help in running the experiments: Alice, Bob, Carl, Dee, Ephraim, Frieda, Gary, Heloise, Ian, Jillian, Karl, Lara, Manuela, Norbert, Ophelia, Pete, Quentin, as well as the many participants from the THX, SEN, SRT, LUH, PTO, TWA, NCH, and OMM families.

I also want to thank a lot of real people; my mother, Diane Heckel, my sisters, Cherye Paulson and Annie Heckel, and their husbands, Steve Paulson and Ryan Harper. Without their support and encouragement, I do not believe I would have completed this process. Evan Suma and Behrooz Mostafavi helped me relax while working on the DASSIEs project. Katie Doran, Shaun Pickford, and Samantha Finkelstein deserve special mention as well. Working in the Games + Learning Lab at UNC Charlotte has taught me a lot, and I'll miss the unique working environment we've had here. My classmates and friends from St. Louis have been part of this process: Thank you to Michael & Rachel Dixon, Nisha Sudarsanam, Richard Souvenir, Nathan Jacobs, Robert Pless and all the others for your friendship and help learning how to perform research. I am indebted to Bruce Maxwell for introducing me to research back at Swarthmore. I cannot forget Nicolas Ward, Dan Crosta, Andrew Abdalian, Branen Salmon, Nori Heikkinen and the others from the SCCS.

Finally, thank you to my advisor, Dr. G. Michael Youngblood, for helping me find my way through this process, and Hunter Hale for his patience and his navigation meshes.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
BEHAVEngine	Behavior Emulating Hierarchical Agent Vending Engine
BT	Behavior Tree
CGUL	Common Games Understanding and Learning
DASSIEs	Dynamic Adaptable Super-Scalar Intelligent Entities
FI3RST	FIrst and 3rd-person Real-time Simulation Testbed
FPS	First-Person Shooter
FSM	Finite State Machine
HFSM	Hierarchical Finite State Machine
HTN	Hierarchical Task Network
RPG	Role-Playing Game
RTS	Real-Time Strategy
SA	Subsumption Architecture
Splat	Subsumption and Python-based Lightweight AI Toolkit
SSPS	Static Spacial Perception Service
3PS	3rd Person Shooter

CHAPTER 1: INTRODUCTION

In *Reality is Broken*, Jane McGonigal provides Bernard Suits' definition of games as "the voluntary attempt to overcome unnecessary obstacles"[51, 74]. The process of overcoming these obstacles gives game players a sense of accomplishment and joy when they succeed and even when they fail. Video games, as opposed to physical games, allow game players to experience a truly novel and dynamic set of obstacles, and provide the creativity of the designer to augment their own creativity in play. Artificial intelligence techniques amplify this effect. AI characters provide companions in play when none are available as well as additional dimensions to the game rule set. For these reasons, AI is increasingly an important part of the biggest games on the market.

Academic AI largely ignores the aspects of AI systems that are most important to games. The important qualities for AI in games are that it must produce interesting—not necessarily optimal—behavior that acts in a reasonable—but not necessarily error-free—manner that does not require excessive computational resources. Game AI not only may fail, but is expected to fail. If it did not, the player would be either faced with an insurmountable challenge or no challenge at all, depending on whether the AI controlled an enemy or a friend.

Artificial intelligence has become a more important factor in video games in recent

years. Powerful graphics processing units (GPUs) have been developed for consumer use, moving graphics processing off of the main processor, which leaves more processing power available to the game logic, including the AI [56]. Recent games, such as Left 4 Dead, have brought AI into the core of the game experience and have been wildly successful [78, 3]. Left 4 Dead sold "almost 3 million copies" as of September 2009, less than a year after its release and has demonstrated just how important AI can be in games for entertainment when used effectively [79]. Games are not just used for entertainment though, as they are frequently used to create training simulations, such as DARWARS Ambush! or Virtual Battle Space 2 [18, 39]. For many training games AI is necessary to produce realistic environments.

Good game AI is difficult to produce. Even with the additional processor time freed by the graphics pipeline, the resources available to AI are quite limited. While graphics no longer dominate the CPU, game logic and animation are still very demanding. Furthermore, games are expected to run at very fast frame rates; Valve's popular Source engine uses a default tick rate (the rate at which the server updates) of 66hz for its the multiplayer server [77]. Running one or two AI characters along with game logic in this time frame is not difficult, but scaling may become a problem for large numbers of characters. In addition, game environments tend to be fast paced and highly dynamic. Characters may be frequently injured or destroyed, and events occur in real time. Achieving team coordination produces additional problems, because team plans may be frequently invalidated as key characters are disabled or situations require changes in roles.

Simulating intelligent adversaries and allies in games can be a very difficult task.

Many different techniques for controlling game characters have been developed, including scripting, planning, cognitive modeling, and reactive techniques. A class of methods called *reactive techniques* in particular provide very fast control, allowing many characters to be simulated at a time. Unfortunately, it can be difficult to build teams of reactive characters that work together to provide interesting and fun gameplay. This research aims to address the problem of building sophisticated game characters using inexpensive reactive techniques and by enhancing the richness of the game environment in which the characters live. We start from reactive control methods because they have the important quality of being computationally very inexpensive.

Reactive control was recognized as a valuable paradigm for controlling robots by Rodney Brooks when he developed the subsumption architecture in 1986 [5]. Like video game characters, robots must respond in real time in a highly dynamic environment—the physical world. In addition, the CPU is frequently dominated by other processes, though for robots these processes tend to be concerned with perceiving the world. Due to battery constraints, robots may have very low-powered processors, further limiting resources. The key observation underlying reactive control is that "the world is its own best model," [6] so decisions should be based primarily on the observed state of the environment. Reactive control techniques therefore make decisions based on current perceptions of the environment and limit decision-making to the immediate next action that the system should take.

In contrast, methods that reason primarily on a model of the environment, such as planning methods, are known as deliberative control. Deliberative methods make

decisions based on a model of the character's environment, rather than immediate perceptions of that environment. This allows deliberative models to make decisions for not just immediate next actions, but a series of actions to take in the future. They also frequently make more use of symbols, which are abstractions that provide a representation of some aspect of the environment (*"chair"*, for example, is an abstract symbol describing the physical object that you are currently sitting on). While reactive systems provide fast and computationally inexpensive response times to environmental changes, their decisions may be less optimal than those made by a comparable deliberative system. Deliberative control methods are typically better at generating sequences of actions to overcome local minima problems, such as the classic U-shaped obstacle in robotics (seen in Figure 1.1).

The local minima problem occurs because the reactive system is unable to look into the future; when attempting to achieve a goal, it follows what appears to be the shortest path, but does not anticipate that the obstacle will block its path. As it attempts to move around the obstacle, it fails, because escaping the obstacle requires moving away from the goal. The local minima problem is easily visualized with an obstacle, but it also applies in other circumstances. Even if some path planning is allowed, without considering any symbolic knowledge, a character will fall into a local minima of taking the shortest path through a dangerous region of the map. A group of characters acting together may all attack from the same direction, rather than reasoning about flanking tactics that will produce a better, though less obvious, result.

The local focus of reactive control decisions can clearly cause some difficulties for

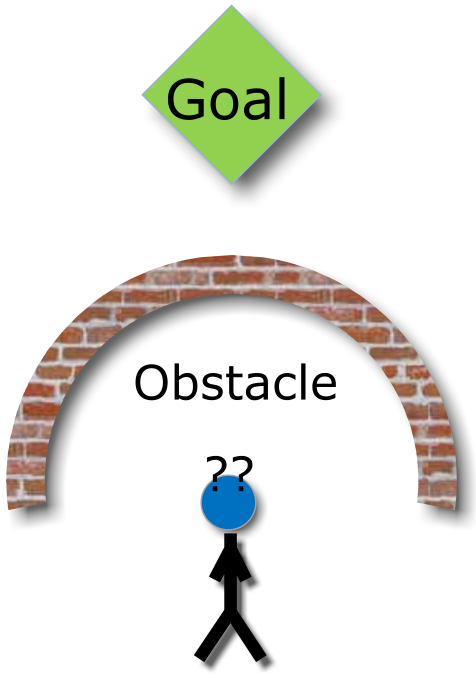


Figure 1.1: The local minima problem. One illustration of this is a reactive character that attempts to achieve the goal by moving so it is always closer to the goal. The system does not know how to handle the U shaped obstacle because going around the obstacle would require moving away from the goal. Deliberative systems are able to plan around the U-shaped obstacle.

game characters. It is recognized in the game industry that *perfect* tactics and strategy is not desirable for game AI [47]. Even so, characters should exhibit some level of tactical/strategic thinking. Without the ability to strategize, characters may fail to recognize that a chosen path through the world is dangerous or a particular area is important to guard. The sophistication of interaction with users may be diminished as well, with AI teammates unable to execute an attack plan with the player. Frequently, without the ability to strategize effectively, AI characters will cheat by making use of unlimited resources or perfect knowledge of players, which can cause frustration if the player notices the cheating [14]. Another issue that can rise for reactive control methods is difficulty in applying multi-agent techniques, since common

reactive methods such as finite state machines, hierarchical finite state machines, and subsumption architecture typically have static structures which are not modified at run-time. When designing characters, representations for building these characters can be difficult to manage as the structures can quickly grow very large, and individual components of the controller may interact in ways that reduce modularity. The difficulty can be compounded when AI designers are not programmers and are less experienced in managing this type of complexity. For training simulations, developing realistic characters may require a domain expert who understands the cultural norms of another society but does not have significant AI background. These shortcomings reduce the sophistication of reactive characters, and can make reactive control less appealing to designers.

An alternative method of control known as *behavior-based control* aims to solve some of the problems of reactive control. The goal of behavior-based control is to create systems with the speed and response times of reactive control but with more flexibility. To achieve this, restrictions on the use of symbolic information and extra internal state are relaxed, allowing better generalization of individual behavior components. Better generalization leads to better modularity of the architecture, improving the reusability of characters and reducing the complexity of their controllers [48]. The components of a behavior-based controller are modular behaviors with a common interface, designed so that direct behavior interaction is limited. In the game industry, the same difficulties with reactive methods (especially finite state machines) inspired methods such as behavior trees [35]. Behavior-based control makes characters easier to manage and enables better reuse, but does not provide a complete solution to the

problems of creating strategic characters or coordinating characters in teams.

This presents a serious question for reactive and behavior-based control: can the strategic and multi-agent shortcomings be overcome while still providing a simple representation that is accessible to designers? *Our central hypothesis is that behavior-based characters in games can exhibit effective strategy and coordinate in teams through the use of knowledge embedded in the world and a new dynamic approach to behavior-based control that enables characters to transfer behavioral knowledge.* The primary components of this research are:

1. Applying information compartmentalization to embedded world knowledge to enhance the capabilities of intelligent characters and represent uncertainty indirectly.

2. Applying the reactive paradigm to multi-agent coordination to achieve computationally inexpensive teams of intelligent characters that can produce tactical and strategic behavior.

3. Comparing the use of world embedded knowledge together with reactive teaming and behavior-based control to existing techniques from the game industry and research community to evaluate the applicability in games.

These new methods are developed for and evaluated in real-time games. In particular, we use a combination of tactical first person shooter/3rd person shooter (FPS/3PS) and role-playing game (RPG) environments. FPS and 3PS games tend to be highly action focused, requiring the player to engage in fast-paced tactical en-

counters. RPG games, by contrast, frequently have more exploratory and puzzle components, and encounters with enemies are not as fast-paced.

These two general categories of games will test the ability of our work to use both *tactics* and *strategy*. Tactics is, according to the Oxford English Dictionary, "the art or science of deploying military or naval forces in order of battle, and of performing warlike evolutions and manoeuvres [58]." Tactics focus on particular defined maneuvers that are used in a combat environment, usually to achieve a particular objective. Strategy, on the other hand, is "the art of a commander-in-chief; the art of projecting and directing the larger military movements and operations of a campaign [58]." The key difference is that strategy has a more global focus; a strategy will direct groups of characters, which themselves may employ tactics. In other words, tactics focuses on the short term, fine-grained scale of character control and coordination, where strategy focuses on the long term, broad scale of control. FPS games require careful use of tactics, with less emphasis on strategy, while the puzzles of RPG games require longer-term consideration of actions to overcome local minima.

The underlying control method used throughout this work is a behavior-based subsumption architecture. Subsumption uses a prioritized list of behavior layers, as seen in Figure 1.2. Layers are a 3-tuple, composed of a trigger, a behavior, and a *subsumption policy*. Triggers read filtered information from the environment, referred to as *percepts*, and return true when one or more conditions are met. Behaviors may be a single primitive behavior, or composed of multiple behaviors structured hierarchically. Behaviors make decisions based on the current state of the world and output *action requests*. Because layers may be run in parallel (one of the primary

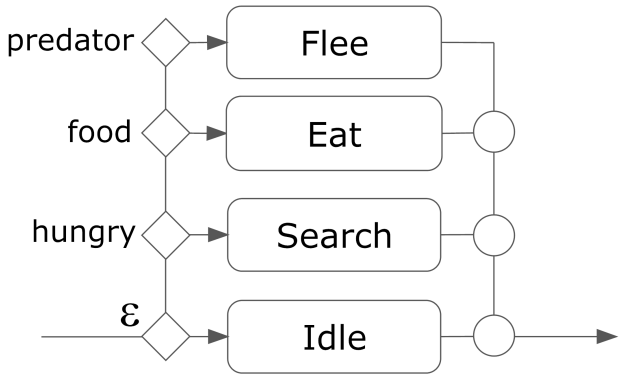


Figure 1.2: Subsumption architecture representation. Layers represent behaviors, which may be simple or composed of multiple behaviors. Higher layers have higher priorities. Circles represent subsumption policies, which dictate whether higher layers override lower priority layers or allow them to run.

benefits of subsumption), each layer is also required to have a subsumption policy that dictates how its action requests are combined with the requests made by lower-priority layers.

1.1 Environmental Intelligence

Embedding intelligence in the world can be done in a number of different ways. A common approach that is used in the game industry is called *influence mapping* [75]. Influence mapping is used to annotate areas of the world where different players are powerful, where battles have taken place, or that may be important to protect. One problem with this approach is that it is based on an underlying grid-based map; updating the influences can be very expensive if a fine-grained grid is used (in a 10 meter by 10 meter map, with 0.5 meter resolution, 400 values would be needed). Multiple maps may be required to track different types of influences, and influences may decay over time. These factors can make influence maps expensive to store and maintain.

It is important to note that perceptions of the game world are generated from the underlying world representation. The choice of representation has consequences not just for AI, but also general game logic as it is used for collision detection, graphical culling, and other important tasks. Influence maps are used when the underlying representation is a grid. Another representation that is frequently used in modern games is the navigation mesh. Navigation meshes decompose navigable space in a virtual environment to create a number of convex regions, as seen in Figure 1.3. These regions, and the portals between them, can be transformed into a graph, which allows navigation to be split cleanly into a global path planning problem (finding a path from one convex region to another) and a local navigation problem (moving within a region). An important quality of navigation meshes is they allow information to be compartmentalized; calculations for visibility, collision, and other factors can be accelerated by only considering objects and characters within the current region. For AI, the navigation mesh can be used to provide additional information, such as whether a region is inside or outside a building, and how the character can move from one region to the next (it may need to climb a wall, or crawl on hands and knees, for example). We have adapted the approach of influence mapping to navigation meshes as *influence points*. Influence points can be far cheaper in terms of both computational and space complexity due to the reduction in size provided by the navigation mesh representation.

In addition to world representations, the action and perception space of characters must be defined. Modeling actions and perceptions can be non-trivial when a large variety of objects are present in the environment. Without additional information,

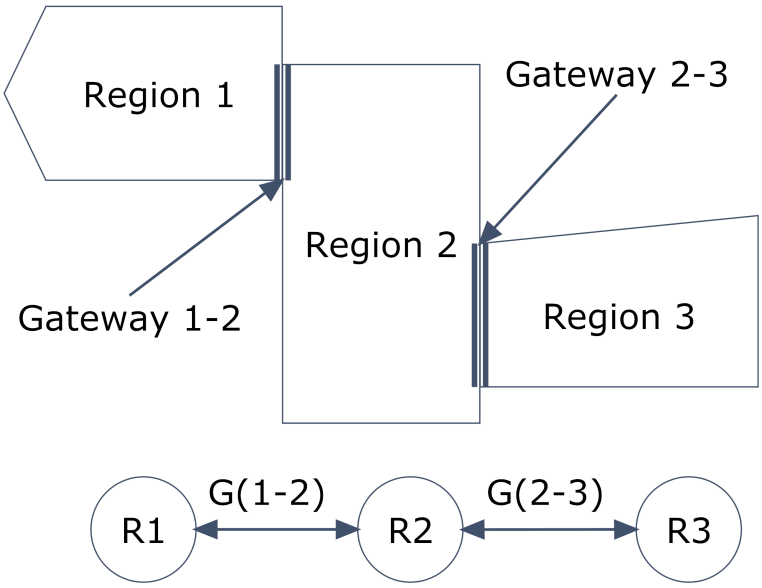


Figure 1.3: Example navigation mesh. Geometric and graph representations are shown. Regions must be convex. Gateways may be directed or undirected; in this case, the gateways are undirected, allowing passage each way, as shown in the graph.

deciding whether an object can be picked up, for example, may require a great deal of geometric computation. Instead, characters can carry the innate knowledge of what objects are possible to pick up. This presents a problem when new objects are made available, as the characters must be changed to be aware of them. One solution is to attach *affordances*, or information about possible actions, to the objects themselves. When a character is created, it may not possess any knowledge of what actions can be applied with what objects, and receives this information from the objects themselves.

We extend this concept of simple affordances by adding context, enabling multiple characters to coordinate through affordances, and representing uncertainty through these smart objects. These techniques increase the power of even simple random agents, and decrease the complexity of individual behaviors for crafted agents. We

evaluate our affordance extensions by comparing the performance of agent controllers that are aware of contextual affordances with controllers that are not.

1.2 Reactive Teaming

A key aspect of player immersion is team coordination. Achieving team coordination can be a real challenge, and may require the creation of specific team-based behaviors. There are two major flaws with this approach. First, behaviors that are team-based may be very different from solo behaviors. This means that redundant behaviors must be created, first for solo characters, and then the same task for team-based characters. Second, these team behaviors may not adapt well to varying number of teammates. Ideally, behaviors should take advantage of as many teammates as are available for the task, and adapt as the team size changes.

If creating team-specific behaviors is not the solution, then the architecture itself needs to be modified to allow more dynamic behavior. Reactive architectures generally provide no clear mechanism for coordinating multiple solo characters to perform a team-based task. Behaviors are predefined for each character, and usually static. Subsumption architecture in particular was designed to mimic the operation of electronic circuits, and does not provide for dynamic changes to the character controller. Working with modular behaviors in software provides more flexibility than subsumption as originally defined, and dynamically changing the behaviors in a behavior-based system is not as daunting a task. One of the major problems in multi-agent systems is deciding how to assign tasks or behaviors between multiple agents on a single team. This task is the *commitment* problem of Jennings, and the problem of selection of agent activities as described by Lesser [36, 43]. The assignment problem

is the focus of auctions, planning, free market and other approaches to multi-agent coordination[8, 25, 37]. A secondary, related problem in games is how to create tasks that generate more direct character interactions—not only must we assign character behaviors, but we must make sure that the behaviors take advantage of the team. Reactive teaming decides how to assign behaviors through interactions between pairs of characters. By making this decision pairwise, rather than requiring information from the entire team, individual behaviors can be assigned very inexpensively. In addition, the choices of when to request a new assignment, and what assignment to transfer, are local reactive decisions based on the current state of the agents.

We approach the problem of creating behaviors that coordinate with other members of the team at both the individual behavior level and with world-embedded knowledge. Individual behaviors can be divided so that both the originating character and the receiving character each possess non-overlapping parts of the task. World-embedded knowledge can be used by allowing characters to annotate the environment with influence points, and can also provide multi-agent probabilistic affordances to enable close character coordination. While reactive teaming does not provide guarantees about the quality of the team assignments, it can create team coordination at a much lower computational cost than other techniques.

Reactive teaming is evaluated by comparison with auction-based methods applied to both behavior-based control and the industry technique of behavior trees, as well as crafted teams that statically define the role of each character on the team. Because it is extremely difficult to find objective measures for AI performance, we compare the performance of our techniques with accepted game industry standards. This type

of comparison only shows whether the techniques have a similar range of behavior generation, so we also compare the complexity of characters in both design and run-time computation. We chose this approach because it can capture improvements in resource consumption and verify that our extensions do not compromise baseline performance.

1.3 Overview

We begin this work with an overview of existing methods for embedding knowledge into the world and accepted practices for building AI controllers. In Chapter 3, we describe each technique in detail, including expected run-time and design complexity, starting with intelligence embedded in the environment and continuing with dynamic extensions to behavior-based subsumption. Chapter 4 provides details of our implementation in two testing environments, DASSIEs and SPLAT. Evaluation of all techniques is covered in Chapter 5, and we wrap up with a review of key findings as applied to the thesis hypothesis in Chapter 6.

Throughout the proposal, we will generally use the terms *character* and *agent* to mean the same thing. Character is preferred except in certain contexts, such as the term *multi-agent*, when agent is a well-established term. We will refer to single updates during a game loop as *ticks*.

CHAPTER 2: BACKGROUND

2.1 World-Embedded Knowledge

Embedding knowledge in the world is a powerful technique for making behavioral information available to agents without individual programming. Embedding knowledge requires an underlying world representation that is appropriate for these techniques, as simple grid-base representations can be extremely expensive in terms of computational resources.

Our methods use navigation meshes provide a topographical representation of virtual worlds. As game worlds grow in size and complexity, it becomes important to move away from standard occupancy-grid style maps, which provide fine resolution but use large amounts of memory and can be expensive to update with dynamic information. Navigation meshes provide decompositions of the world into negative space (traversable) and positive space (obstacle) regions. These regions are usually much larger than a single grid cell and different regions may vary in size and shape. Because they can be represented as graphs instead of a large map grid, navigation meshes use far less memory, and information can be stored with far less expense than in grid maps. Each of our embedded knowledge techniques uses navigation meshes as the underlying organizational representation for storing information (though affordance-based techniques are possible to use without navigation meshes).

There are a few methods of generating navigation meshes. The Hertel-Mehlhorn algorithm decomposes the world into triangular regions, but can result in large numbers of very thin triangles [32]. Space-filling volumes places square seeds throughout the world, and grows them until they reach an obstacle, at which point growth stops in that direction [76]. This method works well for worlds where obstacles are axis-aligned, but fails with more complex geometry. Other methods that can be used include Voronoi diagrams and probabilistic road maps, but these are useful primarily for agent navigation, and are not as effective for other purposes [66]. We use Adaptive Space Filling Volumes (ASFV) to to generate navigation meshes [27]. This method is based on Space Filling Volumes, but can handle complex world geometry that is not axis-aligned. This method produces meshes which are far less complex than triangle-based approaches. While we use ASFV-based navigation meshes, it should be noted that our techniques are independent of the navigation mesh generation method.

2.1.1 Influence Mapping

Influence mapping is a technique to provide tactical information to agents in games [75]. An influence map is a grid representation of the game world, with values representing the local power, or *influence* of the different forces in the world. Influence values may be based on enemy unit locations, bases, resource banks, lines of fire, or many other sources of information. They have been used in games such as *Age of Empires* [63]. Often multiple influence maps are used to track different types of tactical information, and the different types must be combined for decision making. Influence map trees store influence maps as leaves in trees, which can be manipulated at each level[52].

While influence maps are extremely useful, the use of map grid representations makes them very expensive. The full maps must be regenerated as conditions change; if maps are implemented using a full size grid of the world, each insertion of an influence and update of the map is an $O(n)$ operation, where n is the number of cells in the map grid. Furthermore, the memory cost is also $O(n)$ if a single map is used, or $O(n*m)$, where m is the number of influence types, if multiple maps are used. Given the large size of virtual worlds, this cost can quickly become non-trivial. Loading only part of a world at a time can improve this, but at the cost of additional complexity. Our approach to annotating the world with influences shares many common features with influence maps, but uses navigation meshes to greatly reduce the computational and memory costs.

2.1.2 Affordances

The concept of affordances, as introduced by Gibson, describes all "action possibilities" available in the environment [26]. For example, chairs have affordances for sitting; a lightweight chair may also have an affordance for lifting, and a folding chair has an affordance for folding. In design, Don Norman used the concept to describe the importance of making the use of objects clear to consumers [57].

Affordances are a useful tool in virtual environments, as they can be used to reduce the complexity of character intelligence by moving behavioral information out of the character controllers. One of the most important examples of affordances in games is from *The Sims* [71]. Apart from simplifying agents, *The Sims* also used affordances to allow major content updates without having to make changes to the carefully tuned AI controllers. Other games have used affordances to create smart objects. The game

F.E.A.R used affordances in the form of smart objects [61]. The smart objects in *F.E.A.R* provided all information about animations (and therefore actions), allowing many different behaviors to be specified in a single state in conjunction with a planning system. This effectively decouples objects from the animation system, but does not significantly reduce the decision space of characters. Cerpa described an architecture that used smart objects in conjunction with behavior trees [11]. Cerpa's smart objects would dynamically add behaviors to the AI controller to perform different actions. This is an effective way to reduce the decision space for agents using smart objects, but requires adding full behaviors to objects, which may require an AI programmer. Our approach for contextual affordances strikes a compromise by filtering the available actions reported by objects. This still provides animation information to characters, but reduces the burden on the level designer for building full behaviors to be attached to objects. In addition, we provide techniques for adding probabilistic and multi-agent coordination information to objects, reducing the complexity of the AI controller.

2.2 Agent Control Techniques

Agent control techniques broadly fall on a spectrum ranging from purely reactive to purely deliberative. One particular type of technique, scripting, can be said to fall on both sides. Depending on the script, it may be purely reactive (only having reflexive responses to environmental stimuli) or purely deliberative (once started, it will ignore environmental stimuli and execute its programming blindly). Scripting can be used to implement other techniques as well, as some scripting languages are general programming languages.

From most deliberative to most reactive, excluding scripting, there are many types of techniques: planning, cognitive models, behavior-based, reactive, and reflexive. Examples of each of these techniques can be found throughout game AI.

2.2.1 Planning

Planning methods such as STRIPS allow the input of set of possible tasks with preconditions and expected postconditions, along with the starting world state and a goal world state (including agent state) [22]. From this input, a complete plan from start to finish can be computed if it is possible. HTN (hierarchical task network) planners search for plans using a hierarchical decomposition [67]. Initial plans are decomposed into subplans, and subplans are further processed until an ordered set of actions are discovered. HTN planning can use primitive tasks such as those required by STRIPS planners, but is best used with a plan library of known task decompositions. Relationships between subplans are are expressed as a network rather than as individual tasks with preconditions and postconditions.

Planning methods can be very expensive; in fact, without major restrictions, deciding whether a plan exists for a given HTN planner is a PSPACE-hard problem (though restrictions can be made to reduce it to a NP-complete problem) [20]. Despite this, in practice planning algorithms can be used quite effectively. The game *F.E.A.R.* used a STRIPS-style planning system called GOAP (Goal-Oriented Action Planning) [59, 61]. GOAP uses a number of modifications to allow the planning system to run with sufficient speed. First, costs are applied to actions to represent that some actions are preferable to others. In addition, instead of using add and delete lists, GOAP uses a fixed-sized array to represent world state. Procedural precon-

ditions are used when certain preconditions are expensive to calculate and are not always needed in the world state array. Finally, because actions do not always have an immediate effect, the actions required by a plan are executed by a finite state machine to allow actions to take place in the correct order.

Killzone 2 used HTN planning for its bot AI [73]. The authors report that with 14 bots in a multiplayer game, they were able to generate about 500 plans per second, with each plan requiring about 15 decompositions. These numbers indicate that at this scale, planning can be achieved, but the authors do not provide how much further the system is able to scale before it becomes infeasible.

While computational expense is a major concern for planning-based techniques, the knowledge-engineering problem may be an even larger barrier to building planning systems. Preconditions, expected postconditions, and models for handling uncertain world state are frequently non-trivial to develop. Even for simple domains, the number of operators required for planning can be extremely large. Planning is frequently used in games in ways that closely resemble reactive control methods. We do use some minimal planning methods in our research, but this is limited to the use of planning paths through the world. When used in combination with navigation meshes, path planning is a very effective way to enable global navigation for intelligent characters.

2.2.2 Cognitive Architectures

The goal of cognitive architectures is to create cognitively plausible models of human cognition. They are typically rule-based systems. Some cognitive architectures approach cognitive plausibility very carefully, including issues of timing, while others primarily focus on processes while running at more expedient speeds. The ACT-R

architecture, created by John Anderson, is a rule-based system that includes cognitively plausible timing information [1]. Because of the attention to timing, agents created in ACT-R sacrifice speed for plausibility.

Soar is another major rule-based architecture [41]. Unlike ACT-R, Soar allows much faster execution of agent rules. Intelligent game characters have been written in Soar, unlike ACT-R.

Apart from the computational costs and timing issues associated with cognitive architectures, building models in cognitive architectures is a skill that requires a large amount of experience to develop. Cognitive modeling is very different from usual methods of programming, frequently resembling the process of knowledge engineering for planning. Unlike planning, though, these models are still building procedures rather than models for discovering action sequences to achieve goals. Projects such as Herbal attempt to make this process more accessible, but more work is necessary before cognitive modeling can be considered a generally accessible technique for building intelligence [13].

Cognitive models have the potential to enable realistic human-like behavior in games. This type of behavior is sometimes desirable, notably in training simulations. Outside of high fidelity training simulations, though, there is little focus on generating realistic behavior. Given the difficulty of building cognitive models, this greatly reduces the value of cognitive modeling for the creation of intelligent characters. The structures of cognitive architectures can still provide some useful insight and mechanisms for other AI architectures. The overall structure of the BEHAVEngine architecture was inspired by cognitive models, and we used concepts in the memory

model that are derived from ACT-R and Soar.

2.2.3 Reactive and Behavior-based Control

Given the difficulty in building cognitive models, and the effort required to generate world models to be used with planning systems, we chose techniques that fall into the categories of reactive and behavior-based control for our architecture. Many different reactive architectures for agent control exist, though only a few are commonly used for games. In general, reactive agents can be said to map from an input of *perceptions* to a set of output *actions*. Rather than relying on an internal model of the environment and producing plans, reactive agents make decisions based on the world state directly. An important distinction between planners and reactive controllers is that planners build a sequence of actions to provide behavioral guidance through a point in the future called the *planning horizon*, but reactive agents only make decisions for the current time step.

Finite state machines (FSMs) and hierarchical finite state machines (HFSMs) are a popular and simple to use method for building reactive agents. They are inherently modular, allowing a library of primitive behaviors to be developed along with transitions based on combinations of simple conditionals. The behaviors can then be connected by these transitions using a graphical representation. While FSM-based methods are simple as a concept, they suffer from the problem of becoming unmanageable past a certain size, since maintaining the $O(n^2)$ transitions between states becomes an increasingly difficult problem. HFSMs reduce this issue by providing super-states that encapsulate portions of the machine. Concurrency issues can be addressed with non-deterministic finite state machines by treating ϵ transitions as

indications of states that run in parallel. Without this convention, concurrency can result in an exponential increase in the number of states. Note that nondeterministic finite state machines can be converted to deterministic finite state machines, though the number of states are much larger for the deterministic equivalent. Hierarchical finite state machines have proven useful in games, and provide the basis for some types of behavior tree [24]. The lack of built-in parallelism in both architectures is a major limiting factor, and can be solved with other architectures.

Another reactive control method that may be used is subsumption architecture. Subsumption was proposed by Rodney Brooks as a robot control architecture [5]. Agents are defined as a series of prioritized layers. Each layer is composed of a triggering condition, a primitive behavior to run, and a policy for interacting with other layers. Higher layers may adjust the control of lower layers, but lower layers may not interact with higher layers. Higher priority layers may completely override or *subsume* lower layers, or adjust their output in some way. Transitions need not be explicitly specified, though the closest analog in subsumption is the subsumption policy. Subsumption is inherently parallel, which eliminates the need for constructs such as ϵ transitions or schedulers. The prioritized structure of the layers reduces the number of subsumption policy decisions to be much smaller than the number of transitions in an FSM. Behaviors can be executed in sequence by composing a layer of multiple behaviors, but executing layers in a sequence is more complex than in a standard FSM unless each layer has a perceivable effect on world state.

Nakashima and Noda introduced a version of a dynamic subsumption architecture called Gaea [55]. Gaea uses *organic programming*, using *cells* instead of the usual

layers. This organizes the subsumption into behavioral layers and functional layers. Reconfiguration of the architecture is allowed by swapping cells. This is similar to the dynamic extensions presented in this work, though we use failure monitoring to inform the changes to the architecture and allow adding completely new behaviors to the architecture. Also, our work differs in that low level behavior (performing primitive actions) is completely separate from the subsumption controller.

Subsumption has been used in commercial game titles. Eric Yiskis describes a subsumption architecture for games with layers divided into immediacy of goals [87]. The layer division used is Tactics, Behavior, Action, and Movement, with each layer composed of a finite state machine. Each layer has a well-defined responsibility, and no layers may overlap. Higher level behaviors send requests to the lower level behaviors, and can only communicate with the layer immediately below. This can greatly limit the flexibility of subsumption architecture, as controllers can generate more complex behavior when layers are allowed to overlap. Loew and Hinkle have also described a simple subsumption architecture to allow agents to execute multiple actions in parallel, allowing opportunistic actions to be taken while other behaviors are active [44]. Our approach is more similar to the work of Loew and Hinkle.

Many criticisms of reactive architectures assume the lowest possible level of abstraction, where the controller is responsible for *all* decision making, including low-level animation execution and perception filtering. This is not a fault of reactive control techniques, but rather of engineering. Highly integrated controllers can be easily simplified by separating decision logic from control logic. When this distinction is made (as it is in our work), working with reactive controllers is greatly simplified. Further

improvements can be achieved by using behavior-based versions of common reactive architectures.

Given the nature of game environments, reactive control places some unnecessary limitations on character controllers. Notably, in games the environment is entirely symbolic, which eliminates most of the problems associated with symbol grounding. Since world state contains symbolic information, rather than the low-level sensor readings that robots depend on, reactive systems in games typically fall into the category of more state-rich behavior-based systems.

Even outside of games, reactive techniques are frequently implemented as behavior-based control methods. Behavior-based control was initially proposed by Maja Mataric as an alternative to pure subsumption for robot control [48]. Behavior-based techniques differ in that the individual modules that make up a controller may not be simple states, but could be full finite-state machines or other state-based controllers themselves. In other words, behavioral components may contain state, including simple world representations. Aaron Khoo described the use of behavior-based subsumption as a game AI method with abstracted trigger/action modules [38]. While subsumption in its basic form is perhaps the simplest type of behavior-based control, as it provides a clear conflict resolution system in the use of prioritized layers, Khoo describes other methods to resolve action conflicts without the use of static priorities. Behavior-based hierarchical subsumption techniques are also in use [42]. Our work uses hierarchical subsumption techniques, as the additional abstractions enabled by hierarchical methods are extremely important for simplifying the character design process.

Extensions to behavior-based HFSMs are provided in the behavior tree (BT) formulation of Damian Isla [35]. In behavior trees, the HFSM is represented using a tree structure. Internal nodes of the tree act as transitions between leaf nodes, which execute behaviors. These internal nodes provide conditional tests and implicitly define transitions between groups of nodes; for instance, a *sequence* decision node results in child nodes being visited in order without explicitly creating transitions between the children. The result is that transitions are only explicitly defined for parent-child relationships and not for sibling relationships. Behavior trees also allow *impulses*, which effectively reduce the number of common subgraphs. An example of an impulse would be to give a subgraph multiple priorities in a *prioritized-list* decision node; under most circumstances, subgraph A should be lower priority than subgraph B, but given a particular set of percepts P, A should be considered before B. While the resulting behavior tree controller is equivalent to a similar HFSM, these abstractions can potentially reduce authorial burden, but the resulting transition functions have a higher complexity. Behavior trees also have a larger set of concepts that must be understood by the AI designer prior to building agents. The behavior tree methods used heavily in industry frequently fall in the category of behavior-based control. Two major aspects of behavior trees is that they are highly modular and the control structures include additional state– more than is commonly used in state machines.

The popularity of behavior trees in industry make them a natural choice for comparison with our subsumption-based methods. In addition, several of the techniques we describe in this work can be directly applied to behavior trees (or even FSM-based controllers) with no modifications. The complexity of the representations created for

behavior tree-based approaches is very similar to that of the complexity of subsumption architecture.

2.2.4 Behavior Languages

Several approaches to agent control exist which do not fit cleanly into the other categories. Notably, the family of approaches based on the Reactive Action Packages (RAPs) of Firby lie somewhere in between reactive, planning, and cognitive architectures [23]. RAPs are similar to behaviors, and define a task net, which is a partially ordered sequence of actions. RAPs may have preconditions which must be true to run, constraints which must remain true for the behavior to continue running, and a success clause that can detect when the goal the RAP is meant to satisfy spontaneously completes before the task net has finished running. These RAPs are dynamically added to the behavior controller, and the executive system of the agent decides which to run at any given time. The Hap architecture of Bryan Loyall is based on RAPs, but simplifies the RAP language in some ways and adds support for a number of additional features [45]. A further refinement of Hap is A Behavior Language (ABL), which changes Hap to have a Java-based syntax and further extends the language to provide multi-agent coordination, specifically to support story development in games [50].

While these techniques have their own approaches to controlling behavior execution, their languages can be used effectively to generate behaviors for many other architectures. For example, an ABL interpreter embedded in a subsumption behavior would enable the use of the ABL language structures for creating the building blocks of a subsumption-based agent.

2.2.5 Multi-Agent Control

Much of the research in multi-agent planning focuses on providing near-optimal task assignments for teams. Multi-robot task assignment, a close parallel to game team coordination, can be defined as a scheduling problem [17]. Scheduling defines a number of jobs which must be executed by a number of machines, frequently by a certain deadline. Many different approaches have been taking to finding near-optimal solutions for team coordination, including planning, auctions, and free-market methods [8, 25, 37]. These methods can be expensive to use, and provide optimality guarantees that are frequently unnecessary for game characters. We seek to develop approaches that can be used for coordinating teams of characters when optimality guarantees are unnecessary or the team size is very large. In cases where traditional coordination methods are needed, the extensions we have developed for subsumption architecture allow the use of these methods.

Approaches to team coordination of behavior-based agents often avoid global methods, instead focusing on local approaches. Werger observed that some emergent team behaviors developed from agents developed with no awareness of the team [85]. While this is possible, it becomes difficult to design teams which coordinate through emergent behavior. Learning and swarming approaches without explicit communication have been successful [70]. Adding explicit communication which conveys a teammate's state provided success with a robotic box-pushing task [49]. Another approach divided the working area of the robots in a team into territories, so that each robot was responsible for one area; this approach also dealt with the problem of broken

robots [68]. Our approach of reactive teaming integrates some of these ideas. Reactive teaming is an emergent approach, but it uses explicit communication. In addition, reactive teaming allows the designer to specify restrictions that limit the emergent qualities of the team behavior.

In games, teams of agents are commonly coordinated through a combination of decentralized and centralized AI. Decentralized team AI extends individual agent controllers to take its teammates' current observations and intentions into account when make its own decisions [80]. Decentralized methods are considered to be too weak to perform more sophisticated coordinated actions, so centralized methods typically provide an additional level of AI that issues commands to individual agents. When centralized methods are used, the team coordinator can use either an authoritarian or coaching style, depending on the responsiveness required [81]. Our main focus is on providing decentralized methods, though our extensions do allow the use of centralized techniques for controlling team behavior.

Decentralized teams may use blackboards to communicate information between agents, which provide a central shared memory area that agents can read and update with their current state [60]. The type of information shared by AI agents in the game *No One Lives Forever 2* includes requests for agents to move when they are blocking another agent, messages to change pathfinding behavior, notifications of events, and requests for particular behaviors to be executed. The blackboard provides an advantage in that agents only need to query the blackboard rather than sending individual messages. Instead of using a blackboard, our agents communicate either through knowledge embedded in the virtual environment or through behavior (such

as speaking to one another).

The GOAP architecture implements team coordination by providing different levels of goal planning [62]. When an action is chosen by a high level unit, the action includes suggestions of goals for the subordinate units under its command. Each level of the command hierarchy filters goal suggestions down in this manner to the last units in the chain of command. Note that in this implementation, goals cannot be *added* to the controllers of lower level units, but instead importance modifiers are applied to the goals which are already available to the units. Positive modifiers make a goal more likely to be executed, while negative modifiers suppress goals. This approach is more restrictive than reactive teaming with subsumption. Characters using our reactive teaming approach can receive entire behaviors from other characters, allowing them to execute behaviors that are not initially provided in their controller. In addition, because reactive teaming characters instantiate their hierarchy behaviorally, command structures are more flexible.

Orwellian State Machines provide an approach to using subsumption to coordinate teams of agents [4]. In the OSM framework, higher level agents in the command hierarchy subsume the lower level agents by sending orders to perform tasks or modifying the actions taken by lower level agents. The framework also proposes a mechanism to treat individual characters as a multi-agent OSM. The OSM is organized as a directed acyclic graph (DAG), so multiple higher level agents may send commands to a single lower agent. Our approach also allows multiple agents to command a single agent, but the level of control is not as strict as OSM-based teams.

Other approaches to teaming use a combination of multiple techniques. One of

the major domains for multi-agent coordination is that of robot soccer. Robot soccer teams are of fixed size, and must deal with a large number of challenges in sensing and control. Many of the techniques in robot soccer are directly applicable to sports video games. Both domains require tightly coordinated teams, and characters are more permanent than in other video games. This means that much of the work is focused on developing skills such as passing the ball, shooting goals, and developing play patterns. The CMU CMDragons team, which won the small-size competition in 2006 and 2007, had highly specialized roles in the team for offense and support [7]. One method the CMU team has used for coordinating the team and choosing the appropriate *play* for the robots to execute applied case-based reasoning approaches, which have also been applied to the simulation league [65, 84].

Another interesting approach to multi-agent coordination is an outgrowth of auction-based methods, known as market-based coordination. This technique has agents bid on tasks, and once tasks are assigned, the agents themselves can run additional auctions to further split the task or exchange a task for another [19]. The market-based approach combines centralized coordination (an auction) with decentralized techniques (multiple agents may run auctions). Market based methods have been used in the robot soccer simulation league, and even combined with other techniques such as reinforcement learning [40]. These methods are applicable to video games, and may be especially useful for sports games where individual character skills are well-defined, and teams need to execute a variety of pre-defined plays. In addition, these approaches are valuable to highly realistic *tactical shooter* games which require closer adherance to real combat tactics. Our focus throughout this work is on less con-

strained teams. The reactive teaming approach could be used in concert with some of these methods through the use of the *command* transfer and request policies. We do not focus on these types of teams, as we are more interested in introducing team coordination for contrained situations which would otherwise have only minimal team interactions.

2.2.6 Agent Failure Detection

Another important aspect of intelligent agent control is detecting failures. Failure detection can be important for repairing agents that are exhibiting errors, or recognizing when a team task should be reallocated. Plan-based methods need to detect when a plan has been invalidated, and some (such as goal-driven autonomy) attempt to identify not just when the current plan is failing, but also *why*, so that a new goal can be chosen.

Several attempts have been made to identify the types of failures that can occur and ways to detect them. Watanabe et al. provided a classification of different failure modes in a subsumption-based robotic system [83]. They classified all errors as either *local* or *global*. Local errors occur as part of the sensing process or behavior execution process, and are sometimes detectable in the robot. Global errors are detectable with additional monitoring systems. While this distinction is important, it only provides two general classes, and we have found that it is possible to identify finer-grained classes of errors.

Zang et al. use an emotion model that is affected by behavior success and failure to detect when errors in a controller have become serious, and have several defined operators to change the ABL-defined behavior [88]. Unlike the work presented here,

Zang's system relies on a concept ontology to repair behaviors. This is more flexible, but requires significantly more metadata. Lussier et al. discuss different types of errors that can arise in robotic systems and the common ways of handling these errors, but dismiss subsumption architecture as a feasible choice of controller because it is too simple [46]. On the contrary, we have found that only minor extensions are required for discovering and handling errors in subsumption-based systems.

Goal driven autonomy is another approach to handling errors in an agent controller that has been used in virtual environments [53]. The ARTUE agent is controlled with a hierarchical task network planner, and is able to reason about failures to select new goals when the controller experiences errors. It follows the goal driven autonomy framework by using a *discrepancy detector* to detect goal failures, which are then analyzed with the *explanation generator*. This explanation can be used to choose a new goal that the planner will use to decide a new control strategy.

CHAPTER 3: METHODOLOGY

This work falls into two major areas: environmental knowledge representations and dynamic extensions for behavior-based subsumption. New representations for placing knowledge directly in the environment are discussed in the first section. Our extensions to behavior-based control are discussed in the second section, first discussing our choice of behavior-based subsumption, then giving an overview of the architecture and the single-character extensions. The third section covers multi-agent extensions for behavior-based subsumption.

3.1 Environmental Intelligence

Environmental intelligence refers to elements of artificial intelligence that are part of the environment rather than part of the agent controller itself. We focus on three major advantages of putting intelligence into the environment. First, it allows agents to treat the world as a blackboard by posting information for other agents. Once the information has been placed in the environment, the creating agent does not need to manage it—transferring this information to other agents occurs through the game engine rather than agent-to-agent communication. Embedded information can also be used by the level designer to modify the behavior of AI characters in a particular part of the world without having to change the AI specification.

Secondly, environmental intelligence allows objects to be annotated with informa-

tion about how to perform actions. It may be possible for characters to sit on a couch, stand on it, lift it up to vacuum underneath, or push it over for cover; if the object is not annotated with this information, the character itself must have pre-built knowledge about couches. If a new object is then added to the world, a table, the character itself must change to be given knowledge about how to use tables. As more objects are added to the world, character AI must be updated. Attaching this action information to the objects allows entirely new objects to be added to the world without AI changes—so long as the agent knows how to perform the action, it need not be aware of the specific object type.

Finally, any knowledge that is designed as part of the level or the objects in the level is knowledge that is not necessary for the agents to maintain. If a particular part of the world is noted as dangerous, the character can use that knowledge directly rather than mapping out the locations of recent deaths or looking for sniper roosts to discover the fact. Furthermore, it allows the creation of more general agents. A guard character that is designed to use an object that is capable of the "threaten" and "shoot" actions to defend its post may be used in different situations. This character could guard a medieval castle with a longbow, an army post with a rifle, or a space station with a phaser.

3.1.1 Influence Points

First, we consider representations that allow the world to be used as a blackboard. *Influence points* allow the addition of tactical information to the environment. Influence points are similar to the idea of influence maps, but take advantage of navigation mesh decompositions. Individual influences are placed in regions instead of using a

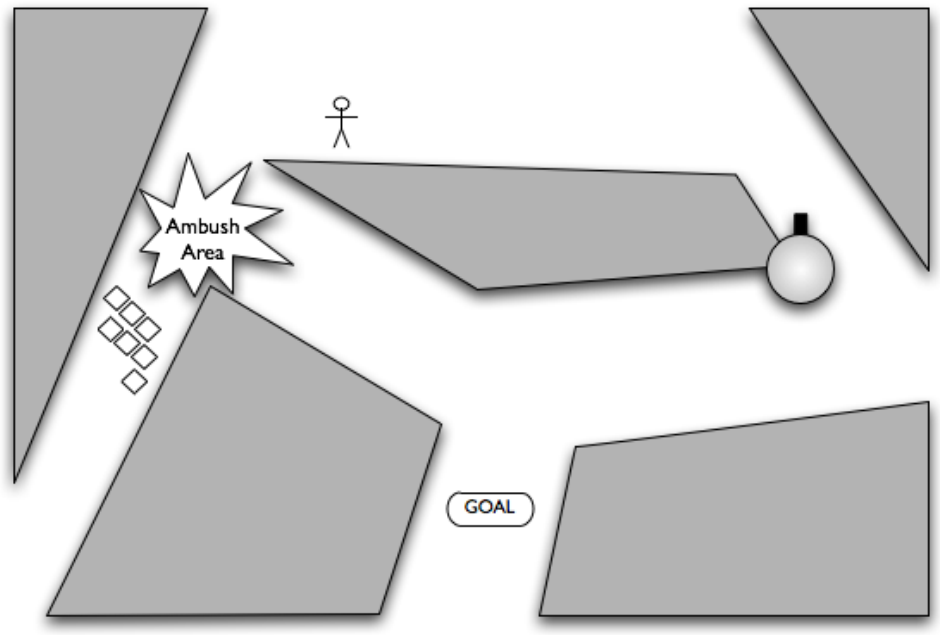


Figure 3.1: Example world with player influences in which influence mapping can provide useful tactical information.

high resolution grid map. Because a region includes an area that would be covered by a much larger number of map cells, this allows much greater efficiency than influence maps. The influence of an entity on a map may not be limited to a single region—in these cases, additional child influence points are added to neighboring regions. To demonstrate the idea, we present an example.

A character, Alice, is attempting to make it to a goal, the gold mine, to fetch resources (see Figure 3.1). Alice starts at her home base, and has a choice between two paths. The longer path is defended by a friendly turret. The other path is short, but unknown. Alice travels the short path, and is ambushed by enemy forces. After retreating from the battle, she updates her knowledge by adding an enemy influence to the area where she was attacked.

Now the path passes through an area where the enemy is powerful. The longer

path, on the other hand, passes through an area that is well protected by a defensive turret. After running her pathfinding algorithm a second time, Alice finds that the least expensive path (incorporating chance of injury) is the longer path that passes the defensive turret. Alice now follows the longer path, and makes it to the gold mine successfully, returning valuable resources to home base.

In this example, if a grid-based technique is used, Alice must spend a nontrivial amount of time inserting influence into the map, and potentially updating the large number of cells affected by the enemy and ally forces. If a navigation mesh is used, however, the insertion and update steps could be far less expensive.

3.1.1.1 Insertion

The first operation on influence points is insertion. This adds an influence point, as well as any child influence points, to the navigation mesh. In our example, as Alice heads towards its goal, she discovers the ambush point. After she escapes, she then adds an influence point to the region where she was ambushed. This initial influence point is the *prime point*, and the initial region is the *prime region*. The influence point stores its information type (from a defined set of types), a strength value, a unique identifier, and the exact point source of the threat.

The insertion algorithm can be seen in Algorithm 1. First, the region of the point is discovered. This point is appended to the region's list of influence points, and then the strength of the influence at each gateway is checked. If the strength is still high at the gateway, the connected region is retrieved and added to the visited list. A child influence point is then added to the connected region. This child point will use the midpoint of the gateway as its location.

Algorithm 1: AddInfluencePoint Algorithm

```
Region r = findRegion(influencePoint);
r.influencePoints.append(influencePoint);
foreach gateway in r.gateways() do
    if strength(influencePoint,gateway) > 0) then
        connectedRegion = gateway.getRegion();
        visited.append(connectedRegion);
        connectedRegion.addChildPoint(gateway, influencePoint,visited);
```

Adding a child point is a similar operation, as seen in Algorithm 2. First the distance from the gateway to the the *prime* influence point is calculated. The distances between regions (gateway to gateway) are already pre-calculated. The child point is created with this total distance value and a reference to the prime influence point. Then it is added to the region's list of influence points. Finally, the strength of the influence at each gateway in the region is checked. If the influence is strong enough, the next region is added to the visited list, and addChildPoint is called again with the gateway and the prime influence point.

Algorithm 2: AddChildPoint Algorithm

```
d = distance(gateway,influencePoint);
ChildPoint c = new ChildPoint(influencePoint,d);
region.influencePoints.append(c);
foreach gateway in region.gateways do
    if strength(c,gateway) > 0 then
        connectedRegion=gateway.getRegion();
        if connectedRegion ∉ visited then
            visited.append(connectedRegion);
            connectedRegion.addChildPoint(gateway,influencePoint,visited);
```

Note that the location of all child points is the midpoint of the gateway closest to the prime point, but the child point is appended to the region's list of influence

points.

In practice, to be certain that influence points are added to the closest gateway of a region, if there are cycles in the navigation mesh, a priority queue should be used, and child points added in order of where influence is strongest. The visited list insures that the influence point is never added to a region more than once. Insertion is $O(n)$ in the number of regions, since no more than n points total must be added to the mesh. Insertion is the most expensive operation, but is still significantly less expensive than influence map insertion, because the number of regions is far smaller than the number of map cells.

3.1.1.2 Updates

At times, the strength of a particular source of influence may need to change. For example, our agent may return to the ambush point with a strike force, and eliminate half of the enemy force at that location. The strength of that source now should be reduced, since the force is less of a threat.

Decreasing the strength of an influence point is very fast: first, find the the prime point in the region's influence point list, and then change the strength value. While the child points could be updated at this time, it is not necessary. Because each child point includes a reference to the prime influence point, the child points can be updated when they are used by agents. This avoids extra work to update points that may never be accessed before the next update to the influence point. This makes the influence point update a constant time operation. A similar operation with an influence map is $O(n)$ in the size of the map.

If the influence update actually *increases* the influence, this advantage is lost. The

cost of updates can be reduced, though. When creating influence points, instead of basing the decision of when to stop placing children on the actual strength of the prime influence point, an estimated maximum strength can be used if the strength is expected to increase. Even if the estimate is poor, and this maximum strength is exceeded, using this method will reduce the number of updates that require new child points to be added. A heuristic can be used that looks one region beyond each stopping point, and stores the strength value required for that region to have an influence point. Once all stopping points have been reached, a second pass can insert extra estimated child points based on the maximum heuristic strength value.

3.1.1.3 Using Influence Points

While influence point insertions are much faster than influence map insertions because the number of regions will be much smaller than the number of map cells, and influence point updates are much faster constant time operations, actually using the influence points is slightly more expensive.

If the agent is in the region of the prime influence point, it must first find its distance to the prime point, then calculate the influence fall-off $f(D_{a,p}, I)$, where $D_{a,p}$ is the distance of the agent to the influence point, and I is the strength of the influence point. $f(D, I)$ is the strength fall-off function, which will be discussed below.

If the agent is in the region of a child influence point, first the strength of the child point is compared to the strength of the prime point and updated if necessary. Next the distance of the agent to the child point is calculated. For static world geometry, the distance of the gateway to which the child point is attached to the region of the prime point should be pre-calculated, making it a simple lookup. The influence decay

$f(D_{a,c} + D_{c,p})$, where $D_{a,c}$ is the distance of the agent to the child point and $D_{c,p}$ is the distance of the child point to the prime point, can then be calculated. The calculation for a single influence point, prime or child, is still constant time.

For regions with n multiple influence points present, the agent can determine a single tactical value using the equation

$$\sum_{i=1}^{n} \alpha_i f(D_{a,c_i} + D_{c_i,p_i})$$

where α_i is a weight value for influence i.

While this is more expensive than a simple table lookup, the only additional costs are an inequality check for the child point, a possible value assignment to update the child point, and a single distance calculation. For multiple points, these three operations must be performed for each influence. The weighted sum of the influence values is also necessary with influence maps.

Because influence maps are typically recalculated on a regular basis, the additional minor costs of the influence point calculations are more than made up for by the savings in the insertion and update steps.

The influence fall-off function may be tied to attributes of the forces or objects in the world, just as with influence maps, so that within weapons range of enemy forces, the strength decreases slowly to take into account greater inaccuracy. Outside of that range, it may fall off very quickly. Other possibilities may take sight lines into account, or use a function such as radioactive decay, using a linear fall-off with distance within the prime region, and a squared fall-off in child regions.

3.1.2 Extended Affordances

The term *affordance* was introduced by Gibson to describe all "action possibilities" available in the environment [26]. For example, if there is an unoccupied chair sitting upright in the world, there is also an opportunity for sitting in the chair. In artificial intelligence, this can be used to sidestep the symbol grounding problem, at least in virtual environments; rather than define the quality of *chairness*, we merely require that anything that can be used as a chair should have a sit affordance defined.

Affordances can be directly implemented in games as entities that provide information about what sorts of actions can be taken using the objects they are attached to. Affordances may exist separate from objects in the environment. If a character possesses a hang glider, for example, it will state that it can be used as a tool to enable flight, but that it requires a launch point. A launch point affordance could then be placed at spot on a cliff edge.

In our system, basic affordances provide two key pieces of information: an action that can be taken with this object, and the *action slot* the object will fill when the action is executed. There are three types of *action slots* that an object can fill: *source*, *object*, and *target*. The *source* means that the given object (which may be a character) initiates the object execution. Usually the action source is a character. The *object* slot is used to specify that the given object is used as a tool to perform the action. If unlocking a door requires a key, then the key will be used to fill the *object* slot. The *target* indicates that the given object or character is affected by the action, frequently by changing its state. In the door unlocking example, the door is the target. The

unlock action will change the state of the door. This is represented by an affordance of the form *(unlock, target)* embedded with the door, and an affordance of the form *(unlock, object)* embedded with the key. Finally, the character must have *(unlock, source)* in its action space to be able to use the key to unlock the door.

By placing this information in the world, the character needs no additional knowledge about keys and door locks, or even the difference between a swipe card and a skeleton key. This basic concept is useful in itself, but it does not address differences in characters, or how world state may affect affordances. The character still must choose between multiple actions that are available with the item, deciding which are possible and which are appropriate for the situation. If the action space for these objects could be reduced to just the possible and appropriate actions in the given context, the character has a simpler decision to make. Simpler decisions then reduce the complexity of the intelligent controller. To further simplify character decisions, we have developed three extended types of affordances: *contextual* affordances, *multi-character* affordances, and *probabilistic* affordances. Each of these types of affordances build upon the *(action, slot)* specification of basic affordances.

3.1.2.1 Contextual Affordances

Contextual affordances are affordances that take the current state of the world and character into account. When a character queries a contextual smart object, instead of returning the full action space of the object, a contextual affordance will only include its action if certain conditions are currently true. For example, our hero, Alice, is carrying a ball. There are a few things that the ball can be used for. It can be dropped on the ground, it can be picked up off the ground, it can be thrown to

break a window or push an out of reach button, or it can be used to bonk Bob, who Alice does not like. The complete action space of this ball is then:

(drop, object) (pickup, target) (break, object)

(push, object) (bonk, object)

Now consider that Alice is in a room with her friend Carl. This room has no exit, but there is an out of reach button which she can hit with the ball. In this context, Alice really only has two choices: drop the ball, or push the button. She *could* bonk her friend Carl with it, but this would not be appropriate since they are friends. It is already in her inventory, so attempting to pick the ball up would fail. There are no windows, so there is nothing to break. If we add an open window to the world, she could now also use the ball to break the window—but it should not show up as an affordance, because breaking an open window would not be appropriate (unless Alice was a vandal). So the *contextual action space* is:

(drop, object) (push, object)

Alice uses the ball to push the button, and a secret door opens. Now, since the button state has changed, the contextual action space of the ball is further reduced, so the only option left is to drop the ball. Just as Alice thinks to do so, Bob runs into the room, and she has a better option: the *bonk* affordance becomes available, and Alice throws the ball at Bob's head.

At each step, Alice must query the ball to discover what actions are currently appropriate. Obtaining this list of appropriate affordances of an object requires a

query that examines a set of condition/affordance pairs to filter out the options that are not currently relevant. Conditions filter on the object state, the character state, and the local environment. Contextual affordances are simple to add to an existing AI system, as the same set of conditions available for building the character controllers can be used to build the contextual affordance filters. This makes adding contextual affordances to a system minimally invasive, as existing AI components can be reused. Now when using contextual affordances, the ball action-space will look like:

(condition: *haveBall?* (drop, object))

(condition: $\neg$ *haveBall?* $\wedge$ *ballInReach?* (pickup, target))

(condition: *haveBall?* $\wedge$ *lockedWindow?* (break, object))

(condition: *haveBall?* $\wedge$ *outOfReachButton?* (push, object))

(condition: *haveBall?* $\wedge$ *unfriendlyCharacter?* (bonk, object))

When a character makes a behavioral decision, it queries the game engine for currently available affordances. The engine uses the list of local objects, including the character's inventory, and evaluates the conditions on each contextual affordance. When a contextual affordance test returns true, the engine adds the related affordance to the list of possible affordances for the character. This reduces the amount of reasoning the character must perform and also reduces the number of errors that occur when characters attempt to perform actions that are not appropriate to the current situation.

3.1.2.2 Probabilistic Affordances

Contextual affordances rely on the character to reason about actions to be taken, and do not provide any information to goal-based agents as to whether taking a particular action will actually provide a helpful result. One easy fix for this problem is to tag affordances with information about what the action will accomplish—the *(push, target)* affordance on the button from the previous example could be tagged with *(outcome: (action:changeState, target:secretdoor, value:open))* to represent that the door will open once the button has been pushed. Sometimes, especially in games, outcomes are not as certain. Either an action has a chance to produce different outcomes (they may have *stochastic outcomes*), or it is unknown what a particular object will do (the character has limited knowledge).

To enrich these interactions, affordances can be extended to include probabilistic information. Probabilistic affordances include an expected outcome distribution, which provides the character a notion of the expected change in world state after the action has been executed. This is a probability distribution to allow the specification of outcomes that are uncertain.

In the previous example, we assumed that the ball Alice had was sturdy, and the button smooth. Instead, Alice may have picked up an old playground ball, and the button itself could actually be surrounded with spikes. Now when Alice throws the ball at the button, there are a few possibilities: the linked door could open with a 90% probability, the ball could hit the spikes and pop with a 5% probability, or the ball could just be too flat to apply sufficient force to push the button with 5%

probability. Now Alice will have to consider the possibility she will lose the ball, or that nothing could happen at all. If she has a long pointy stick, she may choose to use that to push the button instead. The outcomes for using the ball are conditionally independent of the outcomes for using the stick, but each set of outcomes must be created by the level designer. When using probabilistic affordances to create a stochastic *outcome*, and not just to represent limited knowledge on the part of the character, the probability distributions must be created by hand. In this case, they never need to be recalculated.

When used to represent limited character knowledge, probabilistic affordances store the character's interaction history so the probabilities can be recalculated as a character interacts with the world. This allows additional interesting scenarios if we link multiple affordances together, and is our primary use of probabilistic affordances. Imagine that our hero has just entered a dungeon room, and in this room there are 4 levers, 3 doors, and a trap door leading to the Pit of Infinite Doom. The affordances on the levers have outcomes of opening the door to the exit, the treasure, or a peckish dire badger. In addition, one of the levers triggers the trap door to the Pit of Infinite Doom. In this case, the probabilities of the different outcomes are conditionally dependent. As Alice tries pulling the different levers, she should adjust her expectations for what will happen when the next lever is pulled.

Rather than making Alice record this information herself, her interactions with the levers are recorded with the probabilistic affordances. This means that when Alice queries the game engine for the likely result of pulling a lever, it will check to see if she has pulled this lever before. In addition, it will check the other linked levers

to see how she has interacted with them in the past. This information is then used to adjust the probabilities that Alice sees. If she has interacted with the lever that activates the pit trap, this will not be reported as a likely result of pulling one of the other three levers.

So as Alice pulls the levers and observes the results, the probabilities adjust to reflect her experiences so far. If the first lever opens the exit, the probabilities for the other levers then report as 33% chance to find the treasure, 33% to encounter the monster, and 33% chance of certain death. Deciding that this is an acceptable risk, Alice chooses to open one more door. Her second attempt releases the dangerously hungry dire badger. After defeating the monster, Alice can decide that her threshold for danger has been surpassed, and she is not willing to risk the 50% chance of falling in the pit for the chance to retrieve the treasure. If she fails, and falls into the pit of infinite doom, the history of her interactions with this set of affordances is carried with her to her next life. So long as the history of interactions is maintained, no other changes need to be made to character controller to allow her to make the correct choices when she next enters this room. In effect, learning occurs, but without the need to explicitly model a learning process for the agent.

Probabilistic affordances require additional modifications to contextual affordances, and may not be as reusable as contextual affordances. When using probabilistic affordances, we create a set of contextual affordances for each object that can be activated (in our example, the four levers). These affordances are then linked together by giving each affordance a reference to the related affordances. Each affordance then must also be tagged with a set of *outcomes*. In the example with the lever, this means

that each of the levers is tagged with a single outcome—the true outcome for the given lever. Finally, a history is added to each object. This history records how characters have interacted with it. The history is not visible to the characters, and is only used when calculating the outcome probabilities. The history may be used per-character or per-team, depending on the game design. Note that multiple affordances on the same object could be linked together, so if a lever has two different outcomes depending on whether it is pushed or pulled, this can be represented.

When Alice queries a given lever, it checks the possible outcomes of all linked affordances. The lever then discards any outcomes that have been achieved by Alice, based on the interaction histories. The remaining outcomes are then reported as having equal probabilities. At this time, we do not perform additional weighting of the outcomes, so outcomes are reported as a uniform distribution. The advantage to using a uniform distribution is that it is not necessary to specify any probabilities when the affordance is created, as they are calculated directly when characters query the affordance.

3.1.2.3 Multi-Character Affordances

Some actions are only possible with the assistance of others. Affordances as discussed so far only support a single character; one character queries the affordance for the details of the action that are possible, and then can attempt it alone. Multi-character affordances extend contextual affordances by allowing two or more affordances to be linked together. This link is not used to adjust probabilities of different outcomes, as in probabilistic affordances, but to specify that a given action requires multiple characters to execute. This link then allows characters to work together

without an internal model of team coordination.

To return to the levers example, a multiple agent affordance might be required to pull a particularly rusty lever. This lever has three action affordances, two of which are required, and the third is optional. One agent does not have the strength to pull this lever, but two can. However, if only two agents attempt to pull the lever, it only has a 60% chance of working. A third agent can join and improve the chance of the lever pull opening the door.

As in probabilistic affordances, creating a multi-character affordance requires creating two or more contextual affordances. The first affordance is marked as *required*, and is marked as a normal action from the attached object's action space. Each linked affordance in a multi-character affordance is either *required* or *optional*. Required affordances must be filled for the action to be successful while optional affordances may enable it to complete faster, or provide a better chance of success for results that are uncertain. Secondary affordances, after the first required affordance, are marked as having an *assist* action. If the action/animation for the secondary affordance is different from the primary, it is reported with a different action. In all cases, secondary affordances use contextual triggers that prevent them from being reported unless the primary slot is reserved. Optional affordances are not reported until all required affordances have been reserved. A character which is designed to prioritize *assist* affordances will then try to help when a teammate undertakes a large task.

Another addition is necessary for implementation of multi-character affordances. Since exeuting the affordance requires multiple characters working together, the characters must *reserve* the affordance during their decision cycle. Once the decision cycle

completes for the team, the AI engine then attempts to execute all reserved affordances.

3.2 BEHAVEngine and Dynamic Subsumption Extensions

This section describes the BEHAVEngine architecture. The first subsection introduces the concept of *representational complexity*, and provides background on the choice of behavior-based subsumption as the architecture used for BEHAVEngine. The second subsection introduces the basic BEHAVEngine architecture.

3.2.1 Analysis of Reactive Methods

Different agent architectures have different strengths and weaknesses. Qualitatively, we may talk about the difficulty of creating a specific behavior or type of behavior using each model. Our initial choice of subsumption architecture was based on an earlier pilot study that showed individuals with no AI experience found subsumption architecture easier to manage than finite state machines. This was an unexpected result, as the expressiveness of the architectures is similar. User studies with the BehaviorShop interface confirmed that novice users had very little difficulty grasping the concept of subsumption architecture, but we had no metric for comparing the differences between different representations [29].

To demonstrate these differences, we present a scenario. While this scenario is very simple, it is possible to visualize each controller type and it sufficiently shows the difference in complexity of each architecture we are demonstrating. A simple scavenger agent lives in a world with three percepts: hunger, presence of food, and presence of a predator. This scavenger is only aware of three actions: search for food, eat food, and flee from predators. Its preternatural ability to sense food allows it to

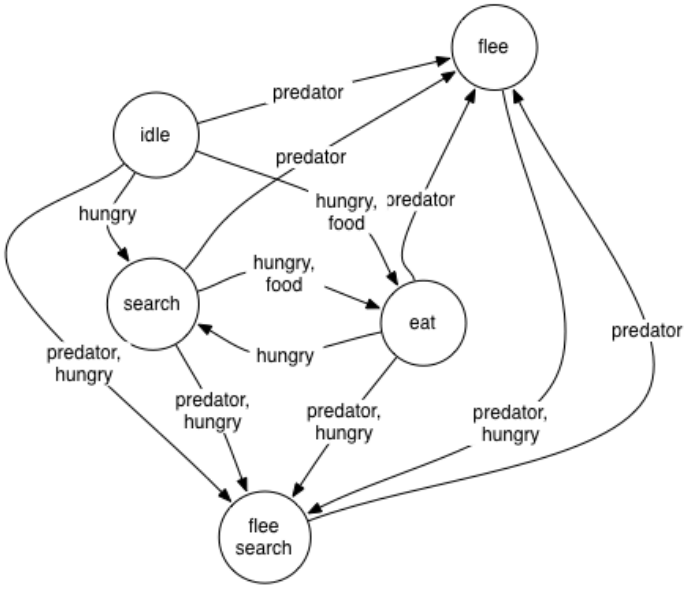


Figure 3.2: Finite state machine representation of the scavenger agent. Note that only a few of the transitions are included for the sake of clarity. The total number of transitions, including those omitted, is 25.

both flee from predators and search for food at the same time. A basic controller for this creature as a finite state machine can be seen in Figure 3.2. Since basic FSMs do not allow concurrency, five states are required. The creature can be idle, perform any of its three actions alone, or perform both search food and flee at the same time. This requires 25 transitions, as every state is accessible from every other state.

Figure 3.3 shows the same scavenger agent, but presented as a HFSM. The HFSM reduces the overall complexity of the agent by creating three high level states instead of five. The *find food* and *flee* superstates are each composed of two state FSMs. While the total number of states increases, the overall number of transitions is reduced to 17, since the search, eat, run, and run/search states are no longer fully connected. The overall representation of the agent is simpler to understand, and it is easy to see that the super-states could be treated as separate modules to be inserted into other

characters.

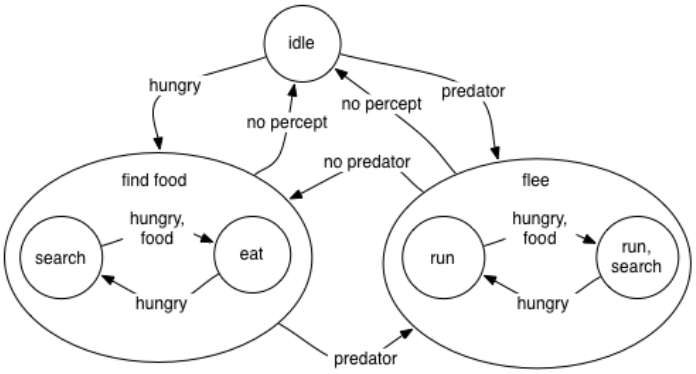


Figure 3.3: Hierarchical finite state machine representation of the scavenger agent. Self transitions are not included for clarity. The total number of transitions in this case is 17.

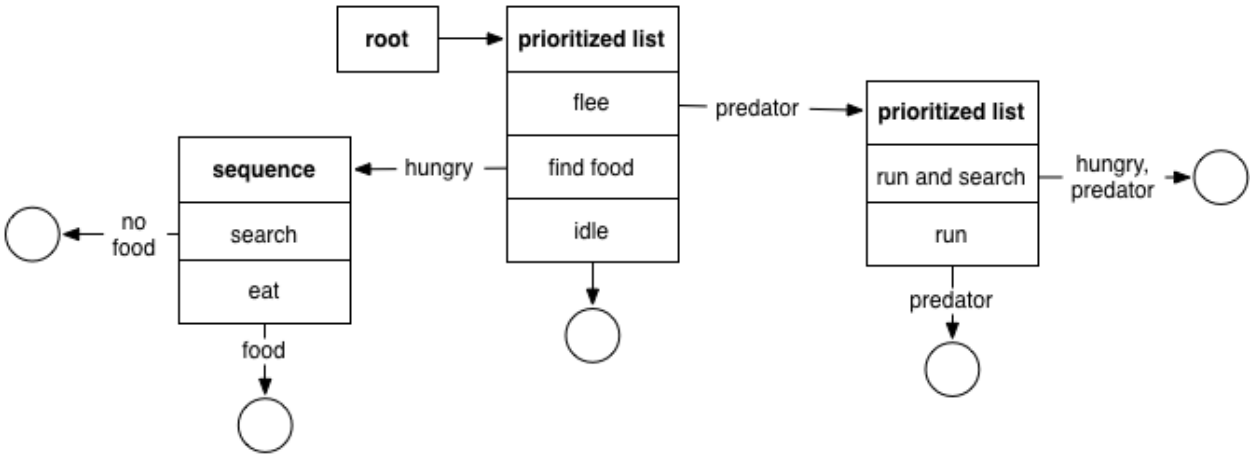


Figure 3.4: HFSM-style behavior tree, composed of prioritized lists and sequences. The use of these structures reduces the number of transitions, but the complexity of the individual elements is greater. The number of transitions is 8, but this does not account for the full complexity of the model.

As a behavior tree, this agent is simpler still to understand, as seen in Figure 3.4. In this case, the agent is a tree with four internal nodes and five leaf nodes. The leaf nodes represent the primitive behavior to be executed, while the internal nodes represent decision points. The main decision to be made is first to choose from the

prioritized list providing the high level behaviors of flee, find food, and idle. If a predator is seen, flee is chosen, and another prioritized list decision is made. If no predator is seen, it falls through to the find food branch, which will activate if the agent is hungry. If the find food branch is entered, the agent will simply activate search and eat in sequence. The priorities simplify the decision that is made at each stage, and eliminate the need for transitions between siblings. There are only seven explicit transitions in this case, though the individual nodes are more complex.

HTN-style behavior trees are very similar in structure to HFSM-style behavior trees, but may be executed differently. Depending on implementation, rather than performing conditional checks in the selector nodes, conditionals may be checked at the leaf nodes. Execution is then performed as a greedy search through the tree. While execution is different from the HFSM-style behavior tree, the structure of the tree itself is very similar, and so will be treated the same as the HFSM-style behavior trees for the sake of this analysis.

The subsumption agent in Figure 3.5 is even simpler. The available actions are listed in order of importance, and each is assigned an activating stimulus. The circles to the right of the layers represent the subsumption policy. The top layer cannot run simultaneously with eat or idle, so it must provide two override policies. The eat layer cannot run simultaneously with search or idle, so it also provides two. Finally, the search layer only specifies that it cannot run simultaneously with idle. This is a total of five suppression policies that must be specified, and three stimulus triggers (since the bottom layer must always be active if no other layer is active).

More complex subsumption architectures can be created using hierarchical sub-

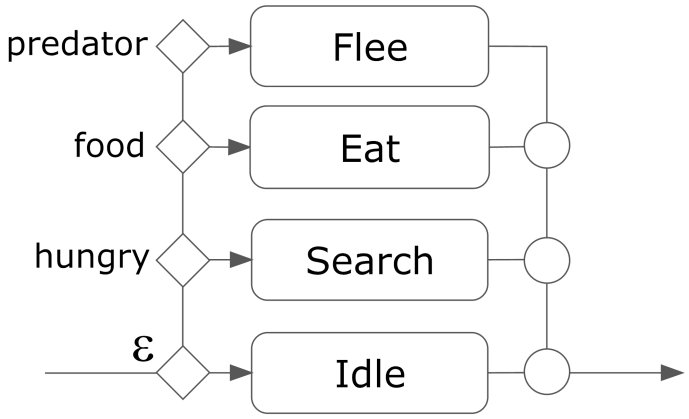


Figure 3.5: Subsumption architecture representation of the scavenger. Counting triggering conditions and the individual components of the subsumption policies, there are 8 transitions. Three for the *predator*, *food*, and *hungry* triggers, two for the *flee* subsumption policy (since search can run concurrently), two for the *eat* subsumption policy, and one for the *search* subsumption policy.

sumption, which allows a single top-level layer to be composed of two or more sub-layers. To demonstrate hierarchical subsumption, a more complex example is needed. For this example, the creature being controlled can see much further than it can reach, so if food is seen, it may be far away. To control this agent, when food is seen, first the food must be approached, then it can be eaten. This approach reduces the number of overrides that must be specified; if this was added as an additional layer, up to an additional four subsumption override policies would need to be added, but by making it hierarchical, only a single additional policy is required.

3.2.1.1 Quantifying Representational Complexity

Representational complexity can be measured by considering the number of architectural elements which must be specified when creating a given agent. In addition to the number of elements, the complexity of the elements themselves must be accounted for. For FSMs, HFSMs, and both hierarchical and non-hierarchical subsumption ar-

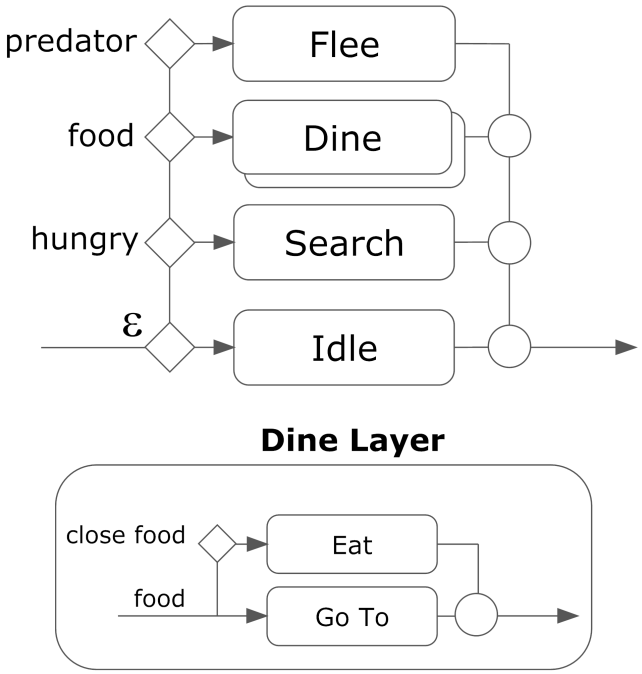


Figure 3.6: Hierarchical subsumption architecture representation of the scavenger with limited reach. The two layers composing the *dine* hierarchical layer only add two transitions: one for the *close food* triggering condition, and one for the *eat* subsumption policy overriding *go to*. In this case, the *eat* subsumption policy could be omitted since go to will not generate any actions once the agent has approached the food.

chitecture, the elements are of equivalent complexity. Behavior Trees present an additional challenge because individual elements may be significantly more complex.

In all cases, architectural elements can be divided broadly into two categories: behavior entities and relationships. Behavioral entities represent the modular behaviors used to generate actions, while relationships represent the transitions from one behavior to another. For FSM, subsumption, and behavior tree-based architectures, these map directly to preprogrammed behaviors. Hierarchical architectures include compound behaviors as entities as well, though they may only provide an abstraction and not instantiated as behaviors in code. Complexity is measured as the growth rate

of relationships in terms of behavior entities.

3.2.2 Representational Complexity of Reactive Methods

In order to make our intuitions about representational complexity more precise, we introduce some notation. We assume that we have a set of percepts P and a set of actions A that the agent can perceive and perform respectively. It follows that the specification of a reactive agent is a mapping of the form $P*$ to $A*$ where $P*$ and $A*$ are power sets with cardinality $2^{|P|}$ and $2^{|A|}$ respectively. Note that, in this particular setting the agent is able to perceive one or more percepts and able to simultaneously perform one or more actions. This is quite typical in the case of agents in a virtual world of computer games and training simulations. A simple example of this is an agent that perceives a friend wants to exchange greetings (waves or says hello) while walking towards a goal.

Given this setting, our intention is to investigate the complexity of the agent specification. In order to introduce a metric for such representational complexity, we simply count the number of entities and relationships between the entities for a particular framework like finite state machines or subsumption architectures. In the case of FSMs, states are entities, and state transitions are relationships. For subsumption controllers, layers are entities while relationships are the subsumption policies between layers.

We can use these definitions to compare the representational complexity of finite state machines and subsumption architectures by comparing the number of entities and relationships in an agent specification. As far as the number of entities are concerned, it is easy to see that a finite state machine would require one state for every

possible subset of actions. The number of entities would therefore be equal to $2^{|A|}$. Of course, in certain cases, it may not be possible to perform all actions simultaneously, so this is a worst case estimate. In contrast, observe that subsumption architectures would require a total of A layers as they inherently allow for parallel execution of actions. As far as relations among entities are concerned, as there are $|P|$ percepts and the agent can perceive any subset of them, a finite state machine will have a total of $|P| \times 2^{|A|}$ transitions. HFSMs may have an equally large number of states, and (in the worst case) may grow as large as a standard FSM if no abstraction of the behavior is possible. The best case for an HFSM may be better than that of its FSM equivalent, assuming that the specified behavior can be decomposed hierarchically, but the main improvement for the HFSM is in the number of transitions in the model.

In the case of HFSMs the explosive increase in the number of transitions is controlled by grouping states into super-states. The intuition is to group states that have relatively high intrastate transitions and relatively low interstate (at the meta-level) transitions. By grouping states in this manner the number of transitions to be specified can be largely reduced. This reduction in the number of transitions is highly domain dependent as in certain problem domains it is possible to exploit the natural hierarchy among states, while in others it may not be possible to group states. So in the worst case scenario there might be as many states as a FSM, but in the best case scenario it is possible to organize the underlying FSM as a structure that largely resembles a binary tree. In such a case it is possible to group states such that each meta-state has two child states, resulting in a drastic reduction in the total number of transitions.

To get a quantitative measure of the number of transitions in such a HFSM consider the following reasoning. First, note that there are three distinct types of transitions, parent-child transitions, sibling transitions, and self transitions. If the underlying finite state machine contains a total of n states and is organized as a binary tree, there are a total of $2(n-1)$ parent-child transitions. In addition to this, we have a total of $(n-1)$ sibling transitions and a total of $n-1$ self transitions. Thus the total of $4n-4$ number of transitions, which is a large reduction from the n^2 transitions in a FSM. It is important to note that this is a best case scenario and, in practice, the number of transitions in a HFSM is between $4n-4$ and n^2, depending on the problem domain.

Behavior trees have more complex elements than HFSMs, and the behavior tree technique differs from the others discussed because it is as much a set of common design and engineering principles as it is a control method. Though behavior trees may differ greatly across implementations, the main building blocks are still behaviors, conditionals, and parent/sibling relations. The parent/sibling relations are compounds instead of simple relations, and can be reduced. The *sequence* relation can be reduced to a parent relation and a set of direct sibling relations, so that for a parent with c children, there are c relations. *Selectors* are similar, and many of the decorator patterns such as *preconditions* and *parallels* have the same attributes. Therefore, while individual implementations will vary, the core behavior tree technique will have approximately $n-1$ relations for n entities in the best case balanced binary behavior tree. Note that a behavior tree with n entities may have fewer behavior or action entities; because conditionals are frequently stored as nodes in the tree, they are counted

as entities rather than relations.

In the case of subsumption architecture, each layer will have a percept and action but the major component of the relationships among the layers is that of the override between the layers. If every layer could override every other layer, then the number of relationships, $|R|$, is given by 1.

$$|R| = |A|^2 + |A| \tag{1}$$

Since no layer can override itself or the layers above it, we actually have a smaller number of relationships. Specifically, we subtract $|A|$ overrides from the base layer, $|A| - 1$ from the second layer, $|A - 2|$ from the third layer, and so on. In addition, we subtract 1 because the base layer always trigger never needs to be specified. So the actual number of relationships is given by Equation 2.

$$|R| = |A|^2 + |A| - 1 - \sum_{i=1}^{|A|} i \tag{2}$$

If we combine the trigger and override terms:

$$|R| = |A|^2 - 1 - \sum_{i=1}^{|A|-1} i \tag{3}$$

Observing that the summation of consecutive numbers is given by Equation 4, we can perform substitution to find Equation 5.

$$\sum_{i=1}^{n} i = \frac{(1+n) * n}{2} \tag{4}$$

$$|R| = |A|^2 - 1 - \frac{(1 + |A| - 1) * (|A| - 1)}{2} \tag{5}$$

Combining terms gives us a final expression for the number of relationships in the subsumption architecture, as seen in Equation 6.

$$|R| = |A|^2 - \frac{|A|(|A| - 1)}{2} - 1 \tag{6}$$

So a two layer subsumption agent has two relationships, a three layer subsumption has five, a four layer subsumption has nine, and so on.

Consider the intelligent agent described earlier (see Figure 3.7). It has four percepts (including the null percept), and four predefined actions. Given these basic percepts and actions, the agent design has a number of key features that affect the representational complexity. First, note that each of the layers maps percepts to actions. For example, layer L3 maps the percept food onto the action eat. Furthermore, note that the layers are prioritized. If we assume that we do not want any layers to execute concurrently, Layer L1 is overridden by layer L2, which in turn is overridden by layer L3. Finally, layer L3 is subsumed by layer L4. Also note that layer L1 does not have a percept and corresponds to a default action that is performed when no percepts are received by the agent.

This simple agent designed using behavior-based control has the following overall behavior. When there are no percepts available, it maintains an idle state. When the L1 layer is triggered by hunger, the agent explores new regions. In the case where the agent perceives food, the L1 layer is overridden by the L2 layer, and the agent consumes the food. If the agent perceives a predator, the L3 layer is triggered,

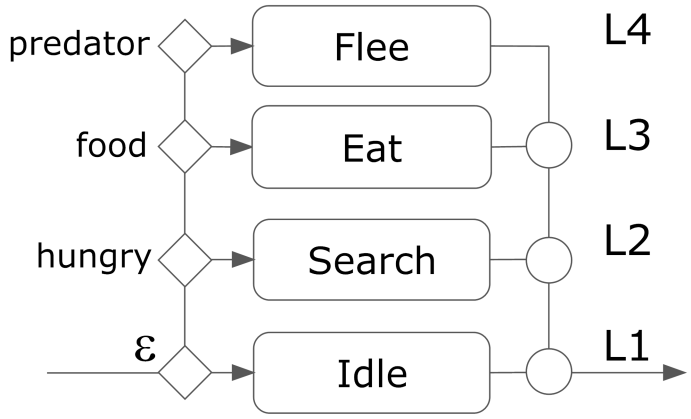


Figure 3.7: Subsumption architecture example

the L2 layer is overridden, and the agent flees from the predator. Note that the prioritization of the layers is the most important part of the agent definition and that higher layers can selectively override lower layers (combinations of layers producing blended actions are permissible). In addition, the L3 layer, in overriding the L2 layer, does not necessarily override the L1 layer.

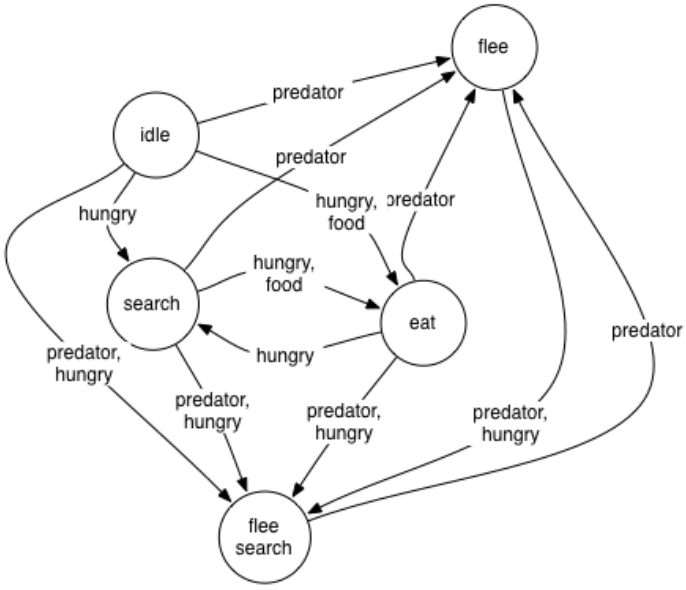


Figure 3.8: Finite state machine example

While the example discussed here is only for the purposes of illustration, it does demonstrate the key aspects of behavior based control, namely, the mapping of percepts to actions via layers, the prioritization of layers, and the ability to override or combine actions from multiple layers. Subsumption architectures have a number of advantages over the use of other architectures for game agents, such as finite state machines (FSMs), hierarchical finite state machines (HFSMs), and behavior trees. Behavior-based control is inherently parallel, as multiple active behaviors can be run at once. The representational complexity of a behavior-based control agent, which is an important consideration for the agent authoring process, is far lower than other architectures. As shown in 3.2, the corresponding finite state machine for the previous example requires many more transitions. If multiple behaviors are allowed to be active at once, the finite state machine becomes increasingly more complex, as each allowable combination of behaviors requires an additional state. HFSMs and behavior trees reduce this complexity over simple FSMs, but still require more complex models to represent the same agent due to the increased number of states. The reduced complexity of behavior-based control makes it simpler and faster to create intelligent agents.

If *hierarchical* subsumption is compared to hierarchical finite state machines and behavior trees, an additional improvement is seen. The number of transitions for this case is better than HFSMs. To correspond to the best case FSM, consider a *binary subsumption*. In a binary subsumption architecture, every layer is a hierarchical layer composed of two layers (so that if the layers are depicted as a tree with parent/child relationships, it would become a binary tree).

In this ideal case of a binary subsumption, the only transitions are between siblings. Because one layer of each sibling pair is the base layer, it has no transitions. This leaves only one override relationship and one trigger condition per sibling pair, creating n transitions in hierarchical subsumption. Like HFSMs, the worst case is when no hierarchical organization is possible. In this case, the complexity is identical to standard subsumption. It is also important to note that this best case is not a realistic scenario, and it is difficult to estimate an average subsumption controller, so the expected number of entities will generally be higher.

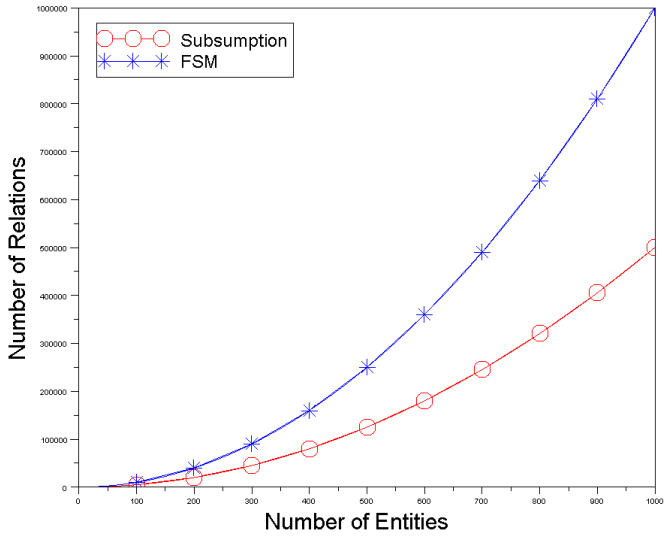


Figure 3.9: Growth of non-hierarchical models. Number of relationships between entities vs. the number of entities. Y scale is 0 to 1,000,000

The growth of the upper-bound worst case number of relationships (transitions or overrides) between entities with the number of entities is visualized in Figure 3.9. Figure 3.9 compares non-hierarchical finite state machines and subsumption architectures. For FSMs, the worst case is a fully connected controller, while for subsumption,

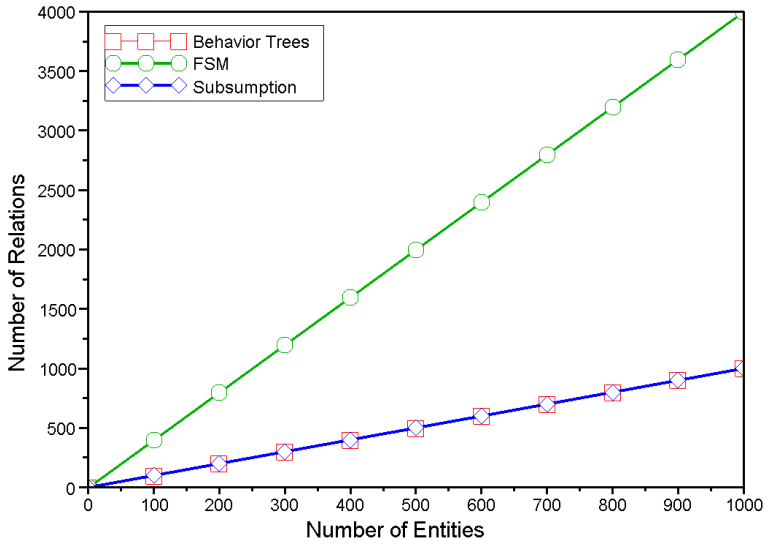


Figure 3.10: Growth of hierarchical models. Number of relationships between entities vs. the number of entities. Y scale is 0 to 4,000.

the worst case is a controller in which each layer fully subsumes all layers below. Best case for each architecture is simply the number of entities. This is equivalent to a FSM that runs a series of actions and stops, or a subsumption controller that allows every layer to run at the same time.

In Figure 3.10, the hierarchical models are assumed to be binary trees, which provides the best case number of relationships for the hierarchical models. The hierarchical layers are assumed to be fully connected (two transitions for HFSMs, one override relationship for hierarchical subsumption). Hierarchical models that are poorly balanced have relationship growth similar to their non-hierarchical counterparts, except for behavior trees, which do not degrade strongly from their best case (though in practice, behavior tree complexity will grow in other ways). Note that the growth of entities as A increases will produce higher numbers of entities for HFSMs than hierarchical subsumption, even though the relationships for an ideal HFSM grow slower

than the relationships in non-hierarchical subsumption.

3.2.2.1 Practical Implications

Representational complexity should be viewed as a guide to help choose between reactive architectures. Each of the architectures considered have similar capabilities,but models are built very differently. Finite state machine architectures provide a very simple set of elements and paradigm for creating intelligence, but if large agents are required, the complexity of the model may grow too great. Behavior trees provide a great deal of flexibility and expressivity for advanced users who are trained in the use of its decorator patterns and other features, but are not as easily accessible to novice users.

Furthermore, representational complexity may affect models created in other ways; if models are learned, there is a trade-off between the simplicity of the input grammar of the model and the size. It may be more difficult to learn behavior trees than finite state machines because there is a larger set of possible options for each element of the model, but the final model will have fewer entities. This aspect of representation needs to be explored more fully to consider the complexity of the individual elements before any strong conclusions can be made.

Other factors may also affect the choice of architecture to be used. FSMs are simple to implement, and FSM/HFSM-based systems make it easier to create sequences of actions than subsumption-based systems. This allows tight control over agents in situations where scripts might otherwise be used, but more responsive behavior is desired. In some cases FSMs may have fewer relations than subsumption controllers, such as when the controller is primarily composed of state sequences that rarely

branch, as may occur with scripted characters—in these cases, FSMs are a better choice than subsumption.

Subsumption provides inherent parallelism and lower overall complexity in the model, but there are still some concerns about managing emergent behavior. If the action model of agents being created is simple, emergent behavior is less of an issue, making subsumption a good choice for novice users. The behavior tree representation is a commonly used standard throughout the game industry, making it a widely understood architecture. If expert users are the target, behavior tree and HFSM editors can be quickly implemented using standard tree control widgets available in most graphical toolkit libraries. These editors will be very basic, but provide a substantial improvement for users who are well-versed in the architectures.

Many factors do contribute to the choice of architecture, but representational complexity can provide a useful metric of the difficulty involved in managing agent definitions. Based on the goals for our character engine, we chose to use the subsumption architecture as the basis for BEHAVEngine. Two major factors affected this choice. First, the subsumption architecture provides inherent parallelism, which we identified as an important aspect for our characters. Second, BEHAVEngine is also designed to be used with the BehaviorShop character builder [29]. More information on character and behavior builders in general, and BehaviorShop in particular, can be found in Appendix A.

3.2.3 Hierarchical Subsumption Architecture

Our primary model for agent control throughout this work is a hybrid behavior-based subsumption architecture. We have augmented the basic subsumption model

in several ways to allow the use of the higher level symbolic concepts generated by our simulation environment. We also chose to work with hierarchical subsumption, as it allows better abstraction, reusability, and a more powerful representation. Elements from cognitive modeling have been added to our subsumption architecture as well. These provide a perceptual model, an action model, and a memory model (see Figure 3.11). Some very limited planning methods are also used.

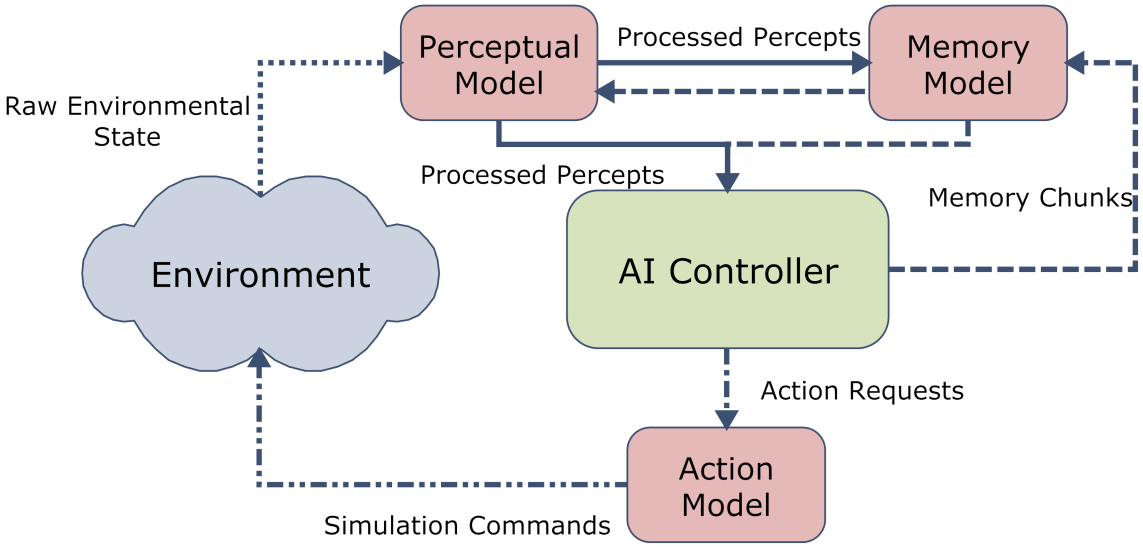


Figure 3.11: The BEHAVEngine Architecture.

The perceptual model accepts raw environmental state information from the simulation and processes them to create symbolic perceptions that are used by the agents. This includes filtering so that agents do not have perfect information, calculating commonly used values, and storing some perceptual information in the memory model to provide a short history. Agent capabilities and effectors are represented in the action model. This provides an abstraction layer over the direct interface to the simulation environment. The memory model provides a working memory for behaviors.

Behavior state is stored in memory chunks that allow arbitrary types of data.

At the core of our architecture is the subsumption controller. Agents are built up from a collection of triggers and behaviors. Percepts are received from the perceptual model to activate the triggers. Triggers may also be activated by items in working memory. The percepts that cause a trigger to fire are passed on to the connected behavior. Actions requests are generated by the behavior and sent to the action model. The action model checks the feasibility of the action before placing it in a queue. After all behaviors run, the subsumption controller then evaluates items in the queue from highest to lowest priority, and discards any low priority requests that conflict. In this way, only actions that will actually be executed are fully processed. The requests are fully evaluated by the action model and passed on to the simulation.

3.2.3.1 Perceptual Model

The perceptual model receives raw world information from the simulation. This is composed of dynamic and static information. Information about other agents, objects, and other dynamic entities is updated every game tick. Static information about world geometry is read from the navigation mesh as needed, as well as from other sources such as a database of object information. These sources are treated as part of the memory model, and are used to build perceptual information. Percepts are generated by filtering this information according to the perceptual abilities of each agent being controlled. The amount of filtering required can be reduced if visibility filtering already occurs in the game engine. Some of the information that may be added to the raw percepts include the distance of the object or agent from this agent, or a relationship value between two agents based on past events. Generating some of

these values requires storage of prior percepts, which is enabled through the Memory Model.

Percepts are divided into seven types: objects, agents, communication, locations, actions, world events, and vehicles. Objects provide information about dynamic objects in the world. Dynamic objects are any objects that are movable and can be acted upon, as opposed to static geometry, with which agents have limited interaction apart from collisions. Agent percepts represent other agents in the world, including the player. Messages between agents, or the player and the agent, are conveyed through communication percepts. Location percepts are used to represent regions or coordinates in the world along with symbolic information about the location. Actions are transitory percepts to represent actions taken by other agents in the world. The world event percept is used in limited circumstances to provide information about global events such as earthquakes. Finally, the vehicle percept is a special type of object percept to describe dynamic objects that the player can enter and use to navigate through the environment.

3.2.3.2 Memory Model

The memory model provides storage for past percepts, agent attributes, and arbitrary memory chunks. Past percepts are important, as this information allows the perceptual model to calculate whether an object is approaching, or keep track of the last known position of an object or agent that has since left the agent's line of sight.

Agent attributes are values that may affect decisions made by behaviors or how the action model generates actions. This may include basic information such as the size of the agent (which affects where the agent can go) or more complex personality

traits such as bravery/cowardice and cultural parameters.

Memory chunks provide storage for behavior state that may be shared between different layers. These chunks are initially created by particular behaviors, and may be modified by higher priority layers. This allows high priority layers to modify the operation of lower priority layers, or provide information such as action targets to lower priority layers. We provide four types of memory chunks: percept, numeric, string, and point. Percept chunks store world percepts of any type. Numeric chunks store floating point or integer values. String chunks store character strings, while point chunks store three-dimensional coordinates.

3.2.3.3 Action Model

The action model provides the functionality to turn the action requests generated by behaviors into simulation commands. A behavior can create a request specifying a particular action and a percept representing the object it should act upon. The action model handles the particulars of the simulation environment to generate a correctly formatted command that can be passed on. In addition, the action model provides the details of steering and navigation so that behaviors can avoid implementing global and local navigation. This approach can make higher-level cognition much easier to manage, as the designer does not need to focus on generating action commands or combining actions carefully. Many algorithms exist for path planning and obstacle avoidance, and these can be implemented directly in the action model.

The action model also allows different types of agents to be implemented at different levels of abstraction. For example, depending on the action model, behaviors may output direct vector controls that translate to motor commands, or take a more

abstract approach and provide goals for path planning through a navigation mesh.

3.2.3.4 Subsumption Model

The subsumption model at the core of BEHAVEngine allows simple layers, compound layers, and hierarchical layers. Each type can be activated with one or more triggers combined with conjunctions and disjunctions. The subsumption policy allows layers to completely subsume specific lower layers, subsume only specific modalities, or allow all layers to execute.

Simple layers are composed of a single behavior. Behaviors are typically either direct perception to action mappings or finite state machines, though this is not enforced. Behaviors store state in the memory model as chunks so that state may be modified by higher priority behaviors. Completion and failure are also stored for behaviors. Completion allows discrete behaviors that may take more than one time step to execute. This prevents a behavior from deactivating before it fully executes. Failure information is also important, as it allows the behavior to notify the agent when an action cannot be performed and abort gracefully (such as when an agent attempts to pick up an object that disappears between the time the behavior activates and the time the agent reaches the object).

All layers execute actions by making requests to the action model. The action model handles the process of transforming the action into a valid command for the simulation. When a layer is activated, it may suppress any lower layers to prevent them from running. This is the subsumption policy.

Compound layers are composed of multiple behaviors linked in serial. When a behavior completes, the next execution of the layer will continue with the next behavior

in the sequence. After all behaviors complete, the layer will be marked as complete, and on the next execution will return to the first layer in the sequence.

Hierarchical layers are composed of several sub-layers that are executed as an additional subsumption controller. The subsumption policy for the parent layer applies first so that the child layers need only specify subsumption policies for their siblings.

3.2.3.5 Affordances

Agents need information not just about where objects are in the world and how to navigate, but also how objects can be used. Rather than hard-coding information about objects and actions, this information is loaded from object descriptions at run time. Objects provide specifications of what actions they can support, and this information is combined with descriptions of the agent models being used. This is combined with the affordances provided by the environment to create a more dynamic simulation.

3.2.4 Failure Detection Extensions

The behavioral errors that can be generated by a failing reactive agent are as numerous as the number of behaviors that can be expressed in the architecture. Fortunately, the general *types* of errors that are generated by reactive controllers fall into a small number of distinct classes. We have identified four classes of errors: activation failures, capability failures, behavior interaction failures, and environmental failures. Each of these failures can be detected in a well-defined manner, though some are more expensive than others to detect.

Monitoring for failures adds additional metadata to each layer. This ranges from a simple value added to the behavior state to additional sets of triggers. Behaviors must

be capable of returning an error value to convey the success or failure of behavior execution. An important aspect of this metadata is that it can be constructed from existing components. There is no need for major architectural extensions to support new constructs.

Monitoring for failures can be achieved by adding a single layer with a trigger that always returns true. By adding this as the highest priority layer and giving it a subsumption policy that does not override any lower layers, the failure monitoring component is guaranteed to execute. This does require that the architecture is designed with reflection capabilities in mind, as the layer must be able to check the status of all other behavior layers for failures.

3.2.4.1 Activation Failures

Activation failures are the simplest types of failures to detect. These failures occur when a component of the controller fails to execute. This indicates that anticipated conditions never occur or that a higher-level behavior is preventing the execution of the layer. Activation failures can be easily detected in both virtual characters and robots, as even in robots they depend entirely on software state without consideration of hardware failures. While they are simple to detect, they may be unreliable if a given behavior is not expected to execute frequently. These failures can be detected by adding an activation history to the layer state.

When activation failures are present, key behaviors will not occur in the game world. The character may appear to be idle when it should be performing an action. In the worst case, this can affect gameplay when characters do not respond appropriately to important points in the story.

3.2.4.2 Capability Failures

Capability failures occur when an agent is not capable of executing a given behavior. These directly correspond to unanticipated changes in agent state; in a virtual character, it may be an item missing from the character's inventory, or an action which is no longer allowed. In robotics, this could correspond to a hardware failure or a partial loss of power. Detection of capability failures can be achieved by adding metadata to behaviors. This metadata can be built from trigger components. While precondition triggers can also be added to the conditions for layer activation, using them to detect failures as metadata eases the agent design task and may provide better information for debugging agent failures.

If capability failures are not detected and handled, characters may execute a partial set of behaviors. This can be disruptive to game play if a character performs all the behaviors to set up for a key behavior, but then cannot perform the key behavior. An example might be a character locating and climbing to a sniper roost, but then being out of ammunition. If it is not apparent to the player why the character is not performing an expected action, it can interfere with suspension of disbelief.

3.2.4.3 Environmental Failures

Environmental failures occur when agent capabilities are correct and the behavior executes, but the behavior action fails in the environment. An environmental failure may indicate that the expectations for the behavior are incorrect given the environment (e.g., a lift behavior may fail if objects in the environment are heavier than expected by the behavior) or that another agent has interfered with the execution

of the behavior. Environmental failures are detected by the perception model. In virtual environments, we expect the environment to give direct feedback about error conditions, while in real world environments, the perception model will have to do additional work. Detection of environmental failures due to other agents is feasible in real-world environments, but other types of failures may not be easily detected.

When an environmental failure occurs, it can result in looping behaviors. A given behavior will trigger and execute, but, due to the failure, there will be no state change. This will leave the character in the same behavior. If a full animation plays this can look especially bad, while in the best case the character will become unresponsive and appear to be idle. This leads to an AI that appears not to respond correctly to the environment and its situation.

3.2.4.4 Behavior Interaction Failures

Behavior interaction failures are potentially the most difficult failures to detect. This type of failure occurs when a behavior can be executed correctly, does not generate an environmental error, but also does not achieve the expected result. Failures of this type are due to behaviors interacting in a way that causes one behavior to undo or cancel out the effects of another. Detecting behavior interaction failures requires specifying expected pre- and post-conditions for a given behavior. Preconditions may be the same as the prerequisites for the behavior, if the behavior has an effect on agent state. If the behavior affects environmental state, then the triggers for the layer should be used. Postconditions can be added as additional triggers to check for expected changes in either the agent or environmental state. Detection is complicated if the failure is not instantaneous or the effect of the behavior is intended

to be short-term. Some behaviors may already be written to have completion criteria (such as achieving a navigation goal), in which case these completion criteria can be used for the postconditions.

As with environmental failures, behavior interaction failures can lead to looping behaviors. In this case, each behavior is successful, but is immediately undone by the next. This can manifest as a character opening and closing a door repeatedly, or picking up an object and dropping it. This type of behavior appears entirely irrational, and can have a negative effect on suspension of disbelief.

3.2.4.5 Applying Failure Detection

Augmenting subsumption layers with the information required for failure detection is relatively easy. Prerequisites and expected results can be described in the same manner as the triggering conditions already used to decide when to activate a behavior. Tracking activation history adds a constant amount of additional state to each layer. The most invasive addition is detection and handling of environmental failures, but most systems will already watch for these error states.

Adding failure detection provides opportunities for major extensions to subsumption. Some of these can be applied to other reactive architectures, such as FSM-based controllers, but we focus on the application of failure detection to subsumption. A major application of failure detection that we have implemented is *reactive teaming*, which uses behavior insertion and removal.

3.2.5 Dynamic Priorities

Subsumption architecture controllers are typically defined as static structures. In a static controller, if a behavior should have different priorities in different situations,

it must be repeated in the specification with more complex triggers. We have created an alternative by allowing layers to have *dynamic priorities*.

The priority of a layer in subsumption is usually described as a static value. Instead of using a static value, we can instead describe the priority as a function. This function can map the current set of percepts to a priority value or in terms of other behaviors. Simple priority values indicating the index of the layer may be incorrect if reactive teaming is used to insert new behaviors at runtime, so describing the static behaviors directly above and below the dynamic location of the layer will be more robust.

Using dynamic priorities does increase the complexity of the controller. Because it is difficult to automatically adjust subsumption policies, the dynamic layer must have a subsumption policy defined for all layers which could possibly run as lower priority. Likewise, all layers that could possibly run as higher priority than the dynamic layer should also define a subsumption policy for interacting with it.

Dynamic priorities provide two advantages, though. First, while this extension potentially increases the number of relations between layers in the architecture, it simplifies the controller by reducing the number of layers. From a controller design standpoint, this can help reduce errors as it eliminates the need to duplicate particular layers in some cases. Second, this extension provides a natural way to enable some team coordination. If an auction-based method for teaming is used to distribute tasks to characters, each behavior capable of fulfilling team tasks can be created as a dynamic layer. When a task is assigned, the priority mapping function then raises the priority of the appropriate layer.

This provides a simple way to enable team coordination between characters with-

out deeper architectural changes. If more invasive changes are possible, other team coordination methods that avoid the expense of auction coordination can be used instead.

3.3 Reactive Teaming

Individual agents do not always provide rich enough behavior for a particular game or simulation. A given scenario may require multiple agents to work together to create a convincing environment, either through a strict command structure or a cooperative team. An extension to our architecture allows multi-agent coordination without significant additional complexity in the authoring process or heavy consumption of resources.

Agents in reactive teams coordinate with one another by exchanging behavior layers. These layers can be transferred in their entirety, or subdivided to reduce the burden of each agent and possibly improve the speed of the task. Reactive teaming can also make use of environmental information to provide convincing cooperation without explicit task division.

3.3.1 Behavior Transfer

The core technique of reactive teaming is agent-to-agent transfers. Typically agents in teams will be selected to perform a particular behavior based on their suitability. This requires evaluation of each agent with respect to each behavior. If all behaviors are assigned at once, the computational complexity is typically $O(mn)$ for n agents and m tasks. The number of messages required to communicate utility values is also typically the same. Online assignments, where one task is assigned at a time, will still require $O(n)$ complexity and communication to assign individual tasks.

Just as reactive control greatly reduces the complexity of individual character controllers, reactive teaming reduces the complexity of team coordination. Instead of assigning behaviors globally, all assignments are local. Furthermore, for a character to be assigned a task, it must request one. The core technique of reactive teaming is a one-to-one behavior transfer. Take two characters in a virtual environment, Alice and Bob. Alice has no current behaviors, or may have simple placeholder behaviors (we will refer to Alice as a *generic character*, and assume that all generic characters have a single default *wander* behavior). Bob has one or more defined behaviors (we will refer to Bob as a *fully-defined character*).

When Alice encounters Bob, she will request a behavior from Bob. Then Bob will choose a behavior to transfer to Alice, offer the transfer, and if Alice can execute the behavior, she will accept it. Otherwise, Alice will reject the behavior. No utility value is calculated, and no other characters are considered during the process. The result is that while it may take up to $O(n^2)$ character interactions to distribute behaviors in the worst case (n-1 generic characters, and each generic character queries all other generic characters before the fully-defined character, with no child behavior transfers), these interactions are amortized over multiple game ticks. This amortization is important as it prevents the task distribution process from dominating the limited CPU resources available during each frame of the game.

In heterogeneous environments, this method of behavior assignment may result in non-optimal assignments; related to this problem is that this technique does not allow behaviors to be *pushed* to other characters, as receiving characters must request behaviors. Behaviors may also be distributed unevenly, depending on the policy for

choosing which behavior to transfer. Additional refinements can address each of these problems in different ways. Behavior pushing may be important in situations where teams of characters are organized in hierarchies. These hierarchies can be created through high-level user logic to issue team commands through the environment. The distribution problem is solved through transfer policies.

3.3.2 Request Policies

Agents must decide when to request a behavior transfer. We have defined four approaches to deciding when to request a transfer: failure, cooperative, greedy, and command.

An agent using the *failure* policy will request a behavior transfer when one or more layers are failing to execute properly. This may occur in a number of different cases. First, a layer may no longer be able to activate correctly because one or more prerequisites are no longer available. An example of when this may occur is if a behavior requires a specific object (or object with specific affordance), and that object is no longer available. The subsumption controller can check behavior prerequisites to discover this without attempting to execute the behavior, so that behaviors which are failing to trigger at all can be detected. In other cases, a behavior may attempt to execute and fail; this can occur when an agent attempts to navigate to a part of the world which is no longer accessible. Another more rare condition that can occur is if too many agents are attempting to use a limited resource in the world, such as an object. An example is a scenario in which several agents are attempting to perform an object removal behavior. If the agents are grouped together and repeatedly attempt to pick up the same object, the behavior will fail. Repeated failures will result in

the behavior being chosen for replacement, and so the team can self-correct if too many agents have the same behavior in this case. Failure cases will not monitor the behavior which failed to check if it is available once more.

The *cooperative* policy will request a behavior transfer if it does not currently have a transferred behavior. If the transferred behavior fails, it will look for a new behavior transfer. It will not monitor behaviors which have been replaced due to failures.

The *greedy* policy is similar to the failure policy. It will request a behavior if one of its own is failing, but unlike the failure policy, it will monitor the failed behavior. In case of a prerequisite failure, it will restore the behavior if the prerequisites are restored. In other types of failures, it will periodically attempt to restore the failed behavior; if it succeeds, the replacement behavior will be removed.

A final policy is possible, which is the *command* policy. In this case, the agent is part of a command hierarchy, and will respond to requests from commanding agents. When the agent receives an order from a commanding agent, it will request a layer transfer from the commander. Note that the actual layer transfer is still initiated by the receiving (subordinate) agent, and the command-response layer could potentially be placed below other layers, allowing for insubordinate agents.

3.3.3 Transfer Policies

Once a request to transfer a behavior is received by an agent, it must decide which behavior to transfer. Before transferring a behavior, the agent must have at least one layer marked as transferable and which has not exceeded the maximum number of transfers. The transferable flag and the maximum number of transfers are defined when the agent is authored. We define four possible transfer policies: priority, failure,

distribution, and command.

The *priority* policy will always choose the highest priority transferable layer. A variation on this will always transfer the lowest priority transferable layer. The layer transferred may be one which was transferred to the agent from another. Behaviors which have exceeded their maximum number of transfers may not be transferred.

Failure policies will transfer the highest priority transferable layer which is currently failing. If no layers are failing, it will fall back on the priority transfer policy.

A *distribution* policy is one which aims for a target distribution of its layers. The distribution policy will transfer a layer which has not yet reached its target number of transfers, choosing layers which are far from their goal first with priority as a tiebreaker.

Finally, the *command* policy is the complement to the request policy of the same name. Command policies will transfer behaviors according to the mapping determined by a command behavior layer. Note that this effectively allows auction and planning teaming methods to be used as a layer of abstraction over reactive teaming– the alternate team coordination method can be implemented as a behavior.

3.3.4 Insertion Policies

While placement of received layers is defined primarily by the request policy, the subsumption policy must also be defined. Possible strategies to choose subsumption policies are *override all*, *override none*, *replacement*, or *modality*. The default policy is for a transferred layer to *override all*. The opposite of this policy is, naturally, *override none* which will only override conflicting output produced by lower layers. *Replacement* uses the subsumption policy of the layer being replaced, or falls back

on *override all* if the transfer is not a replacement. *Modality* policies override one or more of the effector modalities to suppress only specific channels of action.

3.3.5 Behavior Coordination

While layer transfer provides a simple level of team coordination, it generally does not create tight interactions between agents to execute a task. Additional coordination methods are needed to achieve this.

Environmental intelligence provides methods to coordinate behavior. Influence points can be used either per-agent or per-team. If per-team influence points are applied to the environment, then characters can use the world as a blackboard to share information.

Affordances are particularly valuable for teams of characters. Multi-character affordances provide a mechanism through which multiple characters can cooperate. Tasks may be actions that cannot be performed with one character, or actions that are improved through the use of multiple characters. As described in Section 3.1.2.3, multi-character affordances do not require additional specialized logic for each type of task on which they will coordinate. This provides significant savings in character complexity over hand-crafted behaviors for specific tasks.

Finally, some tasks are not easily shared with affordances or influences, and must be modified when transferred. Behavior division provides an additional solution. Some behaviors in the BEHAVEngine behavior library are written to include methods which can divide and merge behaviors. The simplest example of this is division of an explore behavior, which maintains a list of regions to visit. When the explore behavior is transferred to an additional agent, the current list of regions is broken

into two parts, so that two agents will work together to cover the area faster. The division task is most appropriate to spatial tasks, though it is possible to apply to others.

3.3.6 Self-Preservation Layers

One danger with reactive teaming is that certain layers which should be high priority will be subsumed inappropriately by team behaviors. In games, a major concern is self-preservation behaviors for characters. A character in a FPS action game should never have its behaviors to attack hostile characters subsumed, but if it is given a cooperative strategy, these behaviors may be subsumed by the transferred behavior.

To solve this problem, a set of fixed layers needs to be protected from team subsumption. Depending on implementation, this is done by marking a layer as the ceiling for team transfers, so that no behaviors will be inserted above it, or by maintaining part of the subsumption stack separately.

CHAPTER 4: IMPLEMENTATION

The techniques described in Chapter 3 are implemented in two different game testbeds. Our first testbed, DASSIEs, uses a 3D simulation and a high-resolution spatial environment. While working in a high-resolution environment such as DASSIEs demonstrates that our methods can be used in realistic game environments, most of the development time focuses on building the environment (including the creation of art assets). The second testbed is the Subsumption and Python-based Lightweight Artificial Intelligence Testbed (Splat). Splat focuses on core game and AI logic, and, unlike DASSIEs, has no graphical representation. Because there is no need to develop 3D art assets, manage 3D animation, or optimize the graphics pipeline, Splat makes it simpler to focus on the performance of the AI techniques.

4.1 DASSIEs

DASSIEs is composed of four major components. First is the FIrst and 3rd-person Realtime Simulation Testbed (FI3RST) to evaluate AI techniques, which incorporates navigation meshes and embedded intelligence from the ground up. FI3RST is a real-time 3D simulation that provides a high-resolution environment. The Common Games Understanding and Learning Toolkit (CGUL) includes various libraries and utilities used across DASSIEs, including the Static Spatial Perception Service (SSPS), which provides information access to navigation mesh information. Our AI engine

itself is the Behavior Emulating Hierarchically-based Agent Vending Engine (BEHAVEngine). BEHAVEngine implements hierarchical behavior-based subsumption, including reactive teaming extensions. More information about BehaviorShop, the fourth component of DASSIEs, can be found in Appendix 6.4. The major strength of DASSIEs is that it provides a testbed with all the constraints of real 3D games. Controlling characters in DASSIEs requires interacting with a 3D animation system, user interface, and dealing with heavily constrained resources.

4.1.1 FI3RST

FI3RST is our testbed for evaluating game AI techniques. The initial version of FI3RST was written in Python using the Panda3D game engine and used with the first BehaviorShop study [29]. We found several shortcomings in our initial implementation, especially due to the fact that navigation meshes were not fully integrated and the environment needed to be more data driven. FI3RST 2.0 resolves these issues.

Our new game environment is written in C++ using the Irrlicht game engine. Irrlicht provides basic support for loading 3D models, including a level file format and creation of a scene graph. In addition to using these basic facilities, we have added support for navigation meshes and integrated AI debugging visualizations.

Navigation meshes are used in FI3RST to help manage *dynamic objects*—meaning characters, objects, and vehicles. Characters include the player character and any AI characters controlled by BEHAVEngine. Objects are loaded by the game scenario separate from the level or created by characters. Vehicles are a special type of object which can be used by characters to move around the world. Each dynamic object type is maintained in a separate list and individual objects each have their own update

calls. At each game tick, every dynamic object is updated, and at this time, the game locates the object in the navigation mesh using the SSPS library. When world state is read by the AI controller, this information is used to help determine visibility and used to aid pathfinding.

A data-driven approach is taken to characters, objects, and vehicles in FI3RST. While actions that can be performed by characters are necessarily defined using code, the action definitions do not include the animations or objects to be used. Instead, when a character is loaded, an action-to-animation map is included in the character metadata file. This includes an action model which specifies which actions are available to the character. Likewise, the object meta-data file includes not just the location of the 3D model to be loaded, but also information about actions which can be performed with the object. This means that simple affordances are built directly into FI3RST.

The interface between FI3RST and BEHAVEngine uses shared memory. State for all dynamic objects is stored in shared memory, and the state data includes fields for the AI engine to make requests. The action model and object built-in affordances are not currently read from shared memory, and the navigation mesh is also loaded separately by both FI3RST and BEHAVEngine.

FI3RST also has an extensive logging system that can record the movement and actions of all AI characters into XML-based logs which can be easily consumed for analysis. Additional debugging facilities include an overlay of the navigation mesh and character waypoint indicators, as seen in Figure 4.1. One of the most important tools is the character controller overlay, which is shown with more detail in Figure 4.2. This

Figure 4.1: FI3RST contains multiple overlays to assist with AI debugging. Agent controllers are visualized as boxes, with color changes to indicate active layers. Flags in the world show agent waypoints. The navigation mesh is overlaid on the world terrain.

overlay shows the subsumption controller for the different characters including the activation and failure rates for each layers. When layers are active, the color changes, so that character behavior can be understood with a glance. Team behaviors are given identical colors so that it is easy to see which characters are running behaviors that have been transferred.

An additional design consideration taken into account with FI3RST was the need to easily create new scenarios, some of which may use different rule sets. FI3RST

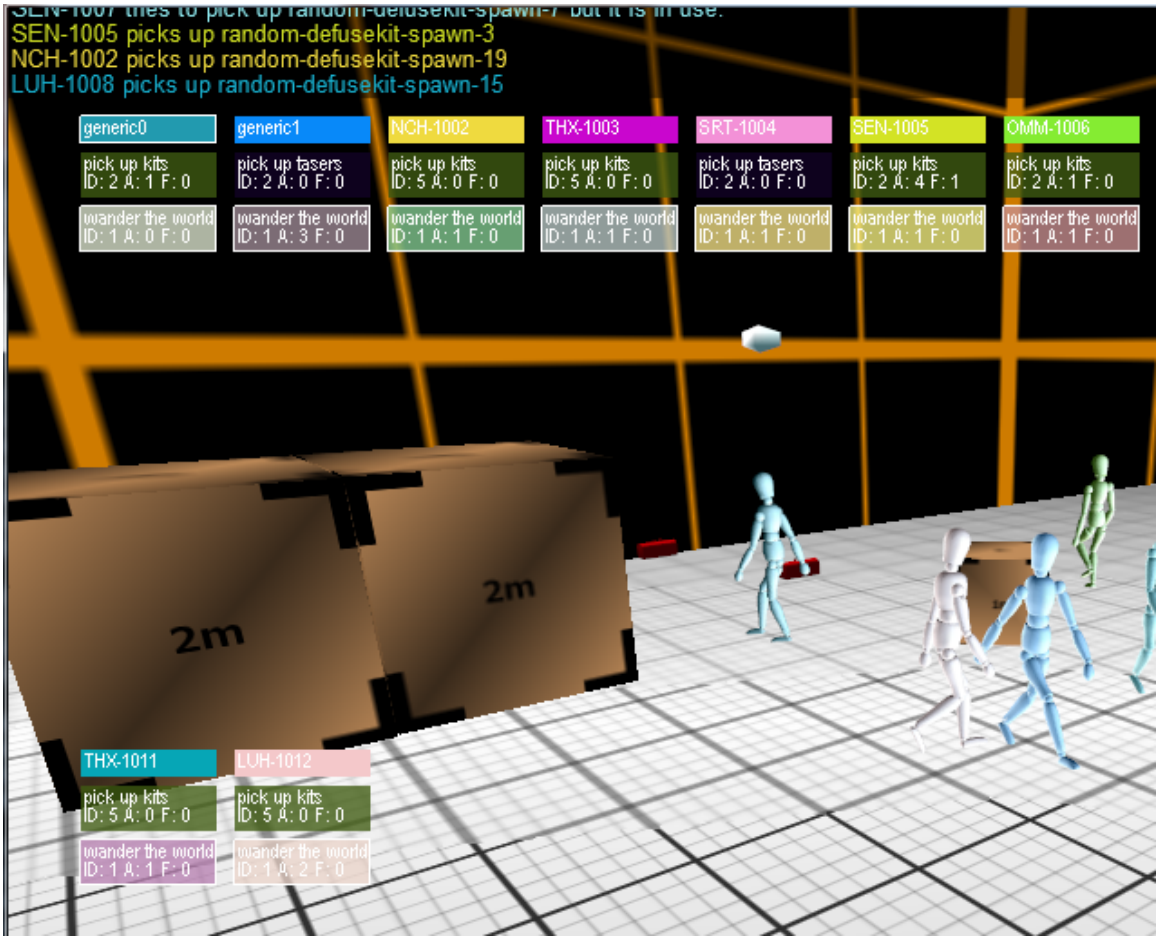


Figure 4.2: Detail view of the FI3RST console output and agent controller display.

consumes a scenario file which defines the level to use, the dynamic objects to load, any game events that should occur and the timing of those events, and the navigation mesh file to be used. This information is also loaded into shared memory by FI3RST at startup so that it is accessible to other game components.

Game assets used in FI3RST were created as part of the DASSIEs project. Character models are animated using motion capture animations from the CMU Motion Capture Library [10].

4.1.2 CGUL

CGUL is a collection of libraries to provide information to AI at runtime as well as preprocess environment information and analyze player data. These libraries include MemStore, which provides communication between game components, and SSPS, which provides an API for accessing navigation mesh data. Other services are provided as part of the AgentUtilities library which provides assorted functionality such as common types, a logging framework, and wrappers around the libxml2 library. Several test programs were created in CGUL to assist in debugging the other game components, and CGUL was used to experiment with an early implementation of contextual affordances.

4.1.2.1 Agent Utilities

The agent utilities library of CGUL provides various useful common data structures and methods. Data structures include a single-precision floating point Point class, a single-precision floating point Orientation class (using Euler angles), and structures for holding information about behaviors, objects, and the BEHAVEngine query system information. The object library used by FI3RST, BEHAVEngine, and BehaviorShop is part of this library, as well as a unified logging system for run-time information and data collection. Finally, a set of wrappers around the libxml2 C library are provided to simplify the process of loading the XML configuration and data files used by other packages.

4.1.2.2 MemStore

MemStore is a shared memory service, implemented as libMemStore and the MemStore server. The library provides a cross-platform interface to shared memory, providing a communication method for other applications in the DASSIEs project.

4.1.2.3 SSPS

The SSPS library provides access to the navigation mesh. The mesh itself is represented in two ways: as a collection of geometric entities and also as a directed graph. The geometric representation is useful for local region navigation and storage of compartmentalized information, while the graph includes A* planning functionality to allow global navigation.

When SSPS loads, it reads the navigation mesh from an XML file. It constructs a default graph to plan through which assumes a default sized agent bounding box. When an agent is instantiated, the SSPS service can generate a new graph based on the actual size of the agent bounding box.

The main SSPS query is *findPoint(location,hint)*. This query searches for *location* in the navigation mesh, returning the unique ID of the region in which it is contained, or `BAD_REGION` if it is not contained in any regions. The *hint* provides a starting point for the search; if provided, this region will be checked first, followed by all of its adjacent regions. The search continues, breadth-first, until the point is found or all regions have been checked.

A* path planning is provided by the *findPath(start, end, starthint, endhint)* query. This query returns a *stl::list* of region ids. If no path is found the returned list is

empty, otherwise there is a path through the navigation mesh from the starting point to the end point that passes through each of the regions in order.

Several other SSPS queries exist to support other needs. *generateRandomValidPoint(&region,ground)* will generate a random point, guaranteed to be in a negative space (navigable) region. The region reference is filled with the ID of the region in which the new point is contained. The ground parameter is a boolean value specifying whether the point should be on the ground; if it is false, the generated point will have a z (height) value between the region's z_{min} and z_{max} values. If a point is believed to be within a positive space (non-navigable) region, the *findPointInObject(location)* query will search all positive space regions for the given location.

4.1.3 BEHAVEngine

BEHAVEngine is implemented in C++, using several libraries from the CGUL toolkit. Multi-threading capabilities are provided by a thread pool implemented as a wrapper over the boost::thread library. In addition to the boost libraries, BEHAVEngine also makes use of the C++ standard template library and libxml2, an open source XML library which includes schema validation.

The major components of BEHAVEngine are the AgentCore, the SimInterface, the ThreadManager, and the Agent (see Figure 4.3). The AgentCore is responsible for loading all agent descriptions from XML, initializing the SimInterface, and initially creating AIControllers. The SimInterface connects to the simulation environment being used, and provides basic services for communicating with the simulation. A custom thread pool is implemented in the ThreadManager to provide multithreading capabilities for BEHAVEngine. For each Agent, there is an AgentController, an

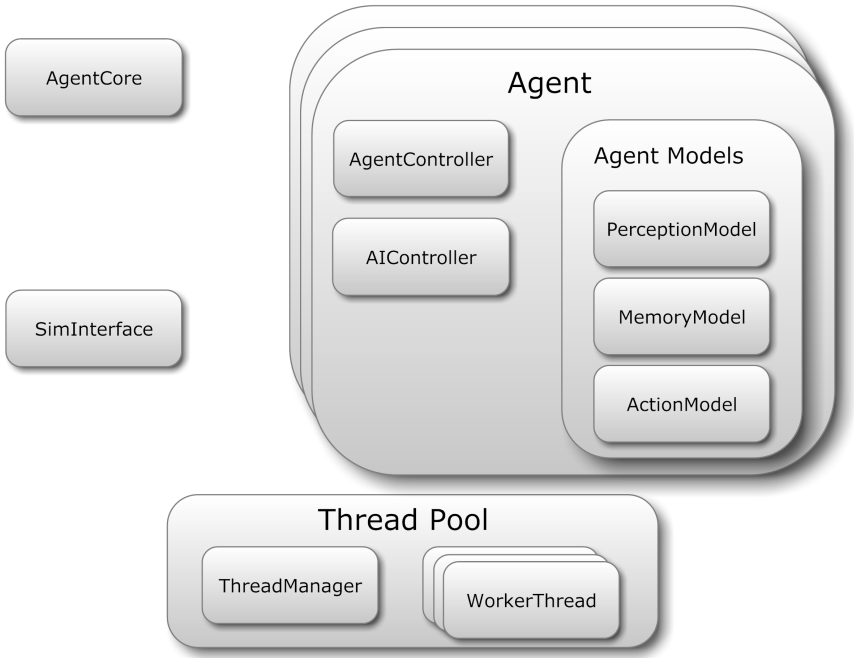


Figure 4.3: Major components of the BEHAVEngine AI engine. The AgentCore provides an execution loop and services to load individual agents. The SimInterface connects to the simulation environment. A worker thread pool is provided by the ThreadManager. Individual agents are composed of AgentControllers, which have an ActionModel, a PerceptionModel, a MemoryModel, and an AI controller.

AIController, and a set of models for processing agent inputs, outputs and memory. The AgentController schedules updates to the perceptual model and AIController, and ensures that commands are passed from the action model back to the SimInterface. Each agent is allocated a thread, which may be shared with other agents. Generally one thread per available CPU core is created, but this default can be overridden.

The main controller for BEHAVEngine is a separate entity from the simulation-specific behavior and trigger libraries, as seen in Figure 4.4. These libraries are statically linked and can be replaced if different capabilities are required. At this time, only the behaviors and triggers are provided as an external library, but the controller

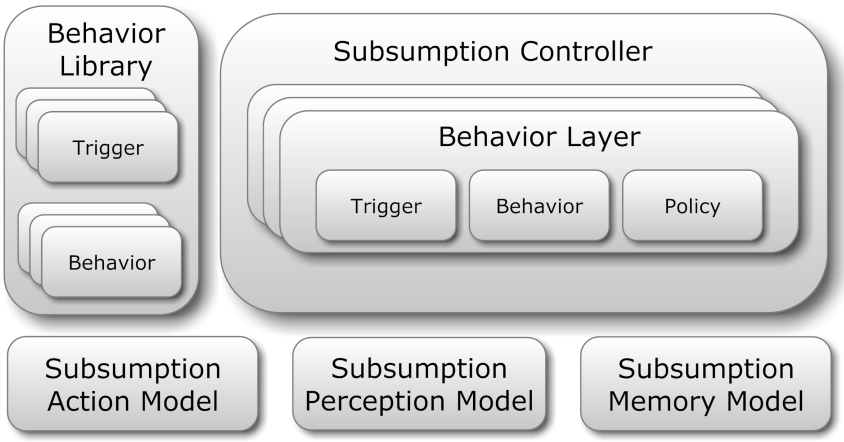


Figure 4.4: BEHAVEngine AI Controller. The AI Controller implements a specific AI architecture. The components of the AI Controller are designed as modules that can be loaded in from an external library, allowing different architectures to be implemented.

is written to allow future support for modules which implement agent architectures other than subsumption.

BEHAVEngine supports reactive teaming and complex layer types. Layers can be constructed as hierarchies of other layers or from sets of behaviors and triggers. Behaviors can be used singly or linked in a series so they run one after the other when the layer executes. Triggers can be composed with conjunctions and disjunctions of tests.

Characters are specified using XML files. Building new intelligent characters does not require writing any code, as they are loaded at run-time. BEHAVEngine has an initial configuration which is loaded from an XML file at startup, but the simulation environment can request the creation of additional characters after the initial startup phase.

BEHAVEngine can provide information about character controllers to the simula-

tion environment so that it may be used for debugging interfaces, such as the overlay in FI3RST. As with FI3RST, character debugging information can be captured to a XML-based log. Data captured includes information about behavior transfers that can be processed to show how behaviors spread through teams over time. Navigation mesh information is provided to BEHAVEngine by SSPS, while scenario information is provided by FI3RST.

Agent Building Interface Support

A major feature of BEHAVEngine is the capability to export information about the agent architecture and available domain-specific behaviors. This information can be imported by the BehaviorShop agent builder and combined with additional language information provided separately. Different domains can then be supported by BehaviorShop through the use of different behavior information files, and the language used in the interface can likewise be modified independently.

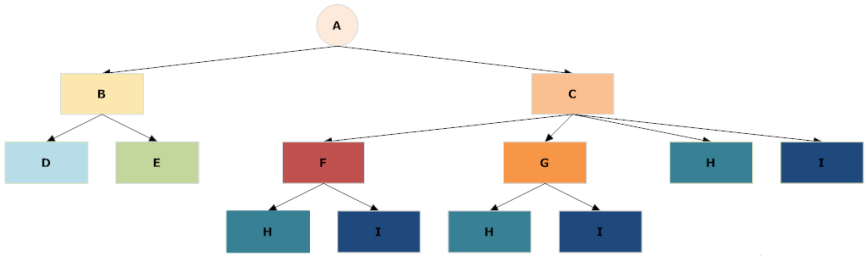


Figure 4.5: The static behavior information tree stored each valid configuration of a behavior or trigger as a separate node in the tree. In this case, B and C are behaviors, and their children are behavior options, such as patrol points for a patrol behavior or object types for a pick-up object behavior. Note that the static nature of the tree means that nodes H and I are repeated three times.

Information is exported as a non-deterministic hierarchical finite state machine– effectively a dynamic tree. Initially, behavior information was exported as a large

static tree. Each valid configuration of a behavior and its options was represented as a separate node in the tree, as seen in Figure 4.5. This meant that building a behavior was as simple as traversing the tree to a node marked terminal. Unfortunately, the tree itself became unmanageably large.

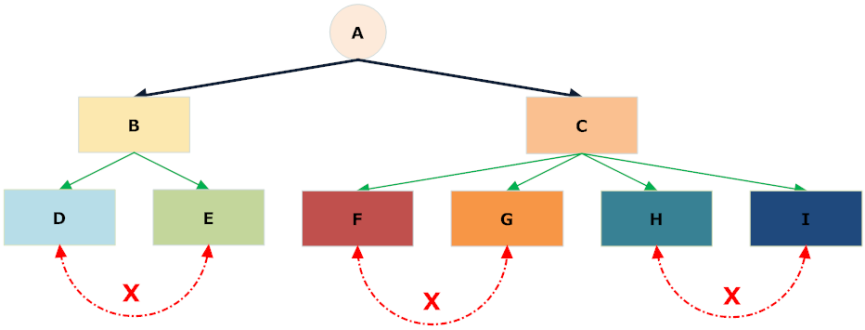


Figure 4.6: The dynamic behavior information tree stores each option for a behavior exactly once, and includes information about options that are mutually exclusive.

The dynamic version of this tree provides a similar structure, as seen in Figure 4.6. This tree is built dynamically, and stores each behavior option exactly once. The data structure also stores information about options that are incompatible. This enables the tree to be expanded as required when queries are made for behavioral information.

This behavioral information can then be combined with an additional source of language information that specifies how individual option types should be specified in natural language. An example of this language information is shown in Figure 4.7. Distance is one of the basic option types that can be used with behaviors and triggers (useful in, for example, the *see object* trigger to specify how close the character should be to the object for the trigger to activate), and the language source provides a more understandable representation for the option values than ungrounded numeric values. This representation can be used to build user interfaces, such as BehaviorShop, that

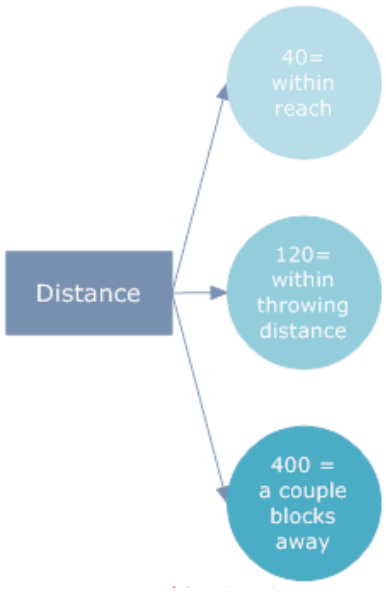


Figure 4.7: Language filters are loaded separately from behavioral information to provide natural language translations of the behaviors and triggers that are built into BEHAVEngine. The resulting translations can be used in character-building interfaces to provide more intuitive behavioral components.

provide intuitive tools to build character intelligence.

4.2 Splat

Splat (Subsumption and Python-based Lightweight Artificial Intelligence Testbed) provides a simulation environment that can focus on evaluating problems in artificial intelligence. While testing in full 3D environments is important, the process of creating art assets, debugging animation systems, and handling other related issues of game development can interfere with developing new AI techniques. Splat focuses on only the core game and AI logic, making it ideal for testing many of the techniques developed as part of this work. Additional information about using Splat can be found in the Appendix.

The major parts of Splat are the simulator, the extended affordances, the character

controllers, and the team coordination methods.

4.2.1 Splat Simulator

Environments in Splat are composed of multiple rooms linked together with doors and windows. These rooms can either be low resolution or high resolution. High resolution Splat environments use the room layout as a navigation mesh, and characters use local navigation to move inside the rooms (as in FI3RST). Low resolution Splat environments treat rooms more like the graph representation of a navigation mesh– characters move from room to room, and may interact inside rooms, but do not navigate locally inside rooms.

The Splat simulator is contained in the World object. The World contains a list of rooms as well as dynamic object and agent lists. All actions are executed through the World, and it provides all percepts to the Splat characters. The World is responsible for initial setup of the environment and tracking the current time step of the simulation.

Splat simulations vary widely depending on whether the simulation is a single-agent, multiple-agent, or team scenario, whether high resolution or low resolution rooms are being used, and the goals of the particular simulation. For this reason, individual simulators are built as python scripts that specify the layout of the environment, the objects in the environment, and the simulation rules.

Several different objects are commonly used in Splat simulations, and it is simple to create new objects. Our scenarios use doors, windows, tables, balls, boxes, and treasures. Doors can be opened, locked, unlocked, and closed. Windows are like doors, but they can also be broken with a ball. Both windows and doors can be

used to move between rooms. Tables can be flipped over and used for cover, but can also be destroyed using a ball. Characters who take cover behind a table cannot be hit with the ball. Balls can be used to break windows and tables or knock out other characters by *bonking* them. When a ball is used to bonk another character, it is removed from the inventory and lands on the ground by the target. Objects can be stored in boxes; when a box is opened and searched, its contents are revealed. Finally, the treasure object is a completely useless item that a character must have in its inventory to complete most scenarios. Rooms may be marked as *sanctuaries*. In sanctuaries, characters are not allowed to attack one another.

Other objects are also available in RPG/Puzzle scenarios. These objects are used to create traps and puzzles for characters to solve and include levers, pit traps, and keys. RPG/Puzzle scenarios also have angry badgers which will attack characters on sight, but go back into hiding when no characters are nearby. Badgers will not leave the room they live in.

4.2.2 Affordances

Affordances are implemented in Splat as Python objects. An affordance stores the object it is attached to, the action it represents, and other information about the action. When a character uses an affordance, it first reserves the affordance. In the case of multi-character affordances, the character may have to reserve and then wait until the rest of the team completes its decision cycle, so the reservation system is important to prevent multiple characters from using the same affordance at the same time.

Five types of affordances are defined. Basic affordances are the simplest variety,

and just specify that an action can be taken involving a given object. Contextual affordances report a set of appropriate actions given the context; each action or set of actions in a contextual affordance is paired with a set of triggering *conditional* functions to decide whether they should be reported. Multi-character affordances provide multiple affordance slots which may be reserved by different characters on the same team to enable multi-agent coordination.

Outcome affordances are like simple affordances, but they also provide information about the expected result of the action. Probabilistic affordances are outcome affordances that store a character's interaction history and provide an estimate of the likelihood of a given outcome. If a probabilistic affordance is linked to a single object, the outcome is uncertain, and the simulator must choose one of the possible results. If the affordance is linked to other affordances and objects, then it is used to represent a lack of character knowledge.

4.2.3 Splat Agents

Splat implements multiple types of agent controllers. The simplest Splat agents are pure random; these agents randomly choose affordances that are present in the world and attempt to execute them. Hierarchical behavior-based subsumption agents are implemented with the dynamic priority and reactive teaming extensions. Finally, behavior tree agents can also be built. Rather than using our own behavior tree implementation, Splat uses the Owyl behavior tree implementation, which provides a python API for creating and executing basic behavior trees (though certain features such as impulses are not included) [21]. Agents are configured to use or ignore context when accessing affordances. Contextual affordances check whether an agent is

configured to use context when actions are returned.

Agents have a set of possible actions, initial states, and inventory slots. To execute the agent decision process, *doSomething*() is called on the agent, which will then perform the necessary operations to choose an appropriate action. The action is then executed through the World object. Actions available to the agents include *open*, *close*, *break*, *exit*, *bonk*, *pick-up*, *search*, *hide*, *push*, and *lift*.

For subsumption and behavior tree-based agents, contextual affordance *conditionals* can be reused to filter perceptual information. These are augmented by a library of *deciders* that contain simple behavior logic for executing actions. Both types of agents use the same *conditionals* and *deciders*, though they are composed into different structures to build decision logic. Conditionals and Deciders can operate using either standard affordances or outcome-based affordances and agents may use a combination of different affordance types for making decisions. Behavior-based characters are expected to use standard (including contextual and multi-character) affordances, while goal-based characters use outcome-based affordances.

4.2.4 Auction-based Multi-Agent Coordination

Reactive teaming is implemented for use by subsumption agents, and multi-character affordances are available to all agents. A third teaming mechanism is available in Splat: auctions. The Auctioneer object manages auctions by offering team tasks to all characters. Each character is allowed to bid once on each task. The Auctioneer then scores each character's ability to perform the task, and allocates the tasks in a manner that maximizes the number of team tasks that are filled.

Subsumption and behavior tree characters can be mixed on a single team. For

subsumption characters, task assignments are used with dynamic priorities to adjust the priority of a layer designed for the given task. For behavior tree characters, since impulses are not available, task priorities are set by their location in the tree structure and the presence of the task is used as a test to activate the branch for each task.

CHAPTER 5: EXPERIMENTS

Evaluating artificial intelligence methods for games is difficult, and there is no easily agreed-upon metric for game AI even for particular genres of game. We discussed the problem of evaluation with industry game AI experts, and there was no consensus on generalized evaluations. Game AI, as implemented and designed in industry, focuses on tailoring methods to very specific games. The measure of success is based on the success of entire games, and proving a technique is useful in multiple genres or types of game may require several big hits.

Creating several games and testing them on the market is infeasible for academic research, so we chose to compare our methods to techniques that we already know are successful from industry. In particular, for our extensions to subsumption, we compare against the industry standard of behavior trees. If our methods can generate equivalent or similar behavior, we can expect them to perform favorably in many different game environments, as behavior trees do. For environmental intelligence, we look at reasonableness of behavior. The goal is to take simple controllers which may not perform reasonably—they generate behavior that does not make sense in context—and modify the environment so they actually do perform reasonably. We can measure this by considering the success rate on tasks that are representative of common game genres.

Finally, for all methods, we look at the complexity of applying our new techniques. The techniques should simplify game AI in two key ways: first, the computational complexity should be low, reducing the resource cost of generating characters behavior at run time. Second, the characters themselves should be simpler, reducing the amount of effort it takes to build new characters and debug them.

5.1 Environmental Intelligence

5.1.1 Influence Points

To evaluate influence points, we compare the number of calculations and memory required for equivalently sized regions of influence using both influence points and influence maps. We evaluated both methods with real maps, placing a series of random influences into the world. Figure 5.1 shows the four different maps and their decompositions.

5.1.1.1 Maps and Region Decompositions

We used four different test maps. For the influence maps, the worlds were 500x500 occupancy grids. The influence points used a world decomposition, with an average of 39 regions per map, calculated on a 5000x5000 resolution map of the environment (see Table 5.1 for the number of regions per map). Influence strength fall-off was $s_i - (D_i * 5)$, where s_i is the strength of influence source i and D_i is the distance from the source. The strength of the influence was 254, a value found to produce reasonably-sized areas of influence in the map (see Figure 5.2 for an example of the influence area this generated). Influence was spread using a grassfire distance transform, which is $O(n)$ in the size of the grid [9]. Grassfire (or prairie fire) is a simple two pass algorithm which is commonly used for calculating Voronoi diagrams

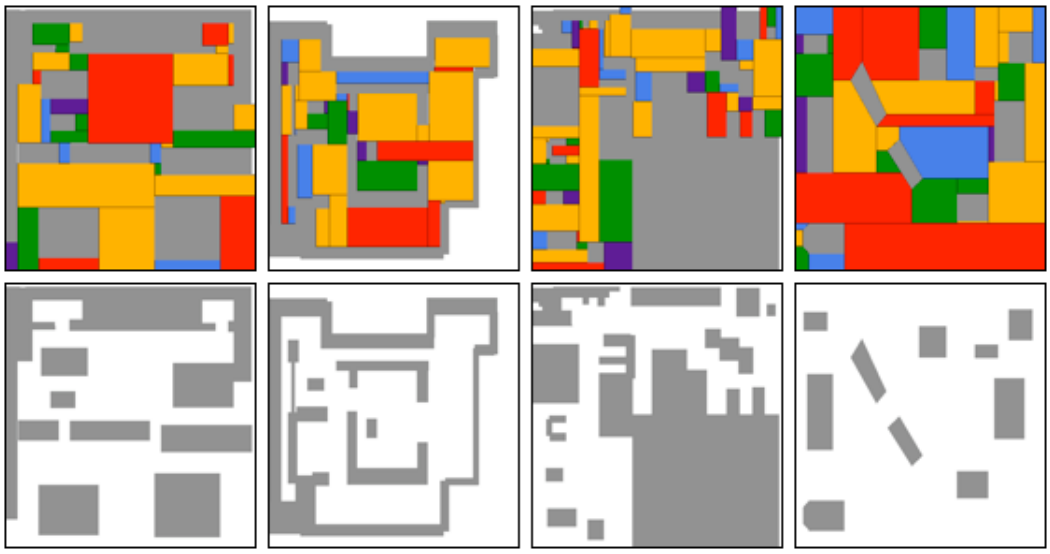


Figure 5.1: The four test maps for influence points. Obstacles are gray. The top image in each pair shows the map without decomposition, while the bottom image shows the navigation mesh decomposition (using Adaptive Space Filling Volumes). From the left, the maps are Docks, Precinct, Rommel, and Twin Lakes. These maps are from the Urban Terror Quake 3 total conversion mod.

or performing erosions and dilations on images.

For influence points, shortest distances between regions were pre-calculated for use during the insert operation; the initial strength and fall-off were corrected for the different world size.

Table 5.1: Number of regions in each decomposition. Based on a 5000x5000 cell map.

Map	Number Regions
Docks	31
Precinct	34
Rommel	55
Twin Lakes	36

5.1.1.2 Computational Costs

Lookups for influence points are more costly than those on an influence map, but the greatly reduced cost of the lazy update means that the overall computational cost

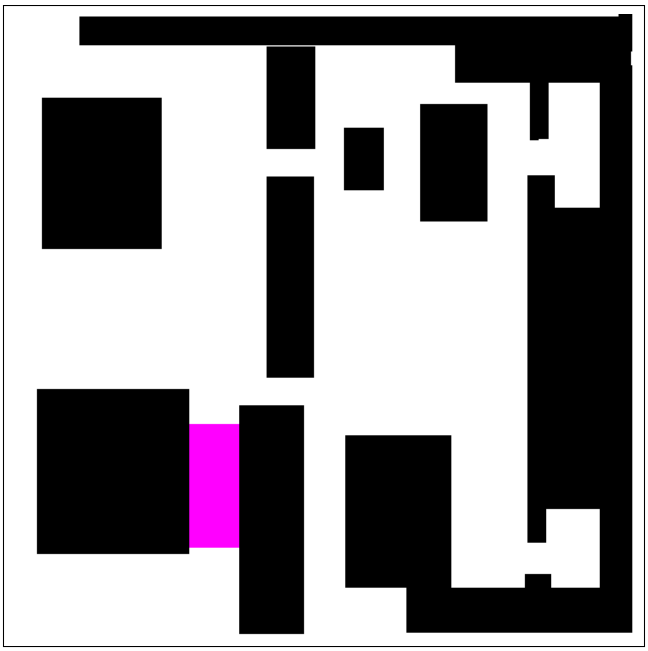


Figure 5.2: Example influence map in the Docks world. The shaded area is the area covered by the influence. Though not shown here, the strength of the influence decreases away from the center of the area

of maintaining and using the influence points is far less. Since the cost of an update and an insert are the same for influence maps, we will just look at the number of cells or regions which must be touched in an insert

As can be seen in Table 5.2, the number of grid cells touched during an insert in a moderately sized world is very large; even if the resolution of the grid is reduced greatly relative to the resolution of the map world, this number will still be much

Table 5.2: Number of updates per influence, average over ten random points. For influence map, this is the number of grid cells touched in an update or insertion. For influence points, this is the number of navigation mesh regions touched in an insertion.

Map	Map Cells	Mesh Regions
Docks	5206	1.1
Precinct	4690	1.1
Rommel	3945	1.2
Twin Lakes	6840	1.6

higher for influence maps than for navigation-mesh based influence points. The number of regions touched during influence point insertions, is actually very low: influence is mostly contained within a single region. If the influence drop-off is decreased, or strength of the influence greatly increased, the number of regions which must be updated is still quite small. A poor decomposition will increase the number of updates, but even a poor decomposition will generally be better than a purely grid based representation.

It is important to note that the actual cost of an insertion depends primarily on the quality of the decomposition (equilateral regions will decrease the cost compared to non-equilateral regions of equivalent area), the fall-off function, and the strength of the influence.

5.1.1.3 Memory Costs

Memory cost savings are even more dramatic than the reduction in computational costs, because full map grids must be stored for each influence type. In the worst case for influence points, each region must store a child or prime of every influence point, but because there are far fewer regions than there would be map cells, this is still a great reduction. The 500x500 cell maps use in our evaluation have an average of only 39 regions, compared to 250,000 map cells. Even reducing the resolution of the map to 50x50 results in almost an order of magnitude more map cells than regions, while providing a very coarse approximation of the data provided by influence points.

Storing the pre-calculated distances between every region costs $O(n^2)$ floating point values, where n is the number of regions. Given the number of regions in our decompositions, this comes out to approximately 1400 values. This is still far better than

the 250,000 values stored in each influence map for a 500x500 map (and these distance values only need to be stored once).

5.1.2 Contextual Affordances

We created an evaluation scenario using a small world where individual or small teams of characters compete. In this scenario, the characters have two goals: find the treasure and knock out its opponents. The world has five areas, as seen in the overhead view in Figure 5.3: Outside the walls, the guard post, the courtyard, the treasure room, and the sanctuary. There is a window from outside the walls to the guard post that can be broken, a window from the sanctuary to the courtyard that can be opened from inside the sanctuary (or broken from the courtyard), a door from the guard post to the courtyard, a door from the courtyard to the treasure room, and a door from the treasure room to the sanctuary. Doors and windows can be opened and closed if unlocked. The sanctuary is treated as a special room where characters are not allowed to attack one another.

There are six types of objects available in the world, each with one to three available actions (seen in Table 5.3). Doors and windows account for two types. There are also balls, tables, treasure chests, and treasure. There is a ball in the guard post and another in the sanctuary. The treasure room has a table and a treasure chest. Balls can be picked up and used to break windows or bonk other characters. When the ball is used, it is removed from the character's inventory. Tables can be pushed over and lifted back up; characters can hide behind a table that is pushed over. The treasure chest can be opened and searched. When the treasure chest is searched, the treasure is revealed. The treasure can be picked up once it has been revealed through

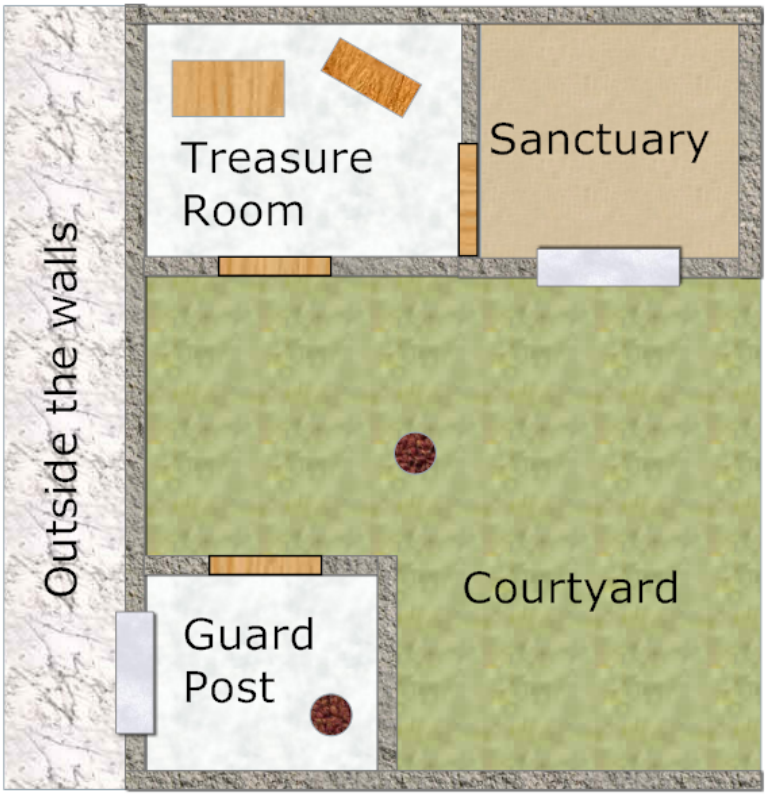


Figure 5.3: Overhead view of the scenario world

searching.

Each of the object actions is restricted by contextual affordances. The *break* action, for example, is only allowed if a window is closed and locked. Other restrictions are related to characters and their states. The table will only report *push* as a possibility if the character has recently been bonked with a ball, and the treasure chest will not provide *open* as an affordance if there are unfriendly characters in the room. Level design affects the *bonk* affordance on the balls, as it will not be available when the character is in the sanctuary.

During each turn, every character may perform one of the actions in Table 5.4. Character perception is limited to the current room. Once a character has been

Table 5.3: Object Affordances

Object	Action Type	Action Slot
Ball	bonk	object
	break	object
	pick-up	object
Chest	open	target
	search	target
Door	exit	object
	open	target
Table	hide	object
	lift	target
	push	target
Treasure	pick-up	target
Window	break	target
	exit	object
	open	target

bonked three times with a ball, it is knocked out. Knocked-out characters drop their inventory, so if a character carrying the treasure is knocked out, the treasure can be retrieved by its opponent.

We created four characters. Each character made its action decisions differently, though actions that would undo the last action taken were prevented unless the character had no other options (e.g., walking from the guard post to the courtyard and then back to the guard post, or pushing the table and then lifting it). The simplest character was a *pure random* character that was aware of basic affordances, meaning that when it queried an object, it received the full, unrestricted action space of the object. To choose an action, it built a list of the available affordances from the objects in the room without taking context into account. This included actions that were not possible or unproductive. An action was chosen randomly from this list, and the character attempted to execute it. The second character was a *contextual random*

Table 5.4: Character Action Space

Action	Description
bonk	Throws the ball at another character. Characters are knocked out after three bonks.
break	Throws the ball at a window, breaking it.
exit	Moves to another room.
hide	Hides behind an object.
lift	Lifts up a pushed over object, revealing any hidden characters.
open	Opens an unlocked object.
pick-up	Picks up an object.
push	Pushes an object over so it can be used for cover.
search	Searches an open treasure chest.

character. This character was identical to the purely random character except that it would retrieve the list of contextual affordances. The contextual affordances on each object used up to three condition tests combined as conjunctions or disjunctions.

The third character was a hand-crafted behavior-based subsumption character with nine layers. Behavior-based subsumption is a form of reactive control that uses prioritized layers of behaviors activated by triggering conditions [48]. The controller structure is seen in Table 5(a). Higher valued layers were considered more important, so if the character was near an opponent, had a ball, and cover was available, it would prefer to bonk the opponent than take cover behind the table. This character would perform all actions except for lifting the table. The conditions used for building this character were a superset of the conditions used for creating the contextual affordances for the second character.

The fourth character was a hand-crafted subsumption-based character that used context, as seen in Table 5(b). For this controller, we only needed to specify affor-

Figure 5.4: Characters facing off in the game. Bob takes cover behind the overturned table while Alice approaches the treasure room.

dances to look for and the priority of these affordances. The layers triggered based on affordances, and in all cases the paired behavior did nothing more than pass the affordances through to the action model. The hand-crafted character without context had to trigger based on seen objects, then apply behavioral logic to execute the action specified by the behavior. With context, this was simplified as separate behaviors did not need to be created for this character. This also reduced the difficulty of testing. Only the contextual affordances needed to be tested, and then the behavior of the character checked to make sure the affordance prioritization was effective.

We chose not to compare these characters with more deliberative planning systems due to the nature of the scenario. Since the environment is unknown and dynamic, plans generated by a deliberative controller would be invalidated after almost every move. In addition, this would require adding substantially more meta-data to objects

Table 5.5: The hand-crafted and contextual crafted controllers differ in that the contextual crafted controller effectively only provides a prioritization for affordances, and so does not require detailed trigger/behavior pairs. This makes the controller substantially simpler.

(a) No context

Priority	Action
8	Bonk opponent
7	Take cover if threatened
6	Push over table
5	Search treasure chest
4	Pick up ball or treasure
3	Open treasure chest
2	Open door
1	Break window
0	Move if possible

(b) Using context

Priority	Action
7	Bonk
6	Hide
5	Push
4	Search
3	Pick up
2	Open
1	Break
0	Move

and actions. Short-horizon planning, such as that used in F.E.A.R, would accommodate the dynamic nature of the environment, but at the cost of additional complexity in building the necessary meta-data [61].

Characters were evaluated by pitting each type of character against each other type in the game world, seen in Figure 5.4. This included trials between characters of the same type. We ran 100 trials of 1,000 games each, where each game ran for a limited number of turns (allowing for stalemates). Two possible start positions were used; each character started in each position for an equal number of trials. We recorded how many times each character achieved a complete win (by both knocking out its opponent and finding the treasure) as well as the number of times only partial wins were achieved. Other information recorded included how many turns it took to achieve a complete win and the number of times each character failed to perform an action correctly.

5.1.2.1 Comparative Character Performance

One important measure for our technique is the win rate of the characters. Contextual affordances should provide better performance than purely random characters using only basic affordances. We measured the complete win rate and partial win rates of each type of character, as seen in Table 5.6 and Figure 5.5. Results shown are the average number of times a character would win over 1,000 games, averaged over all 100 trials.

Table 5.6: Character Performance (Complete Wins), per 1,000 games. Higher values are better. * indicates that performance of crafted and crafted + contextual characters was statistically different with $p < 0.05$.

Character	vs. Random		vs. Contextual		vs. Crafted		vs. Crafted+Con	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Random	496.5	15.7	139.5	11.0	111.0	10.1	33.7	5.7
Contextual	847.0	11.8	484.1	15.5	94.4	9.5	41.4	6.1
Crafted	885.1*	10.2	855.9*	10.1	500.0	0.0	500	0.0
Crafted+Con	965.0*	5.8	955.6*	6.3	500.0	0.0	500	0.2

Table 5.7: Character Performance (Partial Wins)

	vs. Contextual		vs. Crafted	
Character	Treasure	KO	Treasure	KO
Contextual	17.0	17.7	1.23	48.5
Crafted	48.5	1.2	0.0	0.0

The contextual affordance-based character greatly outperformed the random character, as did the hand-crafted character. The hand-crafted subsumption character won against the contextual characters almost as often as it beat the random character, but the contextual character performed slightly worse against the crafted character. The hand-crafted characters were an even match against one another, but the hand-crafted character with contextual affordances performed slightly (but significantly)

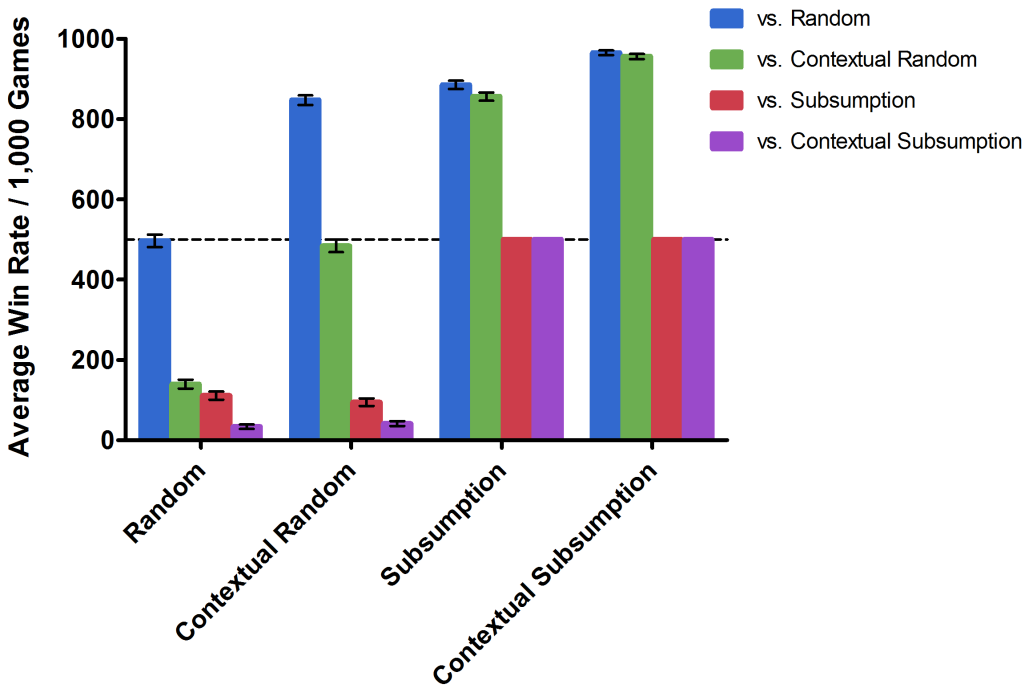


Figure 5.5: Graphical comparison of average win rate for random, contextual random, and hand-crafted characters. Higher values are better.

better against the random and contextual random characters. This is likely due to minor errors in the behavioral logic of the hand-crafted character without context.

Looking at the complete win rate hides one interesting aspect of competition between the contextual and crafted characters. Characters could get locked into a stalemate, where only one win condition could be achieved before the time limit was reached. This could happen when one character never encountered the other to knock it out, or one character never managed to retrieve the treasure. Partial wins were rare with both random characters and contextual crafted characters, but occurred frequently with the contextual characters. Contextual characters tended to knock out the crafted characters, but had a harder time finding the treasure, as seen in Table 5.7. Crafted characters could find the treasure, but had difficulty knocking out

contextual characters.

Table 5.8: Average number of turns to win (lower is better)

Character	vs. Random		vs. Contextual		vs. Crafted		vs. Crafted+Con	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Random	125.3	3.3	112.6	7.6	130.6	8.2	128.8	13.5
Contextual	33.9	0.6	31.4	0.6	32.1	1.6	33.4	2.3
Crafted	16.3	0.2	14.3	0.1	12.9	0.1	12.9	0.1
Crafted+Con	16.5	0.2	15.4	0.2	14.4	0.2	14.4	0.2

The length of time it took characters to win varied greatly depending on the character type, as seen in Table 5.8 and Figure 5.6. Since the random character used very little information to limit its choices, it took the longest to achieve its goals. The contextual character was able to complete the tasks in a much shorter period of time, but still much worse than the hand crafted character.

The character performance comparison shows that while the contextual character did not outperform the random character in complete wins against the crafted character, it did manage more partial wins, and completed the task in less time. The contextual character also outperformed the random character in direct matches. These results suggest that while contextual characters are not a replacement for carefully crafted characters, they are potentially useful for minor characters. Contextual characters always perform actions that are appropriate and can succeed in the game world, but have a far lower authorial burden. Combining hand-crafted subsumption with contextual affordances creates the best characters while still reducing the authorial burden.

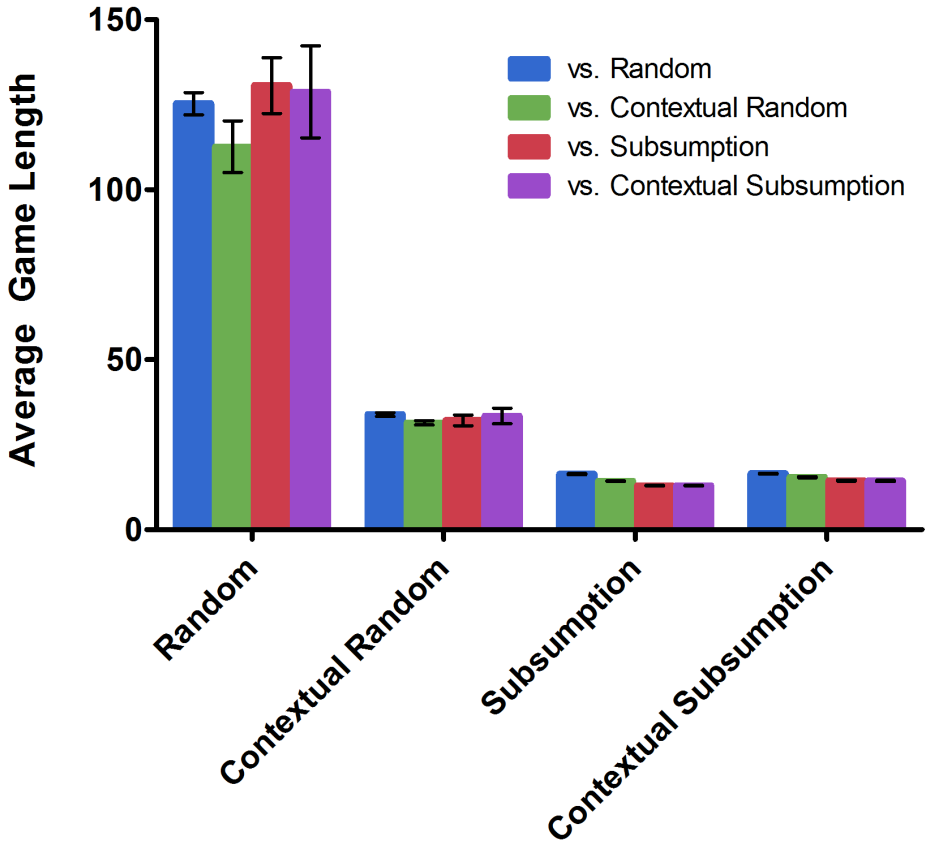


Figure 5.6: Comparison of average game length for random, contextual random, and hand-crafted characters.

5.1.2.2 Complexity and Reuse

Another important metric for consideration of contextual affordances is the complexity of building the hand-crafted character next to the complexity of building objects with contextual affordances.

The hand-crafted character is composed of nine behavior layers. The behavior layers have an average of 3.1 conditional tests per layer, and the most complex layers have 5 tests. The objects required a total of 15 condition/action pairs to be created, with an average of 1.5 conditional tests per affordance. While more condition/action pairs must be created, the complexity of the triggering conditions is substantially simpler. In addition, character controllers can be difficult to reuse in new environments, as behaviors are frequently tightly coupled. Objects with contextual affordances can be easily reused.

This complexity translates to real effort on the part of the AI programmer. Contextual affordances can potentially save a great deal of effort in development and debugging for the programmer. As seen in Figure 5.7 and Table 5.9, even using contextual affordances with hand-crafted agents reduces the number of lines of code required to implement characters, as it greatly reduces the number of behaviors that need to be built per-game. In addition, a small savings in single use character code can be achieved by using contextual affordances with crafted characters.

This saving is also important in terms of debug time, as seen in Figure 5.8 and Table 5.10. Using contextual affordances reduces the behavior combinations that need to be tested, enabling faster debugging of individual characters. In addition,

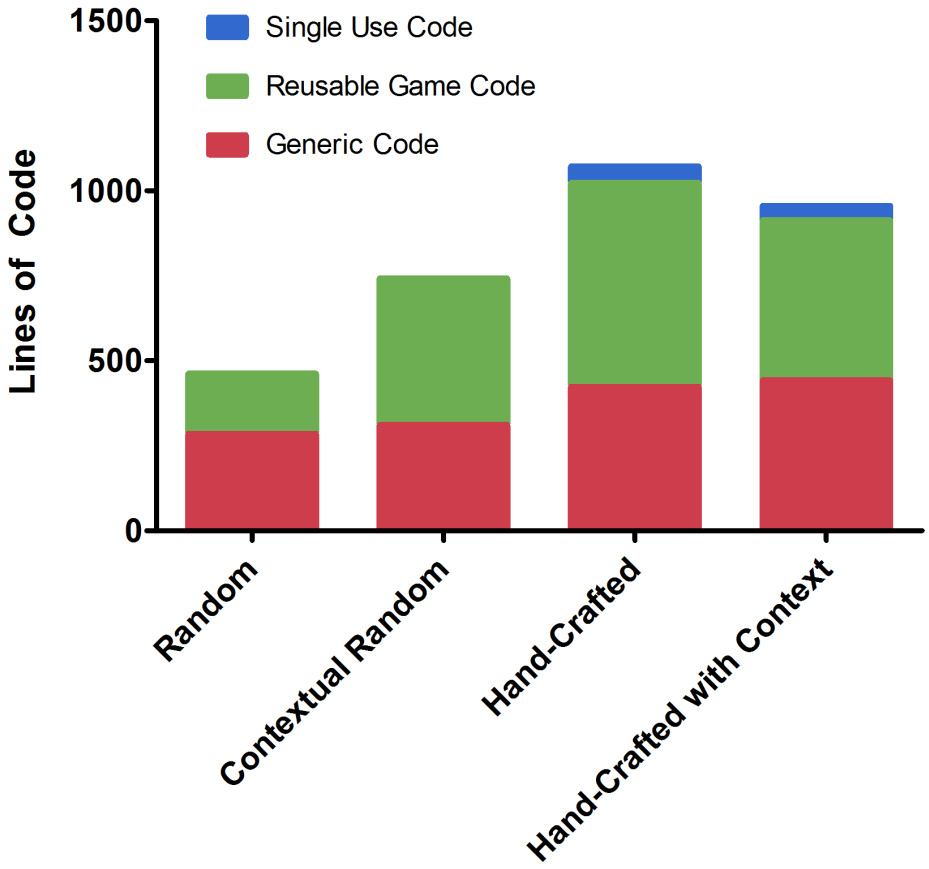


Figure 5.7: Number lines of code needed for each of the four agent types, divided by one-time (framework) code, per-game code, and per-character code. Generic code is reusable many times for different games, game code can be reused multiple times within a game, and single use code is specific to the character.

Table 5.9: Number lines of code needed for each of the four agent types, divided by one-time (framework) code, per-game code, and per-character code.

Type	Framework	Game	Character
Random	284	177	0
Contextual Random	310	431	0
Hand-Crafted	422	600	48
Hand-Crafted with Context	441	471	43

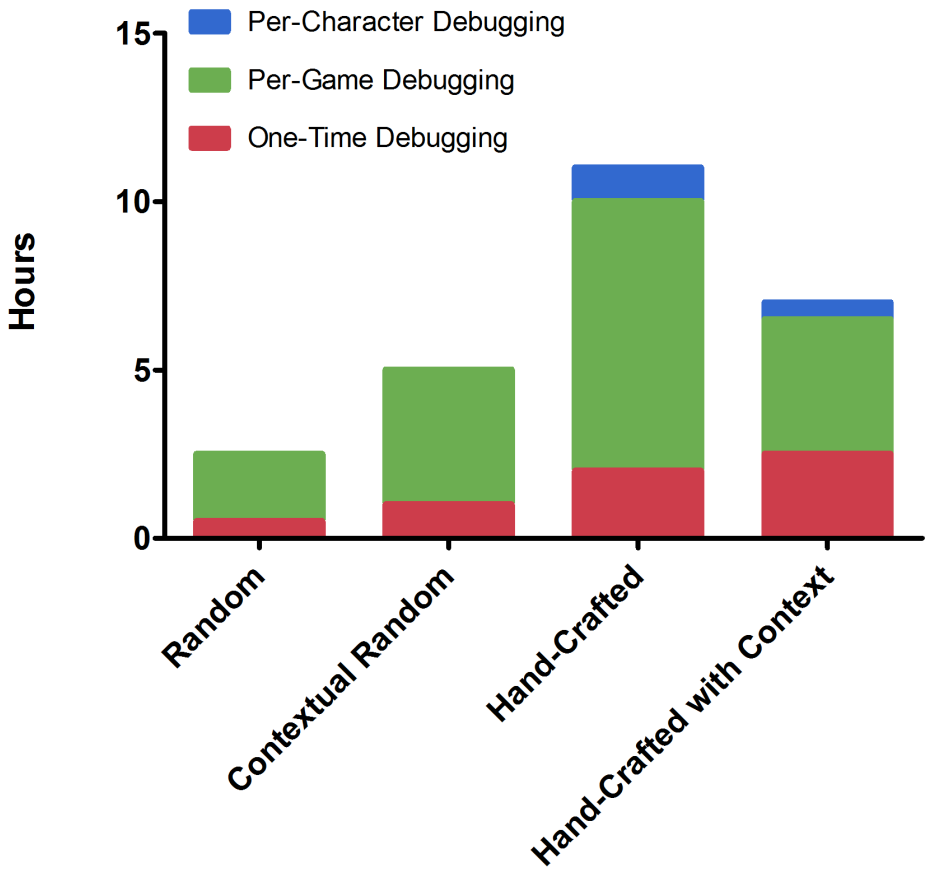


Figure 5.8: Approximate debugging time required for building characters, based on implementation within Splat.

the contextual smart objects created in the process can be used many times in a single game, providing a further gain. Especially important is the reduced need for behaviors, as the most time consuming element of AI debugging is frequently testing and tweaking character behaviors.

Another important advantage to using contextual affordances is the additional control they may offer to level designers. When building games, levels must be carefully designed to provide correct pacing and an immersive experience. Contextual affordances allow not only improved reusability, but enable designers to more closely tailor

Table 5.10: Approximate debugging time required for building characters, based on implementation within Splat.

Type	Framework	Game	Character
Random	0.5	2	0
Contextual Random	1.0	4	0
Hand-Crafted	2.0	8	1.0
Hand-Crafted with Context	2.5	4	0.5

the experience by changing the actions available for specific situations.

5.1.3 Multi-Character Affordances

Multi-character affordances are best used in puzzle environments with multiple characters on a team. To test the efficacy of multi-character affordances as a method for enabling multi-agent coordination, we built an environment with a puzzle that required multiple characters. This type of puzzle is used in many games, such as the Ratchet & Clank platformer series, where the player is required to use allies to move through the environment [33]. One of the most common forms is a doorway that requires multiple characters to hold down a switch to open a door. As soon as any character steps off one of the linked switches, the door closes again. The player must maneuver other characters to step on the switches, then move through the door to find another mechanism that will open the way to let allies through.

The puzzle we created in Splat is shown in Figure 5.9. In this world, there are two different doors which can only be opened through one of the described mechanisms. In the figure, the first door is marked with a double green line in the second room. In the room next door, there are two mechanisms, marked as green squares. Each mechanism must be active before the door opens. Once a character has moved through the door, there is an empty room, then a room with a lever. Once activated, this lever

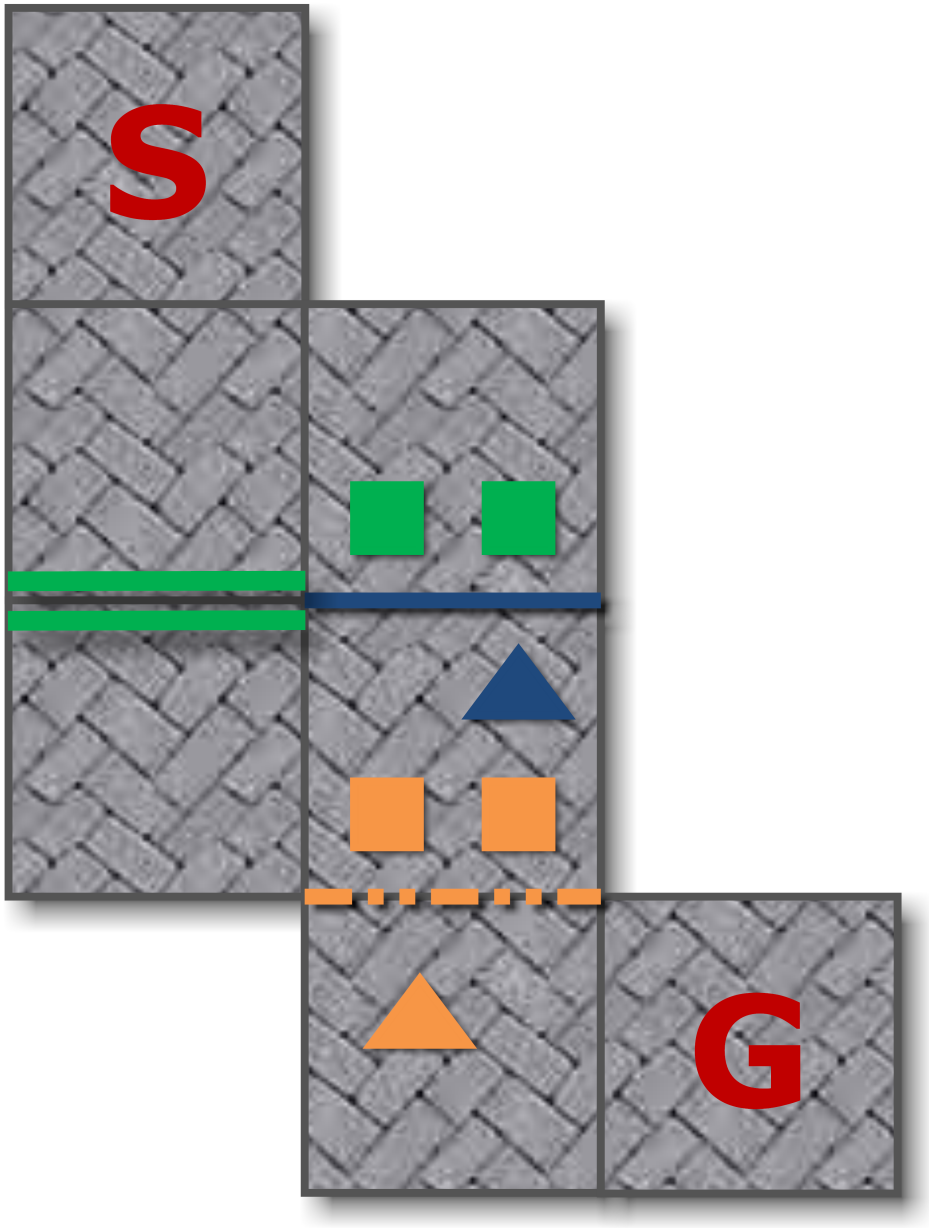


Figure 5.9: Characters start in the room marked S, and must get to the room marked G. Locked doors are marked with heavy lines. The green double line is a door which can only be opened with the green mechanisms in the room next door. The dark blue door can only be opened with the blue lever (triangle) on the other side of the door. The orange dashed door must first be opened with the orange mechanisms, and then the orange triangle to let the last character through.

allows the allies through. Also in this room is another dual mechanism; once this is activated, two characters are able to move through to the next, activate a second lever, and then all characters are able to reach the exit.

Different failures can occur in this type of scenario. Characters may activate only one or the other mechanism at a time, the third character may not make it through the door, or fail to activate the lever to let its allies through. We applied multi-character affordances to the puzzle. Each mechanism has an affordance that is linked to the affordance of its mate. The levers have affordances that open the doors and deactivate the corresponding mechanisms.

Three teams of characters were created for this scenario. The first team was a pure random team. This provides a baseline to see how well characters acting with no direction will perform in the environment. The second team had characters that made random decisions, but used context to choose appropriate affordances. The third team was composed of very simple five layer subsumption characters, seen in Figure 5.10. The subsumption characters would pull a lever if a lever pulling affordance was available, activate a mechanism if a mechanism activation affordance was available, and assist another character if an assist affordance was available. The two remaining layers enabled the characters to deal with closed (and unlocked) doors and move through the world. When moving through the world, characters preferred not to backtrack, but otherwise chose destinations randomly, with no knowledge of the goal location.

We ran 100 trials of 1,000 games per trial for each team, and tracked how often characters successfully escaped, as well as how many time steps it took. Trials were

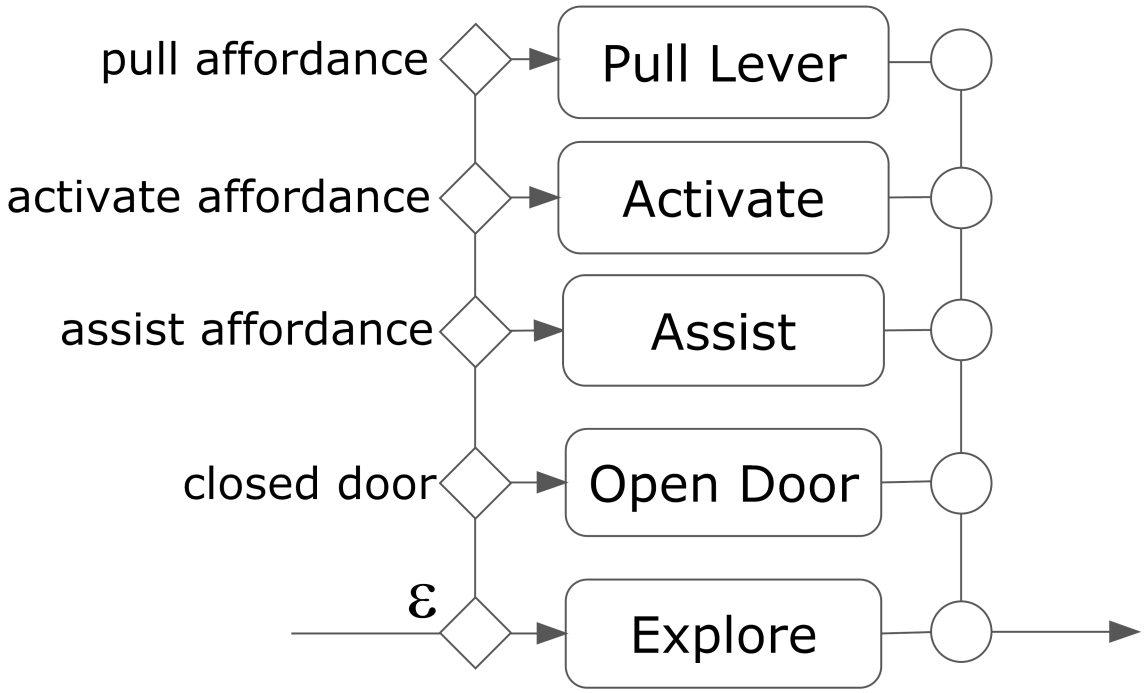
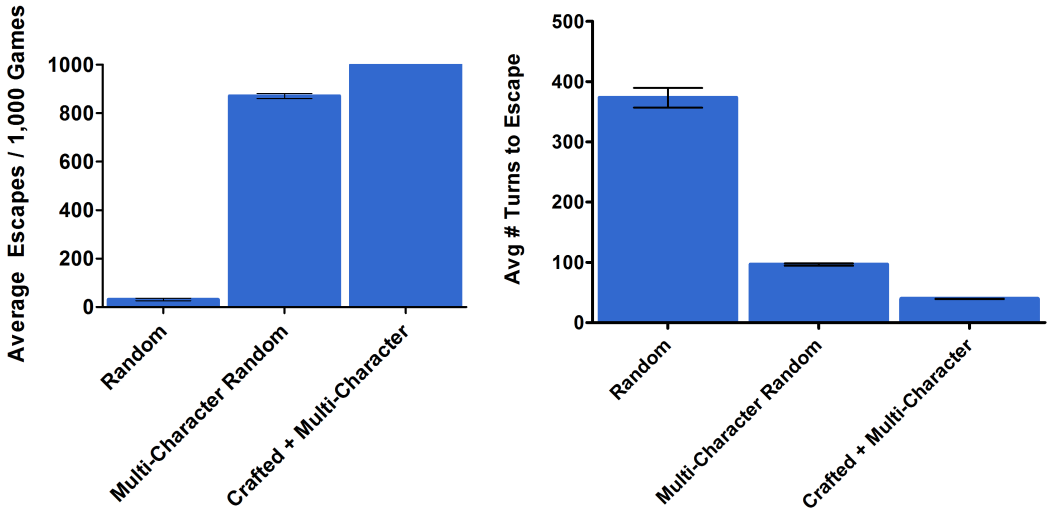


Figure 5.10: The subsumption controller for the multi-character affordance-based agent is five layers, and uses some triggers that activate based on affordances.

ended after 500 time steps. As seen in Table 5.11, even pure random characters occasionally succeeded in the task, though it took an extremely large number of time steps to do so. Characters using context performed significantly better, escaping over 80% of the time, and with a much shorter average escape time. This performance is possible without any prioritization for behaviors on the contextual random characters, and *no knowledge of teammates*. Simply providing a prioritization for the affordances was sufficient to achieve a 100% escape rate in an average of just 39.5 time steps.

Table 5.11: Multi-character Dungeon Performance

Character	Escapes		Avg. Time	
	Mean	SD	Mean	SD
Random	31.4	5.4	373.1	16.4
Contextual	871.2	10.4	96.5	1.9
Crafted	1000	0.0	39.5	0.4



(a) (b)

Figure 5.11: Comparative Performance of random, contextual random, and multi-character affordance enabled hand-crafted characters

Table 5.12: Number lines of code needed for each of the three agent types, divided into one-time (framework) code, per-game code, and per-character code. Hand-crafted auction agent estimate, based on the Splat auction implementation, included as reference.

Type	Framework	Game	Character
Random	284	214	0
Multi-Character Random	396	468	0
Crafted + Multi-Character	508	637	34
Crafted + Auctions	646	637	34

As with contextual affordances, multi-character affordances potentially save the developer a substantial amount of effort in building characters and debugging them. As seen in Figure 5.12 and Table 5.12, multi-character affordances require fewer lines of code to implement. In addition, there are major savings in debugging time per-character, as seen in Figure 5.13 and Table 5.13. The reduction in per-team debugging time is especially important, as the developer may have to develop multiple teams for each level in a game.

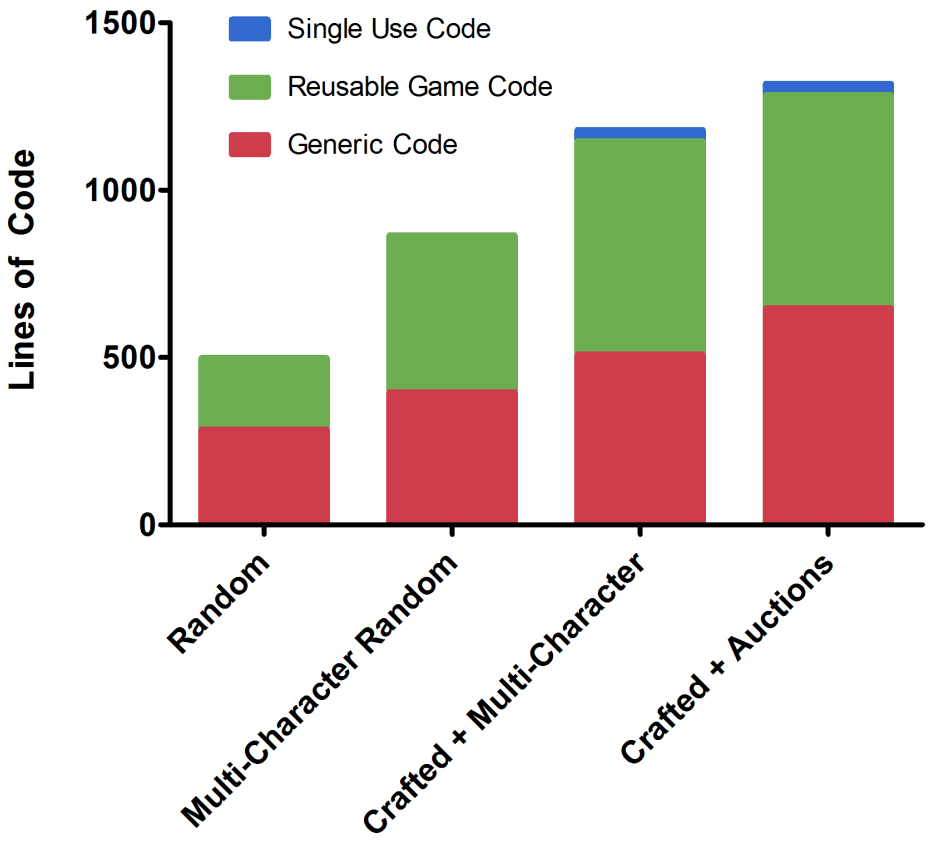


Figure 5.12: Number lines of code needed for each of the three agent types, divided by one-time (framework) code, per-game code, and per-character code. Generic code is reusable many times for different games, game code can be reused multiple times within a game, and single use code is specific to the character. Hand-crafted auction agent estimate, based on the Splat auction implementation, included as reference.

Table 5.13: Approximate debugging time required for building characters, based on implementation within Splat. Auction numbers are estimates based on experiences with implementing auctions in Splat.

Type	Framework	Game	Character
Random	0.5	2	0
Multi-Character Random	1.5	5	0
Crafted + Multi-Character	2.5	9	1.0
Crafted + Auctions	4.0	11	2.0

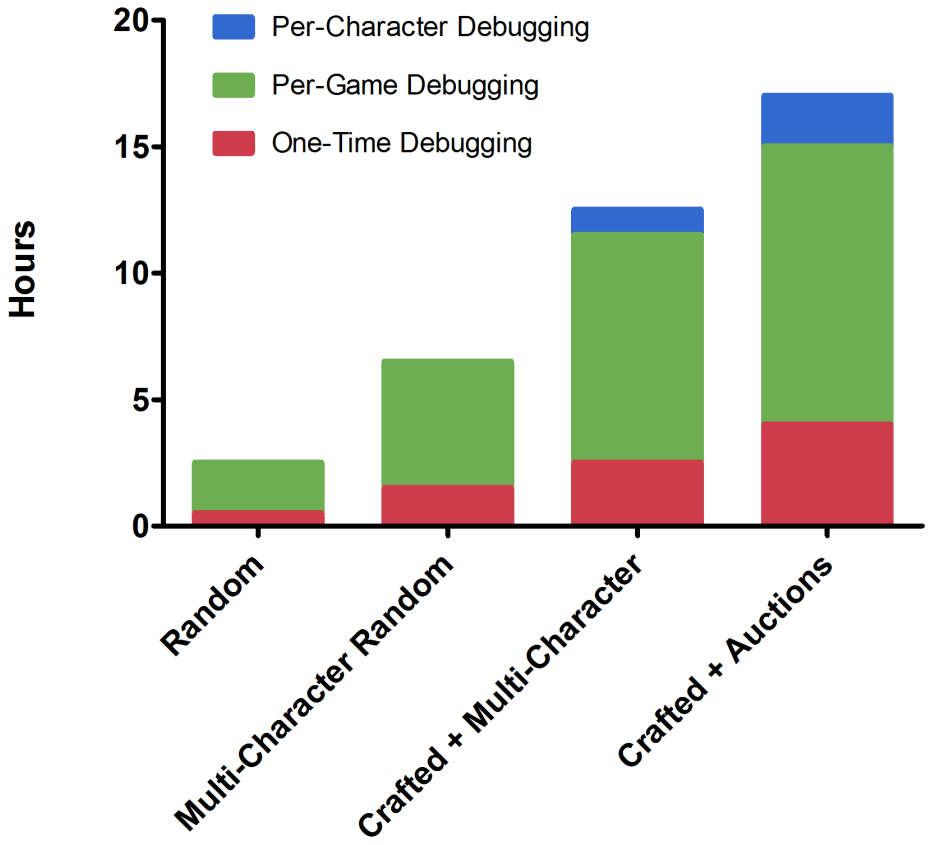


Figure 5.13: Approximate debugging time required for building characters, based on implementation within Splat. Auction numbers are estimates based on experiences with implementing auctions in Splat.

5.1.4 Probabilistic Affordances

To evaluate probabilistic affordances, we used single characters in a puzzle dungeon. For this scenario, we used an RPG-style scenario, shown in Figure 5.14. The player starts at the room marked *S*, and the exit from the dungeon is in the room marked *G*. The character must escape from the dungeon, but doors leading to the exit are blocked, trapping it in a set of 9 squares until it finds the key. In one accessible room there is a set of mechanisms. One of these mechanisms gives access to the key to get through the other doors, one opens a safe containing a bonus treasure, and the other two are traps. One trap results in instant death, one releases a monster (the badger). Players working through a puzzle like this one may make some false starts such as choosing the wrong lever to activate. This knowledge is then carried forward after the player restarts (or *respawns*) at the start of the level after their character has been killed. Probabilistic affordances are useful here, because they give AI characters the same ability as the human player to carry knowledge forward, but without the need for storing memory or having to reason over the likelihood of a given lever having a negative outcome.

In this case, the four levers have probabilistic affordances. Each lever has one affordance with one outcome, and the four affordances are linked. Characters which can make use of the probabilistic information will find the possible outcomes (and the probabilities) adjusted after each interaction with a new lever. When characters are knocked out, they respawn at the starting position, and any items they were carrying are dropped where they were knocked out. Each character also has a sword that can

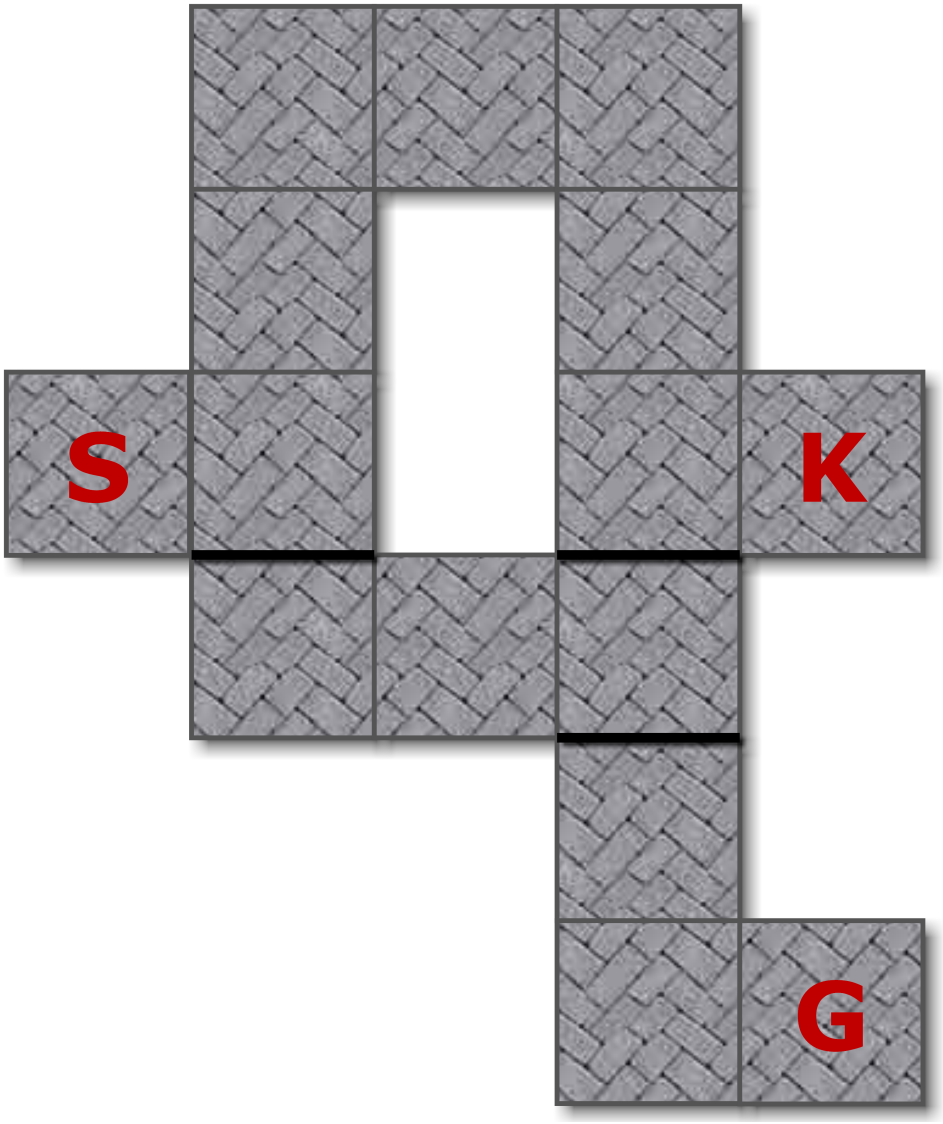


Figure 5.14: Locked doors are marked with dark lines. Characters start in the room marked S, and must make their way to the room marked G. The key to unlock all doors is in the room labeled K. In this room are four levers; one opens the safe holding the key, one opens the safe holding the treasure, one triggers a trap, and one awakens a monster.

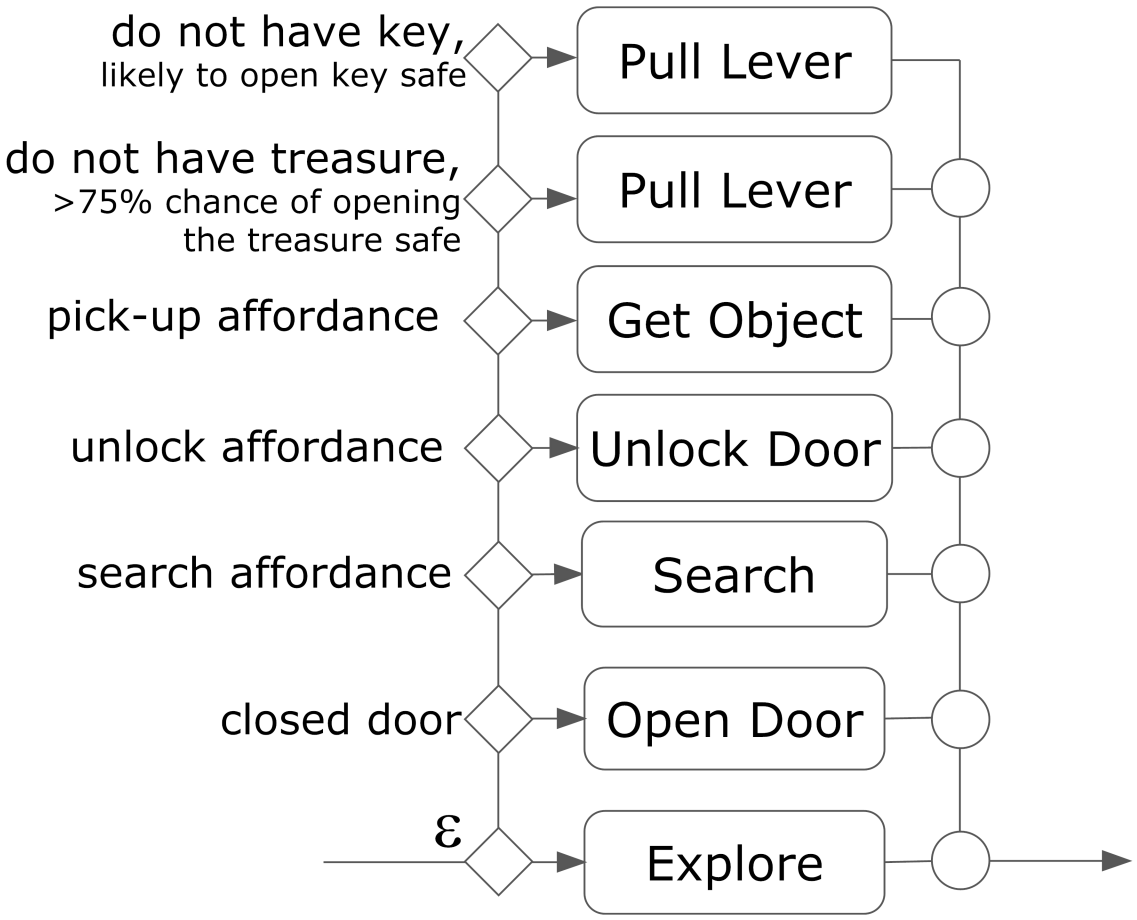


Figure 5.15: The subsumption controller for the dungeon crawling agent is seven layers, and uses some triggers that activate based on probabilistic affordances.

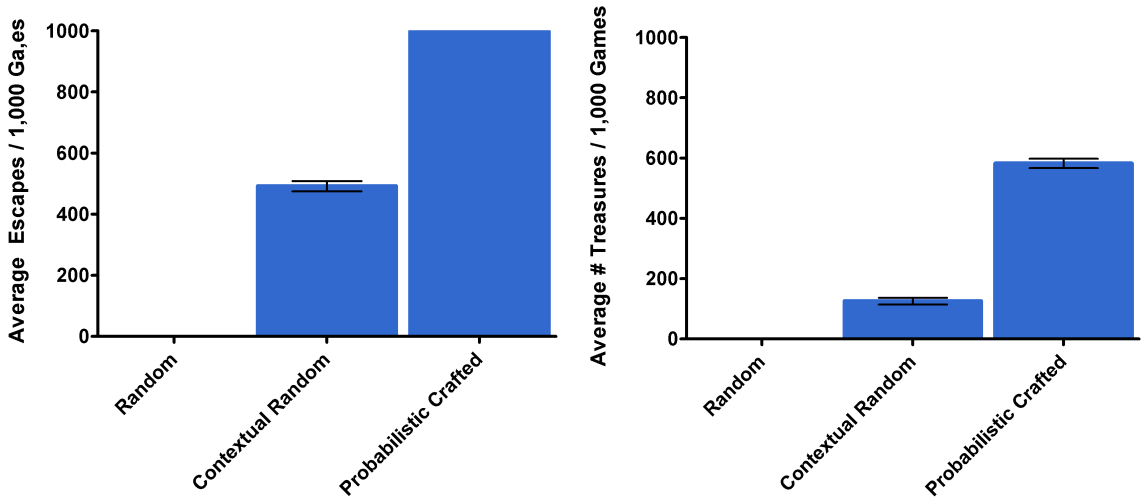
be used for fighting off the badger. This sword is not dropped when the character respawns.

We once again created three characters. The first character is a pure random character, to provide a baseline for random performance in the scenario. The second character is a contextual random character that uses contextual affordances to choose randomly from a list of appropriate actions. The third character is a subsumption-based character, seen in Figure 5.15. This controller has seven layers. The bottom layers enable the character to move around the world by opening doors and randomly

choosing the next room to move to. The character has no knowledge of the goal, or correct direction to move towards the exit, though it does have a preference to avoid back-tracking. The search layer triggers on contextual affordances that indicate an object (in this case, the key safe or the treasure safe) that can be searched. The unlock door layer triggers when the a contextual affordance reports that a door can be unlocked, which requires that the character has the key for the door in its possession. The get object layer will activate if a pick-up affordance is available. The top two layers are more interesting, as these are the layers that use probabilistic affordances.

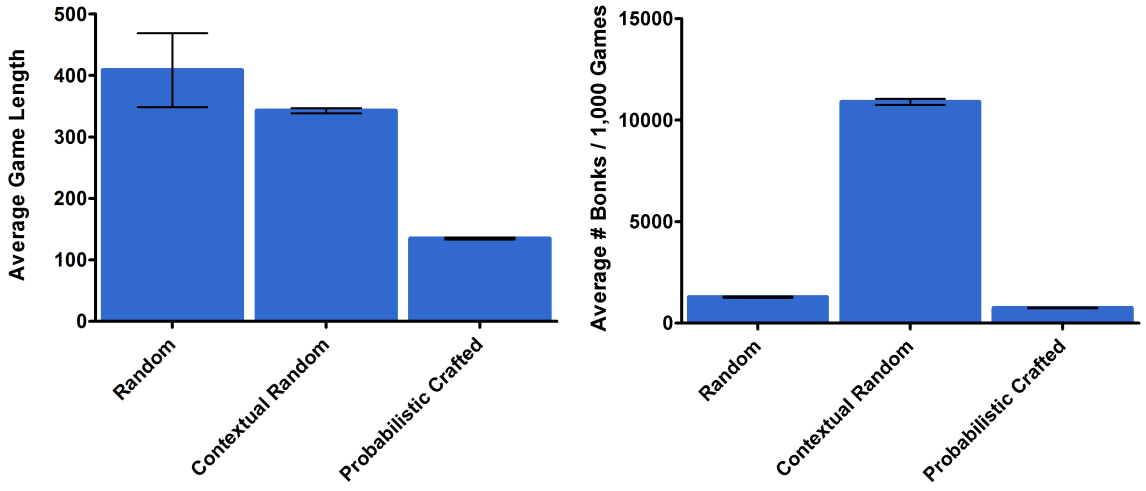
The lower-priority pull lever layer looks for an opportunity to find treasure. It will activate if the character does not have treasure, and it perceives an affordance with a better than 75% chance of opening the safe that has treasure. Specifying 75% prevents the character from doing anything too risky– if it finds the key on the first try, it will not try to find the treasure, because it has a 66% chance of picking a lever with a negative outcome. If it does not find the key until the third attempt, it will know that the treasure is the only remaining possibility, so it is safe to pull the lever. The top layer will activate if the character has a chance to find the key. The "likely" portion of this trigger means that if the chance of getting the key by pulling this lever is *at least as likely* as any other outcome, the character will take the risk of pulling the lever. Note that this controller has no knowledge of traps or monsters, and, unlike the random characters, this character will not defend itself if attacked by the badger.

We ran 100 trials of 1,000 games per trial for each character in the dungeon scenario and tracked the number of times the character escaped, the number of times it got the treasure, how many times it got the treasure, and the average length of a successful



(a) Probabilistic affordance performance (b) Probabilistic affordance treasure retrieval rate

Figure 5.16: Escape rate and treasure retrieval rate for characters using probabilistic affordances vs. random and contextual affordance-only characters. Higher is better.



(a) Average game length (b) Number of bonks per 1,000 games

Figure 5.17: Average length of successful game and average number of bonks per 1,000 games. Lower is better.

Table 5.14: Success Rate in Probabilistic Dungeon

Character	Escapes	Treasures	KOs	Avg. Time
Random	0.4	0.1	1281.3	408.5
Contextual	491.5	125.8	10894.4	342.5
Crafted	1000	582.2	749.5	135.0

Table 5.15: Number lines of code needed for each of the three agent types, divided into one-time (framework) code, per-game code, and per-character code. Hand-crafted with memory model and probabilistic reasoning, based on the BEHAVEngine memory model implementation, included as reference.

Type	Framework	Game	Character
Random	284	207	0
Contextual Random	310	461	0
Crafted + Probabilistic	550	630	48
Crafted + Memory + Reasoning	498	730	43

run. After 500 time steps, the simulation would end if the character had not escaped. As seen in Table 5.14 (and Figures 5.16 and 5.17), acting randomly is a poor approach to this world. The pure random character almost never escapes, and never gets the treasure. The random character suffers far fewer KOs than the crafted character, but only because it rarely makes it to the key room. The contextual random character does better, escaping almost 50% of the time, but still rarely getting the treasure. The contextual random is frequently knocked out by traps or the badger, even though it can defend itself. Finally, the seven-layer crafted agent escapes every time and also retrieves the treasure 58% of the time. This crafted agent also escapes far more quickly than the other agents.

Implementing probabilistic reasoning using a memory model can be a complex task, especially at the level of building behaviors. Figure 5.18 and Table 5.15 show a comparison of the number of lines of code required to implement these characters, along with an estimate of the code required to implement a full memory model

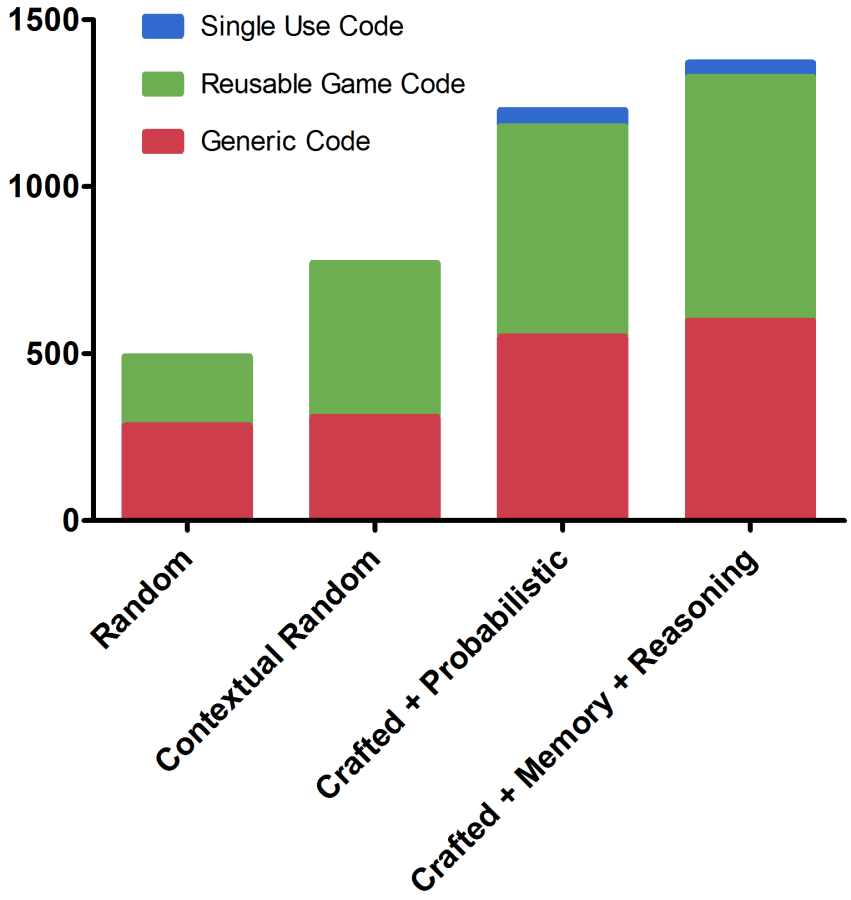


Figure 5.18: Number lines of code needed for each of the three agent types, divided by one-time (framework) code, per-game code, and per-character code. Generic code is reusable many times for different games, game code can be reused multiple times within a game, and single use code is specific to the character. Hand-crafted with memory model and probabilistic reasoning, based on the BEHAVEngine memory model implementation, included as reference.

Table 5.16: Approximate debugging time required for building characters, based on implementation within Splat. Crafted + Memory Model + Probabilistic Reasoning estimate based on experience with BEHAVEngine and Splat.

Type	Framework	Game	Character
Random	0.5	2	0
Contextual Random	1.0	4	0
Crafted + Probabilistic	2.0	10	1.5
Crafted + Memory + Reasoning	4.0	9	3.0

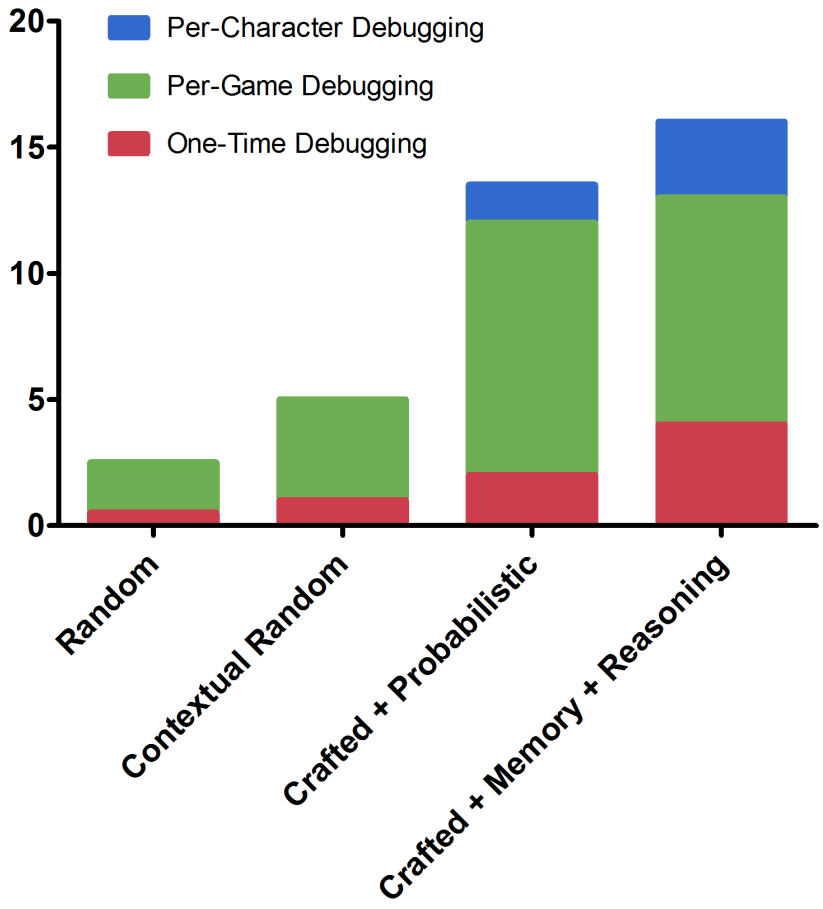


Figure 5.19: Approximate debugging time required for building characters, based on implementation within Splat. Crafted + Memory Model + Probabilistic Reasoning estimate based on experience with BEHAVEngine and Splat.

and behaviors with probabilistic reasoning. This estimate, along with the debugging time estimate in Figure 5.19 and Table 5.16, are based on the implementation of the memory model in BEHAVEngine. It is also important to note that while the per-game debugging time for probabilistic affordances is greater than the per-game debugging time for a full memory model and probabilistic reasoning system, the debugging process can be divided more easily for probabilistic affordances. When using probabilistic affordances, more control is given to the level designers, and more of the agent debugging process can be wholly claimed by designers. This process is also more integrated with the level design process than it is when developing a memory model and behaviors that use probabilistic reasoning. This can be a very important factor in the game development process, and may provide greater savings than those shown here.

5.1.5 Extended Affordances in Game Design

Extended affordances reduce the complexity and development time of character controllers. This reduced complexity can be conveyed through to the character design. Character design frequently uses graphical user interfaces, and the responsibility for building characters may be shared with designers and others who are not AI programmers. Simplifying the character design can help make the process of training non-experts faster and less error-prone.

Extended affordances can have an even greater effect on the game development process when removed from the AI development process altogether. Rather than having the AI programmer build smart objects and contextual affordances, these tasks can be assigned to the level designer. Placing control of AI behavior into the

hands of the level designer allows much more creative control when building the game world. Instead of a process in which the AI programmer creates a behavior, the game designer requests modifications, and the AI programmer tweaks the behavior, the game designer can now control behavior directly. Controlling this behavior allows more creative design work to go into the game experience.

This is a more appropriate distribution of work between the AI programmer and the designer, as it is the designer who manages the entire game experience. In addition, particular parts of a game may be designed to evoke specific responses from the player, and there is a risk of these responses getting lost in the translation process from designer to AI programmer. Finally, it is worth noting that much of the information used for probabilistic and multi-character affordances must already be built into the game environment. Extended affordances can be implemented to use this level information directly, rather than needing to create additional data structures to convey the relationships between game objects.

5.2 Evaluation of Subsumption Architecture Extensions

We evaluated the subsumption architecture extensions using both DASSIEs (FI3RST with BEHAVEngine) and Splat. Initial evaluation with FI3RST and BEHAVEngine provided testing in a high-resolution 3D environment, and proved the usefulness of the extensions in a resource-constrained application. Further evaluation was performed with SPLAT, focusing on teams of characters working together in a scenario.

5.2.1 Reactive Teaming Examples

Reactive teaming has been evaluated with simple scenarios as case studies. Our first scenario included four generic agents and a fully specified agent with behaviors

describing a number of patrol routes in the game environment. The generic agents started up, requested behaviors from different agents until the fully specified patrol agent transferred patrol routes. Communication was treated as independent of distance, so the agents were able to immediately receive behaviors without finding the fully specified agent initially.

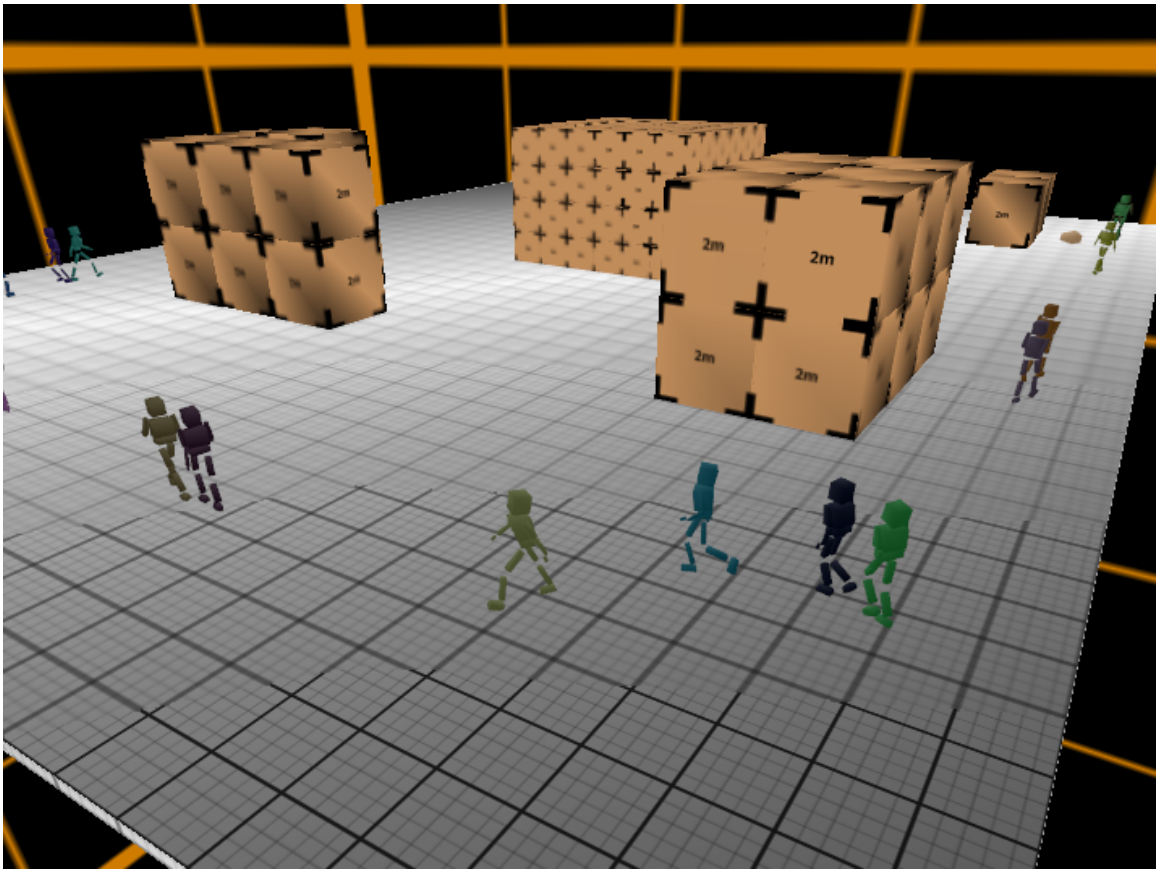


Figure 5.20: Patrol agents in the FI3RST environment. In this scenario, a single agent with knowledge of a large (12 point) patrol route is placed in the world. Generic agents are added, and the patrol behavior is divided into segments.

An additional test scenario created a single garbage-cleaning robot which was to wander through the world and pick up objects. Twelve generic agents were added that wandered the world, and in this case, would request behavior transfers only when within range of one another. Once a generic agent received the garbage-cleaning

behavior, it would continue to wander, and transfer this behavior to other agents if requested. This scenario was extended to place two types of garbage-cleaning bots to pick up different types of objects. Two types of objects were added at different times. As the first type of object was cleared, failure and inactivity transfers occurred, adapting the team to gather the second type of object.

More advanced coordination was tested with the scenario shown in Figure 5.20. In this case, a patrol agent with a large patrol route was placed in the world, and varying number of generic agents were added to the world. The patrol behavior used behavior division to split the patrol route into segments, halving it each time it was transferred until each agent was walking a two-location segment of the route. The AI engine was able to run over 100 agents at 10hz with a low CPU load on an Intel Core 2 6400 system running at 2.13ghz. Larger numbers of agents were not tested as the graphics pipeline became overloaded at this point.

5.2.2 Reactive Teaming Behavior Distribution

The effectiveness of reactive teaming with failure detection can be demonstrated with a case study in our simulation environment, seen in Figure 5.21. In our scenario, the first agent, Alice, searched the environment for cans. The second agent, Bob, searched the environment for boxes. The goal of the agents is to pick up all of the objects scattered throughout the environment. The environment is initially loaded with many cans placed in the world. After a period, a large number of boxes are added to the environment. With static controllers, each agent will perform only the task initially designed. This means that while Alice is well-occupied for the first part of the scenario, Bob is performing very little work. Two different variations of the

scenario were run, one using reactive teaming, and one with static characters.

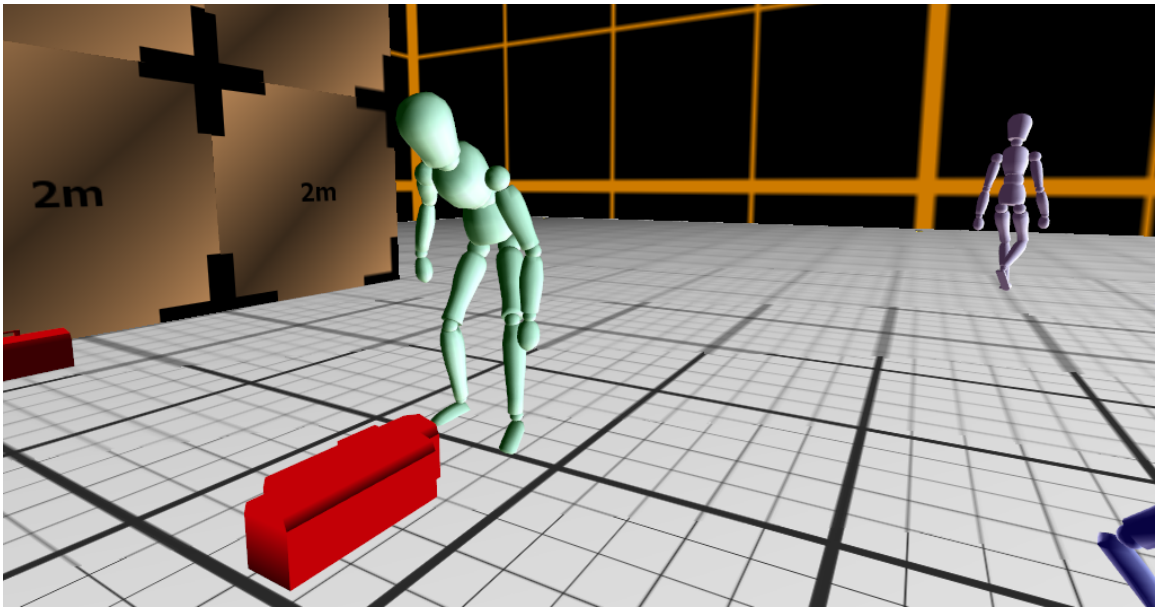


Figure 5.21: Failure detection enables subsumption-based characters to coordinate in teams.

In the static scenario, Alice was replicated to create 11 characters that searched the environment for cans. Bob was replicated to create 11 characters that searched for boxes. Eleven agents were guaranteed to be idle for the first half of the scenario, when no boxes were in the environment, while in the second half, 11 characters were mostly idle as very few cans were present. The overall team was very inefficient because at any given time, up to half of the characters could not perform a useful task.

The reactive teaming scenario allowed better team utilization. Instead of 11 characters in each role, we used Alice, Bob, and 20 *generic* agents. Generic agents have a base layer, but instead of being designed to perform a task, they are designed to have activation failures. These activation failures lead the agents to use reactive teaming to request behaviors from other characters. Figure 5.22 shows the number of characters with each behavior vs. time, averaged over 10 trials of 120 seconds each. As the

scenario started, each of the 20 agents randomly received either the behavior to pick up cans or the behavior to pick up boxes based on whether they meet Alice or Bob first. Since boxes were nonexistent for the first part of the scenario, the can-gathering agents were quite productive, but the box-gathering agents continued to generate activation failures. The box-gathering agents requested new behaviors, resulting in a surge of can-gathering agents.

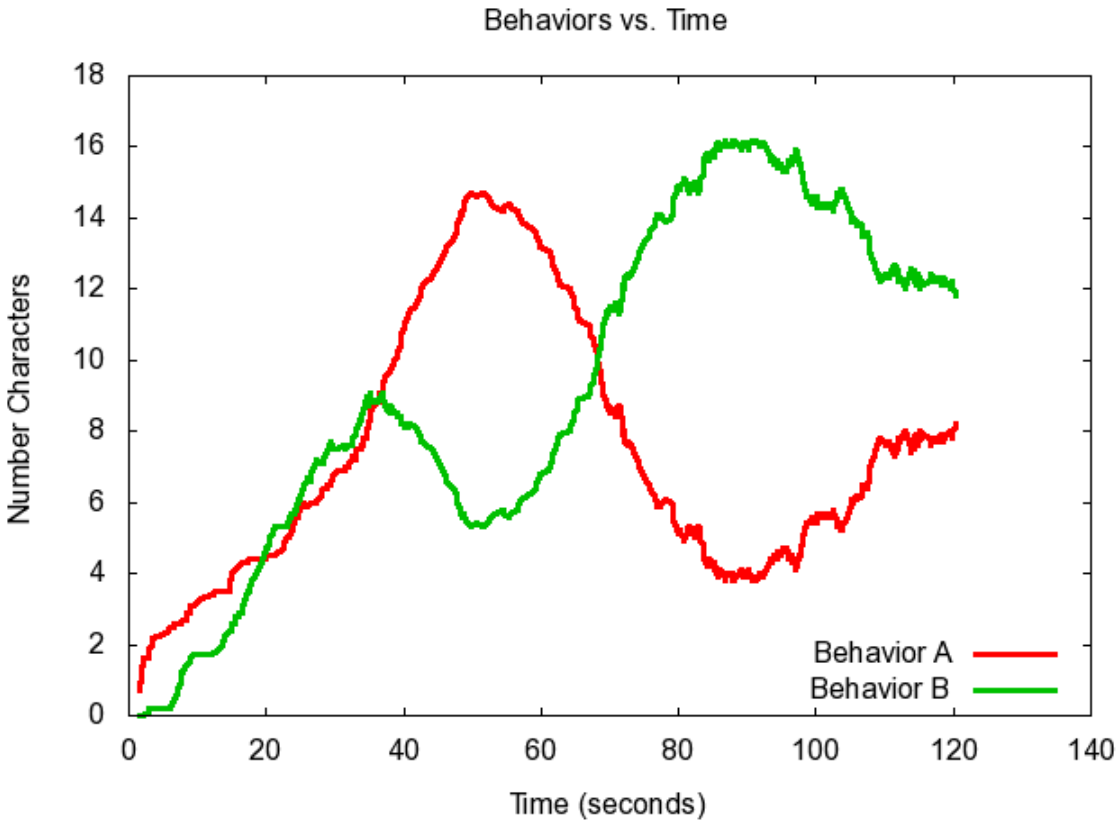


Figure 5.22: Reactive teams adapt to environmental changes. Initially, neither behavior is relevant and they spread at the same rate. As Behavior A becomes relevant in the environment, it peaks, and the number of characters with Behavior B decreases. Once Behavior A is no longer relevant, Behavior B takes over.

As the scenario continued, the boxes were added to the world. With most of the cans gone, the can-gathering agents generated activation failures, causing them to

request new behaviors. The number of box-gathering agents then surged, as the majority of agents received the appropriate behavior. Once all but a few objects have been gathered, the distribution of behaviors changed again, as all of the behaviors had become idle.

Failure detection and behavior stack modification enables the reactive teaming technique. As shown in the scenario, flexible teams that react to changing conditions can be created with this technique. The scenario provides an example of how using the dynamic reactive teaming approach results in better team utilization than the static approach.

5.2.3 Comparison of Team Approaches

Using Splat, we compared different approaches to building teams of characters. In addition to reactive teaming with subsumption-based characters, we implemented a centralized auction to distribute team tasks. We tested the auction-based method with both subsumption and with behavior trees in a fast-paced tactical environment.

Auction-based subsumption used dynamic layers to achieve task distribution. Every five turns in the game world characters would bid on the set of team tasks. Two team tasks were used: one team task was to attack hostile characters, and the other was to search for treasure. The bid scoring for the auction used whether the agent was capable of executing the behavior for the given task, the distance of the character to the given task (if it had a specific location), and was weighted by whether the character was already executing the task. The priority function for task behaviors in subsumption assigned the behaviors a priority of -1 (which prevented them from executing) if the task was not assigned, or an appropriate priority if it was assigned.

For behavior trees, we made task assignment one of the preconditions for the task behaviors. If the task was not assigned, this prevented the characters from executing the task. Otherwise, the character would execute the task at the appropriate point in the tree.

Each team in the scenario had two members. We compared seven teams: two teams with generalized characters that had both team behaviors (one using behavior trees, one using subsumption), two teams with specialized characters that had only one task or the other specified (one using behavior trees, one using subsumption), one auction-based subsumption team, one auction-based behavior team, and one reactive subsumption team. Each team used the same set of behaviors, though specialized characters and reactive teaming characters had behaviors ommitted. Auction-based teams had the opportunity to reassign roles every five turns.

We expected that teams with fully specified characters capable of performing each task would perform better than other teams, but that dynamic characters on auction-based or reactive teams would adapt better than specialized characters. We used the same Splat world as in Section 5.1.2, though it was modified to be a high-resolution environment, with each room varying in size from 9 square meters (the guard post) to 25 square meters (the courtyard). Actions each had a distance requirement applied, so that characters had to move within each room. Characters could move 1 meter in a turn or take an action. The goal was still two parts: knock out the opposing team and get the treasure. Once knocked out, characters would not respawn.

Each team combination competed for 1,000 trials in alternating starting points. Results are shown in Table 5.17 and Figure 5.24, with each row showing the percentage

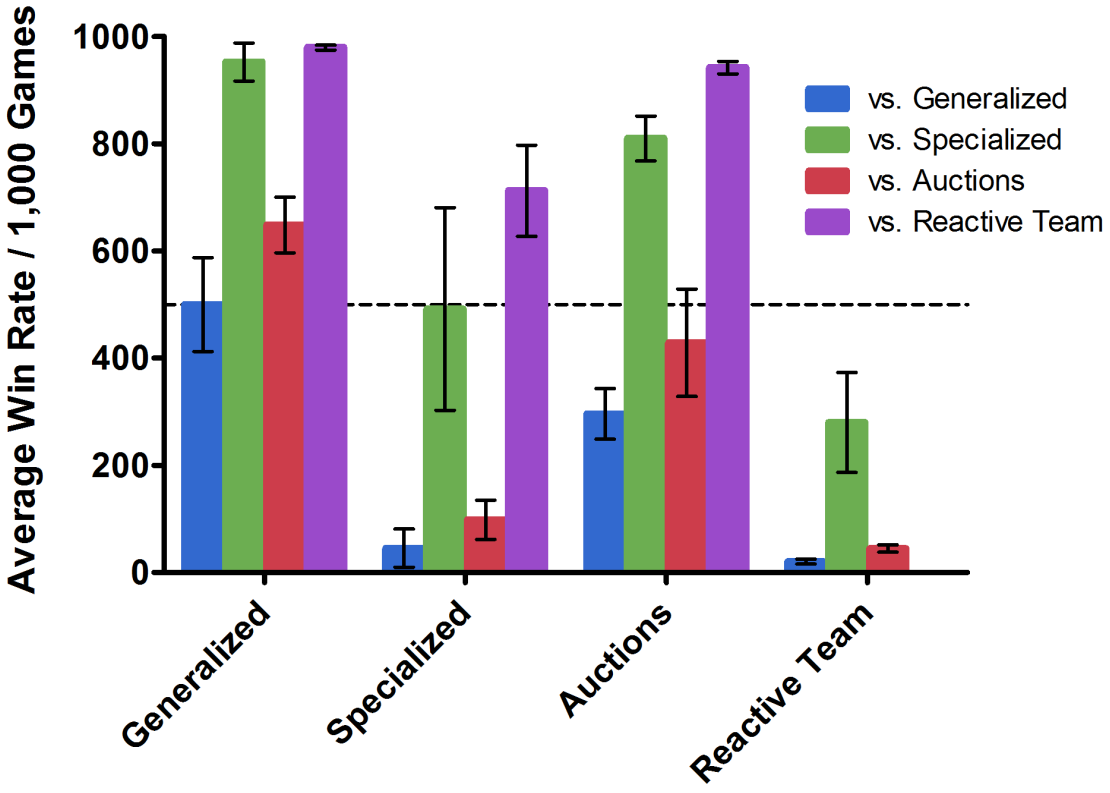


Figure 5.23: This chart shows the performance of the different methods visually. For each group, the vertical label shows the architecture being tested, and the bars show the percentage of trials that a team using this architecture won the scenario.

win rate against the technique listed in the column. As expected, fully specified agents outperformed all teams in this task, with auction-based teams doing better than specialized agents. Reactive teams, surprisingly, performed very poorly. Note that some trials resulted in stalemates (this is why some pairings do not add up to 100%). This is an important result, as it highlights the circumstances under which reactive teaming does *not* perform well. In comparison to the object gathering task, which is a long term scenario in which errors are not fatal, the Splat treasure hunt is much shorter, with very limited goals that can be accomplished once. Errors in the treasure hunt are fatal, and agents cannot recover from being knocked out. This

means that behaviors will typically only activate a small number of times during the scenario, with one of the team behaviors only executing a single time.

Tactical environments like this one do have a variety of well-defined team roles, and in games similar to the treasure hunt (such as the common capture-the-flag game), teams may break into groups with some players providing protection for others that are performing important tasks (such as finding and capturing the treasure or flag). Based on these results, reactive teaming is not well-suited for this type of team task. Subsumption with dynamic priorities assigned through an auction are better suited for these types of scenarios. Auctions score agents based on suitability given the location of the goal or ability of the character to fight, and will react more quickly than subsumption, which will not change roles until a sufficient volume of errors indicate that the current behavior set is not performing well.

Reactive teaming *can* still be used in these situations, however the *command* strategy should be used for coordinating. To test this, we ran an additional team test. We used the same scenario as for the first test, but tested just three teams. Each team also had three agents instead of two. We used the subsumption architecture for each team, as the previous results showed that subsumption and behavior trees performed similarly on these tasks.

The first team was composed of specialized subsumption characters. One character was specialized to attack other characters, one to search for treasure, and the third was limited to gathering treasure. For the second team, we used auction-based subsumption coordination. When bidding occurred, two of the three characters were allowed to bid, and the third was excluded. Finally, the third team used two limited

subsumption characters that did not know how to attack or search, but were commanded by a third character. This commander had a special command behavior that would decide the best tasks to give to its teammates and send them orders. The two teammates would then request a behavior from the commander, and the commander would pass on the chosen task.

As shown in Table 5.18, the reactive teaming characters performed much better than in the first test. Once again, some games resulted in stalemates. The number of stalemates is relatively higher because of the third character that could not perform either team task. This comparison shows that reactive teaming can be used in some types of games where it would otherwise be weak if the command strategy is used. This is expected, as the command strategy can effectively implement many different multi-agent coordination techniques. It does create a level of centralization on teams, and this may be desirable in games– when the commander of a hostile force is eliminated, the ability of the force to adapt to the situation may be adversely affected.

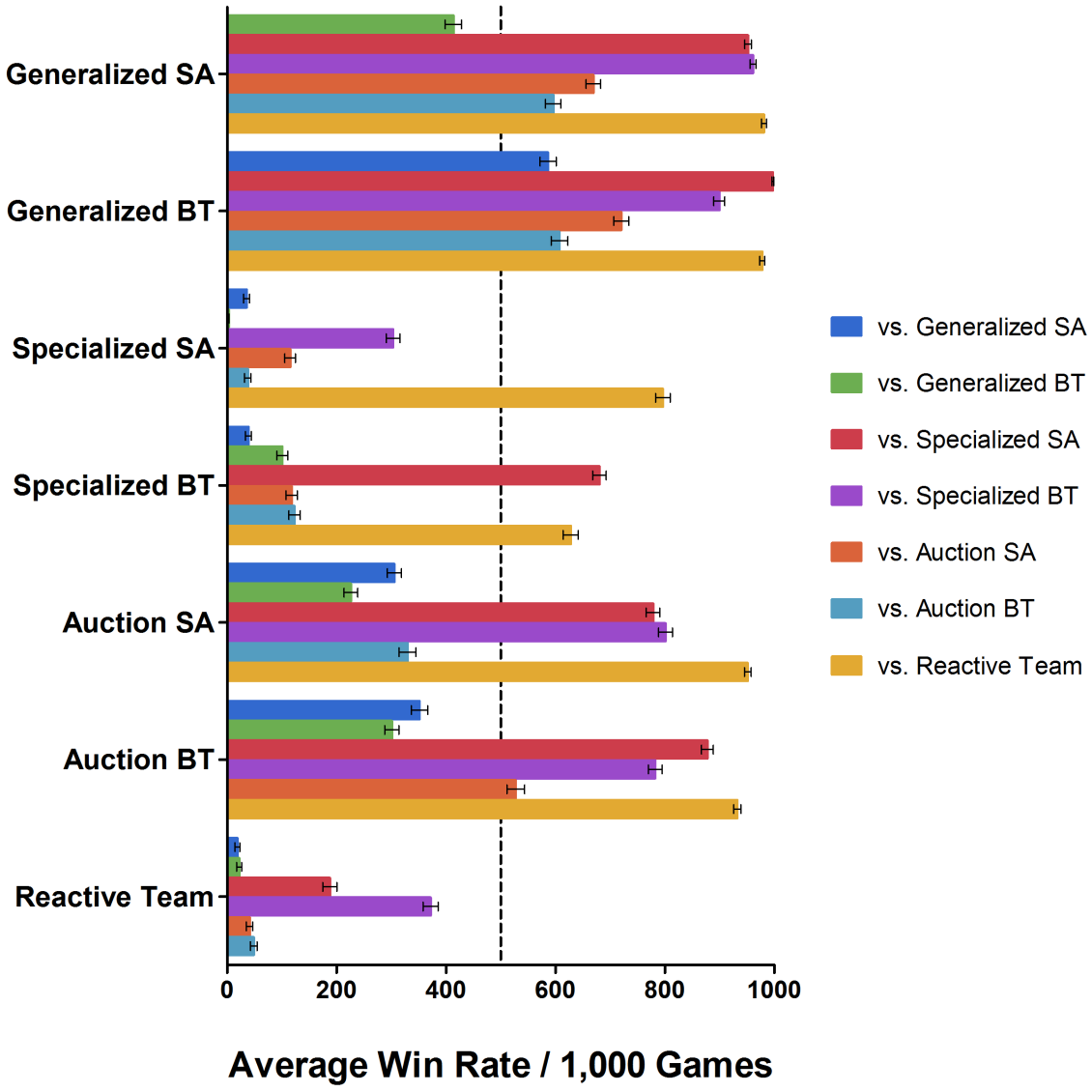


Figure 5.24: This chart shows the performance of the different methods visually. For each group, the vertical label shows the architecture being tested, and the bars show the percentage of trials that a team using this architecture won the scenario.

Table 5.17: Comparative Performance of Teaming Methods: Values are average number of wins over 1,000 games (based on 100 trials of 1,000 games).

Team	vs. SA Gen		vs. BT Gen		vs. SA Spec		vs. BT Spec		vs. SA Auc		vs. BT Auc		vs. Reactive	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
SA Generalized	-	-	413.3	14.7	951.9	6.7	961.0	5.4	669.4	13.2	596.0	14.2	981.1	4.4
BT Generalized	586.7	14.7	-	-	997.2	1.7	899.4	10.0	720.3	13.8	607.5	15.3	978.1	4.5
SA Specialized	35.6	5.6	2.5	1.2	-	-	303.0	12.6	115.0	9.9	37.7	5.6	796.4	13.3
BT Specialized	39.0	5.4	100.6	10.0	680.4	12.4	-	-	117.8	10.4	123.2	10.4	628.2	13.8
SA Auctions	305.8	13.0	225.9	12.5	778.5	12.5	801.2	12.8	-	-	329.7	15.4	951.5	6.1
BT Auctions	351.5	14.9	300.9	12.9	877.7	10.7	782.4	12.7	527.4	15.9	-	-	932.4	6.8
Reactive Team	18.9	4.4	21.9	4.5	187.8	12.9	371.8	13.8	41.1	5.7	48.7	6.0	-	-

Legend:
SA Gen: General Subsumption Architecture Team
BT Gen: General Behavior Tree Team
SA Spec: Specialized Subsumption Architecture Team
BT Spec: Specialized Behavior Tree Team
SA Auc: Subsumption Architecture Team with Auctions
BT Auc: Behavior Tree Team with Auctions
Reactive: Subsumption Architecture Team with Reactive Teaming

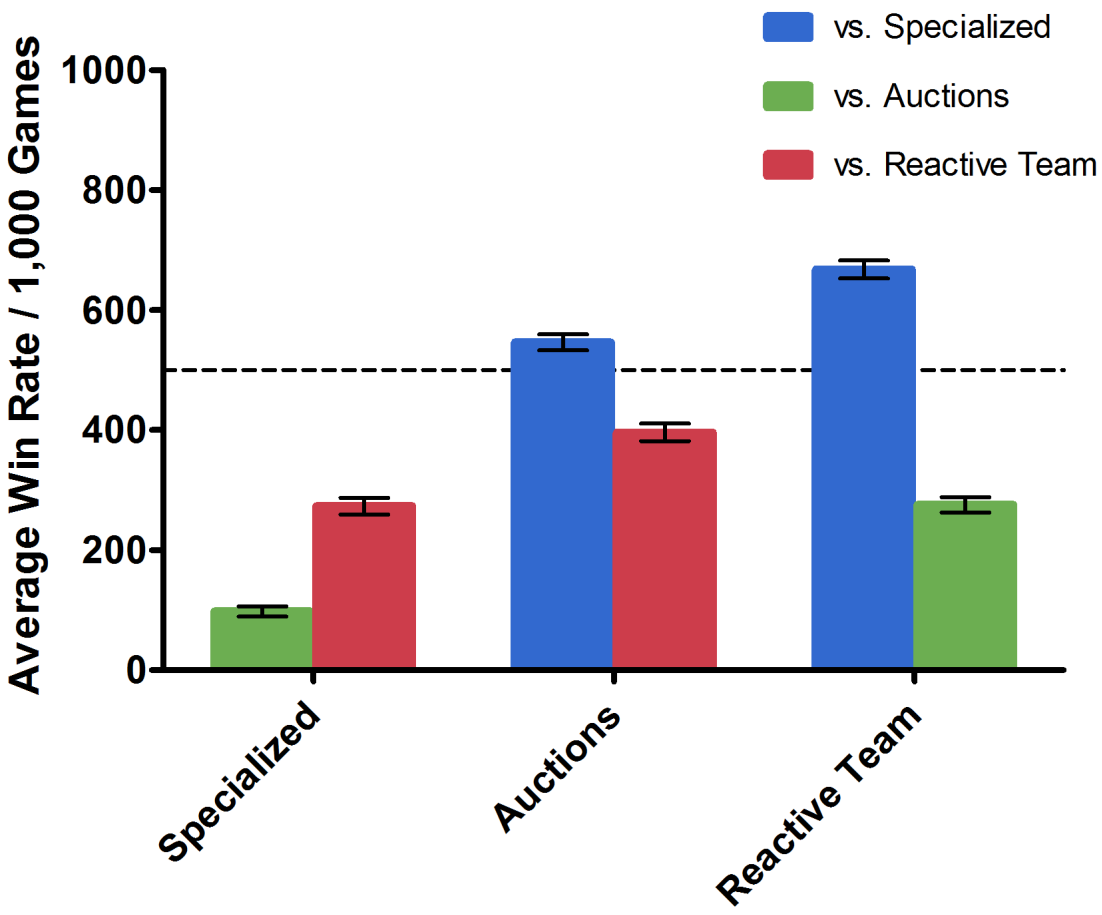


Figure 5.25: This chart shows the performance of the different hierarchical teams. For each group, the vertical label shows the architecture being tested, and the bars show the average number of trials, out of 1,000, that this team won.

Table 5.18: Performance of Command-based Reactive Teams. Values are average number of wins over 1,000 games (based on 100 trials of 1,000 games).

Team	vs. SA Spec		vs. SA Auc		vs. React Com	
	Mean	SD	Mean	SD	Mean	SD
SA Specialized	-	-	97.7	84.4	273.2	13.6
SA Auctions	546.2	13.4	-	-	395.9	14.8
Reactive Command	667.4	15.1	275.3	12.6	-	-

CHAPTER 6: CONCLUSIONS

Artificial intelligence in games serves a different purpose from artificial intelligence in most other applications. Game AI must support and enhance the experience of the player to make a game more interesting, more engaging, and more challenging. This dissertation focuses on new approaches to creating intelligence in games, looking at not just the game characters, but also the game environment. We look at how intelligence can be embedded in the game world itself, building upon the particular advantage that games have: the environment is completely flexible, and creating the best experience requires integrating the environment at every step. In addition, we examined ways in which inexpensive AI techniques can be extended to build intelligence that is faster, cheaper, and easier to design.

The central hypothesis of this work is *that behavior-based characters in games can exhibit effective strategy and coordinate in teams through the use of knowledge embedded in the world and a new dynamic approach to behavior-based control that enables characters to transfer behavioral knowledge.* We have presented several techniques that test this idea, and evaluated their effectiveness in comparison to existing industry-standard techniques. We have shown that the use of navigation meshes can greatly improve the efficiency of existing methods for annotating the environment, making the environmental annotations of influence points usable for more resource-

constrained systems. We have demonstrated that embedding knowledge into the environment through the use of advanced affordance techniques can reduce the complexity of character controllers, and enable strategic behavior with probabilistic reasoning. This complexity reduction is accompanied by a simplification of the agent development process by increasing the reusability of individual components and decreasing the development time for intelligent characters. Finally, we have shown that dynamic extensions to behavior-based subsumption architecture can enable inexpensive team coordination methods, making it possible to build very large adaptive teams of intelligent characters.

Together, these new methods reduce the computational and development costs for intelligent game characters. They demonstrate that inexpensive behavior-based and reactive control methods can be used to produce rich behavior, while simultaneously shifting more control to level designers. The centrality of AI to modern games makes this an important change. The shift improves the ability of designers to craft the full game experience more successfully and directly, with fewer opportunities for mistranslation between designer and programmer.

6.1 Environmental Intelligence

A major advantage of virtual environments is that they are, by definition, virtual and not limited by physical constraints. This allows the world itself to be used as a mechanism for sharing information as it is with influence points. The physical constraints of the real world do have some advantages, though—when looking at an object in the real world, we can pick it up, turn it around, hold it, throw it, or poke at it to discover how it can be used—all actions which are less meaningful in game

environments. These actions convey information and knowledge, and represent a sort of embedded intelligence in the world itself, described in psychology as *affordances*.

The key is to create this information in virtual environments. Once information about using objects and how they interact is embedded into the environment, it is natural to then move even more information into the environment. Contextual affordances allow the designer to inform game characters how actions change under different conditions, reducing the burden of designing intelligence which has to reason about special cases throughout a game world. Multi-character affordances give us information about how characters should work together to change the environment, without the need to design AI that explicitly communicates between characters. Finally, adding a small amount of information to these affordances can even provide a simple way to enable characters to reason about uncertainty without having to store any additional knowledge.

There are two major advantages in using environmental intelligence: first, it simplifies characters. The characters with which we evaluated our extended affordances had no memory model, only extremely simple navigation, and minimal behavioral reasoning. This is an advantage because debugging AI is an extremely challenging problem. Intelligent characters must be tested against each situation that will arise, and there are few shortcuts that can be taken in that process– in the end, they must be placed into environments and situations that are very similar to the final game world. Environments, on the other hand, can be tested much more easily. Affordances can be easily tested using a human player doing a walkthrough in an environment before the AI characters are even complete. Objects with affordances attached, even complex

affordances, can be potentially reused many times throughout different environments within a single simulation. When building environment-specific affordances, such as multi-character affordances, all of the information necessary to create the affordances is already created in the process of building the level. Outcome-oriented affordances, such as probabilistic affordances, likewise use information which is already present in the level (what the action causes to happen in the world). Adding similar levels of intelligence to the characters themselves requires reasoning about team actions and uncertainty.

Environmental intelligence integrates smoothly into the game development process. Contextual, probabilistic, and multi-character affordances are tools that may be during the level design process. This puts more control directly into the hands of the game designers.

6.2 Dynamic Subsumption Architecture

Reactive intelligence in general and subsumption architecture in particular are designed as static techniques. Using reactive control techniques as dynamic techniques provides substantially more power. Our major extensions for subsumption include a decoupled architecture that separates low-level perception and action processing from the main decision process, failure detection, dynamic priorities, and behavior transfers. Decoupling the architecture allows the creation of more generalized behavior controllers, so that the same controller could be applied to a variety of characters with different perception and action capabilities. Failure detection can be used in a variety of ways, but the two most important uses are for debugging character AI and to enable run-time reorganization of the controller. Dynamic priorities potentially

simplify characters with redundant behavior layers while also allowing the simple application of auction-based team coordination. Reactive teaming can be used to build adaptive teams in scenarios with large-scale or long-term tasks performed by moderate to large sized teams.

Our evaluation suggests that the best scenarios for reactive teaming are scenarios with relatively long-term goals, and recoverable failures that inform behavioral performance directly. Tactical 3PS and FPS games tend to have short term goals—capture the flag, destroy the opposing team—with fatal failures that do not provide direct feedback about the behaviors the character is performing. Respawning in these games gives the characters opportunities to recover from even these failures, but when respawns occur, the character capabilities change again, so these failures do not inform the behavior transfer process very effectively. With long-term goals, such as gathering resources in an RTS game, characters experience failures (the gold mine being exhausted) that are more lasting and can actually inform behavior transfers (when the mine is exhausted, the gatherers will propagate a behavior to obtain a different resource).

Between reactive teaming and dynamic priorities, our extensions for behavior-based subsumption allow multi-agent coordination to be applied to teams of computationally inexpensive characters. Reactive teaming can handle large teams that need to be adaptable to long-term environmental changes or smaller teams of directly commanded characters in a hierarchy. Dynamic priorities also allow standard methods for teaming such as auctions to be applied to the subsumption architecture. This is an important achievement as subsumption is an architecture that, as our studies with

BehaviorShop show, is extremely accessible to non-AI experts.

6.3 Team Protocols and Virtual Environments

While we address the use of our team coordination approaches in tactical and strategic games, they are applicable beyond these areas. When using these methods for other types of applications, there are a few factors that should be considered. The type of team coordination used depends on the types of team interactions that are required. Broadly, characters can have loosely specified team roles or tightly specified team roles. For loosely defined teams, multi-character affordances are most appropriate. For tightly defined teams, reactive teaming should be used.

In loosely specified teams, characters will perform actions opportunistically. Characters on these teams should perform tasks as they become available, and may not be specialized for one task over another. In our evaluation, we used a scenario where characters had to work together to open doors and traverse a dungeon. The task of activating mechanisms and pulling levers could be performed by any character, and specializing characters for one or another would provide no benefit. There may be some environmental adaptation, but this adaptation focuses on discrete, short-term task availability. Transferring behaviors would delay task execution and generally degrade performance.

For tightly specified teams, characters take responsibility for a specific set of tasks. Reactive teams are more appropriate to use in these situations, either as adaptive teams (using non-command policies for coordination) or as hierarchical teams (using the command policy for coordination). On these teams, characters will not perform actions opportunistically, but instead focus only on their assigned tasks. This ap-

proach is more appropriate when characters require specialized resources to execute particular behaviors, or in highly tactical environments (such as sports games) which require characters to coordinate through behavior rather than the environment.

In addition, the purpose of the simulation should be considered. When evaluating users in a training simulation, multi-character affordances can provide some simple evaluation measures. For example, firefighters have particular protocols for handling fire hoses [34]. Depending on the number of firefighters available for handling the hose, they should take different positions and perform slightly different tasks. For two firefighters, one firefighter anchors the hose to stabilize it while the other operates the nozzle. For three, one firefighter operates the nozzle, at least one firefighter will anchor the hose, and the third will either anchor or help support the hose. Other methods can be used depending on how mobile the team needs to be.

One way that trainees could be tested in a simulation is to use a scenario with one or more AI characters filling roles as firefighters. The protocol can be adapted for multi-character affordances by providing a number of affordance slots at the different operational positions along the hose. The trainee should observe the number of AI characters present, as well as the locations they take. Then they must choose the correct position and action to execute to handle the hose. The system can easily give automated feedback based on the affordance chosen by the trainee. Generating new scenarios to test with in this simulation can then be automated. Multi-character affordances can be created which use context to provide the appropriate actions to the AI; the context tests can then also be used to evaluate the appropriateness of the trainee's choice. In general, multi-character or even single-character contextual

affordances can assist in training simulations to automatically evaluate the trainee's performance on protocols with minimal additional testing logic beyond the contextual tests for the affordances.

For simulations where the overall system behavior over time is the focus, reactive teaming may be more effective. This may be important in agent-based modeling or other systems using populations of agents to simulate a large-scale system. Reactive teaming can be used, allowing agents to propagate behaviors through the system, and evaluation of the system state can consider the behavior distribution and rate of propagation.

6.4 Future Work

One unexpected result of the evaluation of extended affordance methods is the power of randomness to generate reasonable behavior. One current research topic in game AI is that of *alibi generation*, which focuses on making the random crowds of characters in an open world seem more purposeful and less random. In small environments, contextual affordances can provide a high level of apparent purpose to characters making purely random decisions; this naturally leads to a question about larger environments. The application of contextual, multi-character, and probabilistic affordances to large open-environment games is a natural extension to this work.

We know from the BehaviorShop studies that non-programmers and AI novices can use subsumption to build intelligent characters that act alone. BehaviorShop used an early version of our BEHAVEngine architecture. The early version was a decoupled architecture with some failure detection capabilities, but did not make use of dynamic priorities or teaming techniques. An important question to answer with

these extensions is whether they can be integrated effectively with BehaviorShop. Dynamic priorities add an additional level of complexity to building behaviors which may not be as accessible to novice users, and understanding how to apply reactive teaming effectively to a problem may prove non-intuitive. Integration of affordances with BehaviorShop would also be an interesting direction to explore, but would not require significant modification to BehaviorShop.

REFERENCES

[1] Anderson, J. R. Act: A simple theory of complex cognition. *American Psychologist 51* (1996), 355–365.

[2] Artificial Technology. Eki One. http://www.eikone.com, 2009.

[3] Booth, M. The AI Systems of Left 4 Dead . Keynote at the 5th Artifcial Intelligence for Interactive Digital Entertainment Conference, October 2009. Slides retrieved from http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf.

[4] Borovikov, I. *AI Game Programming Wisdom 3*. Charles River Media, 2006, ch. 4.3: Orwellian State Machines, pp. 275–288.

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

[6] Brooks, R. A. Elephants don't play chess. *Robotics and Autonomous Systems 6*, 1&2 (June 1990), 3–15.

[7] Bruce, J., Zickler, S., Licitra, M., and Veloso, M. M. Cmdragons: Dynamic passing and strategy on a champion robot soccer team. In *International Conference on Robotics and Automation* (2008), pp. 4074–4079.

[8] Brumitt, B. L., and Stentz, A. Dynamic mission planning for multiple mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation* (1996), pp. 2396–2401.

[9] Calabi, L., and Hartnett, W. E. Shape recognition, prarie fires, convex deficiencies and skeletons. *The American Mathematical Monthly 75*, 4 (1968), 335–342.

[10] Carnegie Mellon University Graphics Lab. Motion Capture Database. http://mocap.cs.cmu.edu/.

[11] Cerpa, D. H., Champandard, A., and Dawe, M. Behavior Trees: Three Ways of Cultivating Strong AI. Talk at the 2010 AI Summit at the Game Developers Conference (GDC), http://gdcvault.com/play/1012744/Behavior-Trees-Three-Ways-of, 2010.

[12] Champandard, A. J. What Does a Behavior Tree Editor Look Like? http://aigamedev.com/open/articles/behavior-tree-editor-example/, 2008. January 21,2010.

[13] Cohen, M. A., Ritter, F. E., and Haynes, S. R. Herbal: A high-level language and development environment for developing cognitive models in Soar. In *Proceedings of the 14th Conference on Behavior Representation in Modeling and Simulation (BRIMS)* (2005), pp. 177–182.

[14] The Computer is a Cheating Bastard. http://tvtropes.org/pmwiki/pmwiki.php/Main/TheComputerIsACheatingBastard.

[15] Cote, C., Letourneau, D., Michaud, F., Valin, J.-M., Brosseau, Y., Raievsky, C., Lemay, M., and Tran, V. Code reusability tools for programming mobile robots. *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on 2* (Sept.– Oct. 2004), 1820–1825 vol.2.

[16] Crytek. Cryengine 3. http://mycryengine.com/, 2010. January 21, 2010.

[17] Dahl, T. S., Matarić, M., and Sukhatme, G. S. Multi-robot task allocation through vacancy chain scheduling. *Robot. Auton. Syst. 57*, 6-7 (2009), 674–687.

[18] DARWARS Ambush! DARPA Project. http://ambush.darwars.net/.

[19] Dias, M. B., and Stentz, A. A market approach to multirobot coordination. Tech. Rep. CMU-RI-TR-01-26, CMU Robotics Institute, Pittsburgh, PA, August 2001.

[20] Erol, K., Hendler, J., and Nau, D. S. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)* (Seattle, Washington, USA, 1994), vol. 2, AAAI Press/MIT Press, pp. 1123–1128.

[21] Eyk, D. Owyl. https://github.com/eykd/owyl/.

[22] Fikes, R. E., and Nilsson, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2*, 3-4 (1971), 189–208.

[23] Firby, R. J. An Investigation into Reactive Planning in Complex Domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI)* (1987).

[24] Fu, D., and Houlette, R. *AI Game Programming Wisdom 2*. Charles River Media, 2004, ch. 5.1: The Ultimate Guid to FSMs in Games, pp. 283–302.

[25] Gerkey, B. P., and Matarić, M. J. Sold!: Auction methods for multi-robot coordination. *IEEE Transactions on Robotics and Automation 18*, 5 (oct 2002), 758–768.

[26] Gibson, J. J. *Perceiving, Acting, and Knowing*. Lawrence Erlbaum Associates, 1977, ch. The theory of affordances.

[27] Hale, D. H., Youngblood, G. M., and Dixit, P. Automatically-generated Convex Region Decomposition for Real-time Spatial Agent Navigation in Virtual Worlds. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)* (2008).

[28] Havok. Havok Behavior. http://www.havok.com/index.php?page=havok-behavior, 2010. January 21,2010.

[29] Heckel, F. W. P., Youngblood, G. M., and Hale, D. H. BehaviorShop: An Intuitive Interface for Interactive Character Design. In *Proceedings, 5th Artificial Intelligence for Interactive Digital Entertainment (AIIDE 2009)* (2009).

[30] Heckel, F. W. P., Youngblood, G. M., and Hale, D. H. Influence Points for Tactical Information in Navigation Meshes. In *Proceedings, International Conference on Foundations of Digital Games* (2009).

[31] Heckel, F. W. P., Youngblood, G. M., and Hale, D. H. Making Interactive Characters BEHAVE. In *Proceedings, Florida Artificial Intelligence Research Symposium* (2009).

[32] Hertel, S., and Mehlhorn, K. Fast Triangulation of the Plane with Respect to Simple Polygons. In *International Conference on Foundations of Computation Theory* (1983).

[33] Insomniac Games. Ratchet and Clank. http://www.ratchetandclank.com/. Retrieved April 16, 2011.

[34] International Fire Service Training Association. *Essentials of Fire Fighting*. Oklahoma State University, 1998.

[35] Isla, D. Handling Complexity in the Halo 2 AI. In *Proceedings of the 2005 Game Developers Conference* (2005).

[36] Jennings, N. R. Commitments and conventions: The foundation of coordination in multi-agent systems. *The Knowledge Engineering Review 8*, 03 (1993), 223–250.

[37] Kalra, N., Dias, M. B., Zlot, R. M., and Stentz, A. T. Market-based multirobot coordination: A comprehensive survey and analysis. Tech. Rep. CMU-RI-TR-05-16, Robotics Institute, Pittsburgh, PA, December 2005.

[38] Khoo, A. *AI Game Programming Wisdom 3*. Charles River Media, 2006, ch. 4.10: An Introduction to Behavior-Based Systems for Games, pp. 351–364.

[39] King, W. Virtual Battle Space 2 Army gaming system debuts. Article from http://www.army.mil, February 2009.

[40] Kose, H., Kaplan, K., Mericlu, C., Tatlidede, U., and Akin, L. Market-driven multi-agent collaboration in robot soccer domain. *Cutting Edge Robotics* (2005).

[41] Laird, J., Rosenbloom, P., and Newell, A. Soar: An architecture for general intelligence. *Artificial Intelligence 33* (1987), 1–64.

[42] Lenser, S., Bruce, J., and Veloso, M. A modular hierarchical behavior-based architecture. In *RoboCup 2001: Robot Soccer World Cup V*, A. Birk, S. Coradeschi, and S. Tadokoro, Eds., vol. 2377 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 79–99.

[43] Lesser, V. R. Reflections on the Nature of Multi-Agent Coordination and Its Implications for an Agent Architecture. *Autonomous Agents and Multi-Agent Systems 1*, 1 (1998), 89–111.

[44] Loew, H., and Hinkle, C. *AI Game Programming Wisdom 4*. Charles River Media, 2008, ch. 5.5: Enabling Actions of Opportunity with a Light-Weight Subsumption Architecture, pp. 493–497.

[45] Loyall, A. B., and Bates, J. Hap: A Reactive, Adaptive Architecture for Agents. Tech. rep., Carnegie Mellon University, June 1991.

[46] Lussier, B., Chatila, R., Ingrand, F., Killijian, M.-O., and Powell, D. On Fault Tolerance and Robustness in Autonomous Systems. In *Proceedings of the 3rd IARP - IEEE/RAS - Euron Joint Workshop on Technical Challenges for Dependable Robots in Human Environmnets* (2004).

[47] Mark, D. Does This Mistake Make Your AI Look Smarter? http://aigamedev.com/open/articles/mistakes-look-smart/, 2007.

[48] Matarić, M. J. Behavior-based control: Main properties and implications. In *Proceedings, IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems* (1992), pp. 46–54.

[49] Matarić, M. J., Nilsson, M., and Simsarian, K. T. Cooperative multi-robot box-pushing. In *IROS '95: Proceedings of the International Conference on Intelligent Robots and Systems-Volume 3* (Washington, DC, USA, 1995), IEEE Computer Society, p. 3556.

[50] Mateas, M., and Stern, A. A behavior language for story-based believable agents. *IEEE Intelligent Systems 17*, 4 (2002), 39–47.

[51] McGonigal, J. *Reality is Broken: Why Games Make Us Better and How They Can Change the World*. Penguin Press HC, January 2011.

[52] Miles, C., and Louis, S. J. Towards the co-evolution of influence map tree based strategy game players. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games* (2006).

[53] Molineaux, M., Klenk, M., and Aha, D. W. Goal-driven autonomy in a navy strategy simulation. In *In Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)* (2010).

[54] Morgan, G. P., Haynes, S. R., and Ritter, F. E. Increasing efficiency of the development of user models. In *Proceedings of the IEEE System Information and Engineering Design Symposium* (2005).

[55] Nakashima, H., and Noda, I. Dynamic subsumption architecture for programming intelligent agents. In *Proceedings of the 3rd International Conference on Multi Agent Systems* (Washington, DC, USA, 1998), ICMAS '98, IEEE Computer Society, pp. 190–.

[56] Nareyek, A. AI in Computer Games. *ACM Queue* (February 2004), 59–65.

[57] Norman, D. A. *The Design of Everyday Things*. Basic Books, September 2002.

[58] The Oxford English Dictionary Online. http://dictionary.oed.com/.

[59] Orkin, J. *AI Game Programming Wisdom 2*. Charles River Media, 2004, ch. 3.4: Applying Goal-Oriented Action Planning to Games, pp. 199–206.

[60] Orkin, J. *AI Game Programming Wisdom 2*. Charles River Media, 2004, ch. 3.2: Simple Techniques for Coordinated Behavior, pp. 217–227.

[61] Orkin, J. Three States and a Plan: The AI of F.E.A.R. In *Proceedings of the Game Developer's Conference (GDC)* (2006).

[62] Pittman, D. *AI Game Programming Wisdom 4*. Charles River Media, 2008, ch. 4.3: Command Hierarchies Using Goal-Oriented Action Planning, pp. 383–391.

[63] Pottinger, D. C. Terrain Analysis in Realtime Strategy Games. In *In Proceedings of Computer Game Developers Conference* (2000).

[64] Realmware Corporation. Visual3d game engine. http://game-engine.visual3d.net/, 2010. January 21,2010.

[65] Ros, R., Veloso, M., de Màntaras, R. L., Sierra, C., and Arcos, J. L. Retrieving and reusing game plays for robot soccer. In *Lecture Notes in Computer Science, Proceedings of ECCBR-2006* (Oludeniz, Turkey, September 2006).

[66] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*, second ed. Pearson Education, Inc, 2003, ch. Part 4: Planning & Part 5 : Uncertain knowledge and reasoning.

[67] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*, second ed. Pearson Education, Inc, 2003, ch. Chapter 12: Planning and Acting in the Real World, pp. 417–461.

[68] Schneider-Fontán, M., and Matarić, M. J. Territorial multi-robot task division. *IEEE Transactions on Robotics and Automation 14* (1998), 815–822.

[69] Schwab, B. Implementation Walkthrough of a Homegrown "Abtract State Machine" Style System in a Commercial Sports Game. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference* (2008), pp. 145–148.

[70] Shell, D. A., and Matarić, M. J. *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation.* Cambridge University Press, 2005, ch. Behavior-Based Methods for Modeling and Structuring Control of Social Robots, pp. 279–306.

[71] Simpson, J. Scripting and Sims2: Coding the Psychology of Little People. Talk at the 2005 Game Developers Conference (GDC), https://www.cmpevents.com/Sessions/GD/ScriptingAndSims2.ppt, 2005.

[72] Stottler Henke Associates. Simbionic. http://www.simbionic.com/, 2004. Retrieved November 23, 2008.

[73] Straatman, R., Verweij, T., and Champandard, A. Killzone 2 multiplayer bots. Slides from http://files.aigamedev.com/coverage/GAIC09_Killzone2Bots_StraatmanChampandard.pdf, June 2009. Retrieved May 4, 2010.

[74] Suits, B. *The Grasshopper: Games, Life and Utopia.* Broadview Press, 2005.

[75] Tozour, P. *Game Programming Gems 2.* Charles River Media, 2001, ch. 3.6: Influence Mapping, pp. 281–297.

[76] Tozour, P. *AI Game Programming Wisdom 2.* Charles River Media, 2004, ch. 2.1 Search Space Representations, pp. 85–102.

[77] Valve. Source multiplayer networking. Valve developer documentation. Retrieved from http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.

[78] Valve. Left 4 Dead. PC Game, 2008.

[79] Valve. Valve Press Release. Web press release, September 2009. Retrieved from http://store.steampowered.com/news/2858/.

[80] van der Sterran, W. *AI Game Programming Wisdom.* Charles River Media, 2002, ch. 5.3: Squad Tactics: Team AI and Emergent Maneuvers, pp. 233–246.

[81] van der Sterran, W. *AI Game Programming Wisdom.* Charles River Media, 2002, ch. 5.4: Squad Tactics: Planned Maneuvers, pp. 247–259.

[82] Visual Soar. Software package. http://www.eecs.umich.edu/~soar/sitemaker/projects/visualsoar/.

[83] Watanabe, M., Onoguchi, K., Kweon, I., and Kuno, Y. Architecture of Behavior-based Mobile Robot in Dynamic Environment. In *Proceedings of the IEEE International Conference on Robotics and Automation* (1992), ICRA '92, pp. 2711 – 2718.

[84] Wendler, J., Gugenberger, P., and Lenz, M. Cbr for dynamic situation assessment in an agent-oriented setting. In *In Proceedings of AAAI-98 Workshop on Case Based Reasoning Integrations* (1998).

[85] Werger, B. B. Cooperation without deliberation: A minimal behavior-based approach to multi-robot teams. *Artificial Intelligence 110* (1998), 293–320.

[86] Xaitment. xaitMove and xaitKnow. http://www.xaitment.com, 2009.

[87] Yiskis, E. *AI Game Programming Wisdom 2.* Charles River Media, 2004, ch. 6.1: A Subsumption Architecture for Character-Based Games, pp. 329–337.

[88] Zang, P., Mehta, M., Mateas, M., and Ram, A. Towards runtime behavior adaptation for embodied characters. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence (IJCAI 2007)* (2007), pp. 1557–1562.

APPENDIX A: BEHAVIORSHOP

Behavior Editors

Behavior authoring tools have been developed both in academic research and in industry, and have been recognized as an important step towards improving artificial intelligence in video games during sessions at the 2008 Game Developers Conference, the 2009 Artificial Intelligence for Interactive Digital Entertainment Conference, and other events. A variety of tools exist from robotics, artificial intelligence, and the game industry. The RobotFlow interface from the University of Sherbrooke works with their MARIE framework to build complete robot software systems, but is more low-level than desired for game tools [15].

Artifical Technology, Xaitment, Havok, Presagis, and SimBionic have all produced commercial tools that allow the graphical creation of game agents [2, 86, 28, 72]. These tools are targeted at professional AI and character developers in industry, and are not generally appropriate for novice users. The commercial tools are included with full AI middleware and runtime systems, and are not intended to work with other AI middleware. Crytek includes the Sandbox editors with copies of games using their engine [16].While Sandbox is available to end-users, it has a difficult learning curve. The Sandbox AI editor is FSM-based, but the terminology used in creating individual states is nonintuitive. All of these editors focus on providing authoring environments for finite state machine controlled and scripted agents. Various game studios have developed their own editors for internal use, such as Sony's Situation editor, developed by Brian Schwab for us in sports games [69]. A behavior tree editor has been developed

for an upcoming release of the Visual3D Game Engine [64]. Additional behavior tree editors have been developed by the community [12].

Evaluation of usability for behavior editors is treated as a low priority by the middleware companies, and to our knowledge, the only published user evaluation of a graphical AI editor has been performed with UNC Charlotte's BehaviorShop, as seen in [29]. BehaviorShop is a subsumption architecture-based editor targeted at a non-expert user base.

The Herbal IDE is another type of agent editor. It focuses on producing agents in the Herbal high level language that are then compiled into cognitive models [13]. The current version of Herbal is an IDE built on top of the Eclipse environment, and provides functionality to assist with the creation of models. It does not currently have a graphical component. Herbal itself does not provide an architecture for executing agents, but instead provides compilers for translating the Herbal agents into Soar, Jess, and ACT-R. Evaluation of the IDE showed that the use of Herbal increased the speed at which users could create cognitive models [54]. The Visual Soar environment also provides an IDE interface for building cognitive models, but is specific to the Soar cognitive model [82].

DASSIEs

The DASSIEs (Dynamic Adaptable Super-Scalable Intelligent Entities) Project addresses the problem of building an intuitive user interface for interactive character design (see Figure A.1). DASSIEs is composed of three major parts: CGUL (Common Games Understanding and Learning Toolkit), BEHAVE (Behavior Emulating Hierarchically-based Agent Vending Engine), and BehaviorShop [31]. Our game en-

vironment is FI3RST (FIrst and 3rd-person Realtime Simulation Testbed).

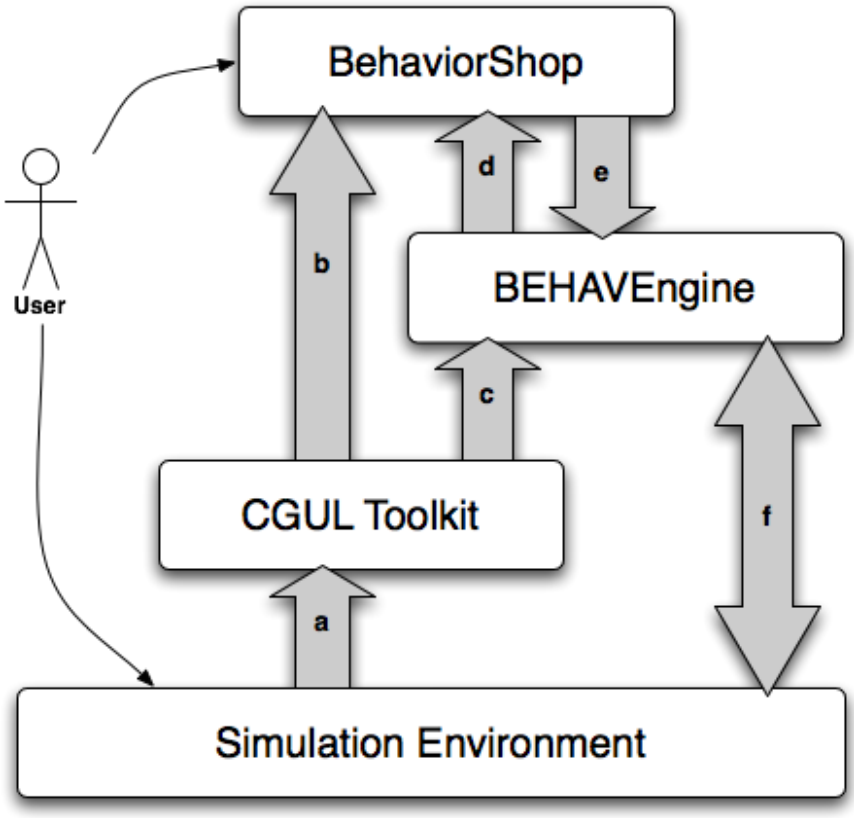


Figure A.1: The DASSIEs system: Information flow is labeled with the large arrows showing a) the flow of the environment information into CGUL for processing into knowledge, b) the flow of processed information and created knowledge for incorporation into the behavior construction task, c) the flow of processed information and created knowledge to BEHAVEngine for use in reasoning and agent interaction, d) the flow of agent action information to BehaviorShop for incorporation into the behavior construction task, e) the flow of the behavioral model to BEHAVEngine for creation of the simulation agent(s), and f) the control and perception information interchange between the BEHAVEngine and the Simulation Environment.

CGUL provides information services to BehaviorShop and BEHAVE, such as navigation services (world geometry and planning) and information tagging through influence points [30]. Game events flow to CGUL, which updates its services, and passes processed information on to BEHAVE and BehaviorShop. BEHAVE uses CGUL during runtime, while BehaviorShop polls CGUL during agent creation to provide

accurate information about the simulation environment to the user. BehaviorShop also queries BEHAVE for information about available pre-built character behaviors.

BehaviorShop and BEHAVE are based on the subsumption architecture, as we have found that AI-naive users find the layered architecture more intuitive than FSMs when creating agents.

BehaviorShop

BehaviorShop is a subsumption-based interactive character builder designed to be utilized by users with little to no AI experience. It is capable of querying game services and the underlying AI engine to receive information appropriate to the given scenario. This helps the user ground the character being created in the game environment for a given scenario.

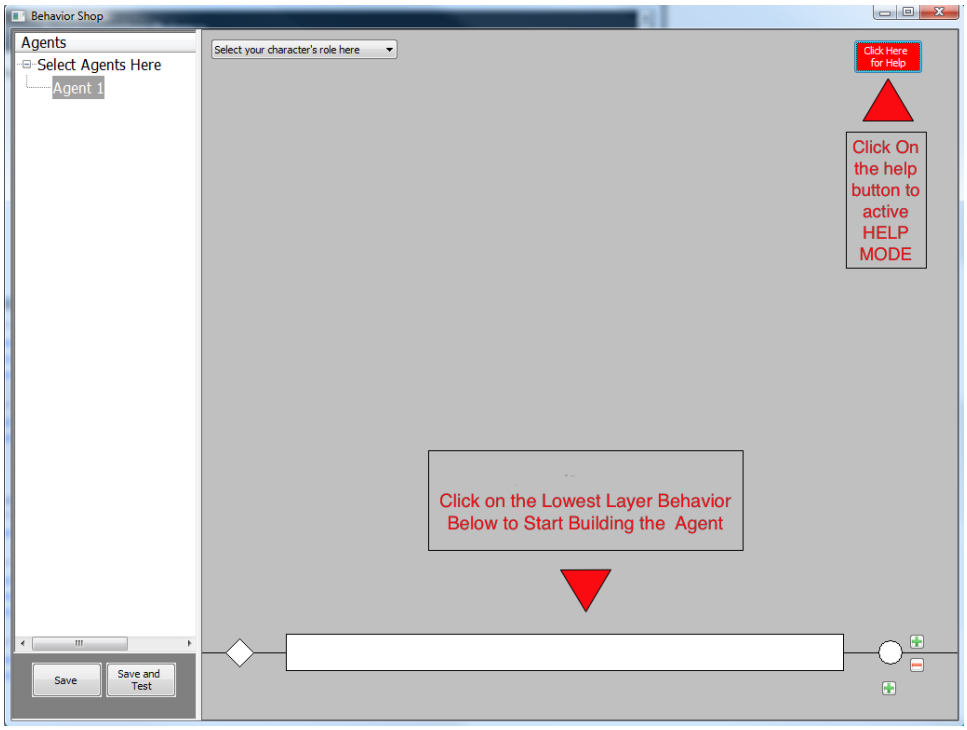


Figure A.2: The main subsumption view in BehaviorShop.

When a user initially starts BehaviorShop, they are taken to the screen shown in

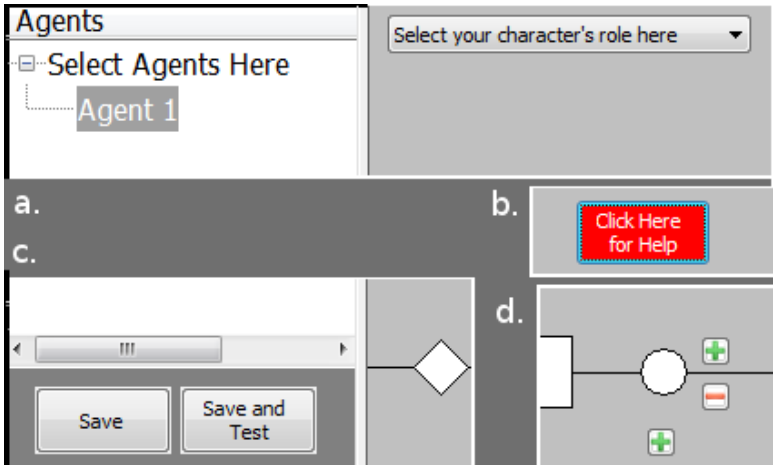


Figure A.3: Details of main view: a) The character role selector, b) Help activation button, c) Save buttons, d) Layer creation controls.

Figure A.2 and detailed in Figure A.3. The left pane provides an area where the user can view the currently open characters. The user first selects the role of the agent to be created(5a). The roles provide predefined information for the agent, such as the model to be used, starting location, and default parameter settings. The help button (5b) provides a tool tip-like facility for obtaining information about each area of the builder, and at the bottom of the pane the first layer is available. When the user clicks on this layer, he is taken to the behavior screen. The + and – buttons (5d) allow the user to add or remove layers from the subsumption. Finally, the user can save the current agent to a file, or save it and view a preview of the agent's behavior in the game engine (5c).

The first layer of the behavior is preset to use an "Always" trigger—this is the base layer behavior, and one must always be present. Figure A.4 shows the behavior screen for this layer, and demonstrates the flow of information from the information services into BehaviorShop. For the behavior "Go To Point", BehaviorShop provides a map of the game environment so that the user can select the target location in the

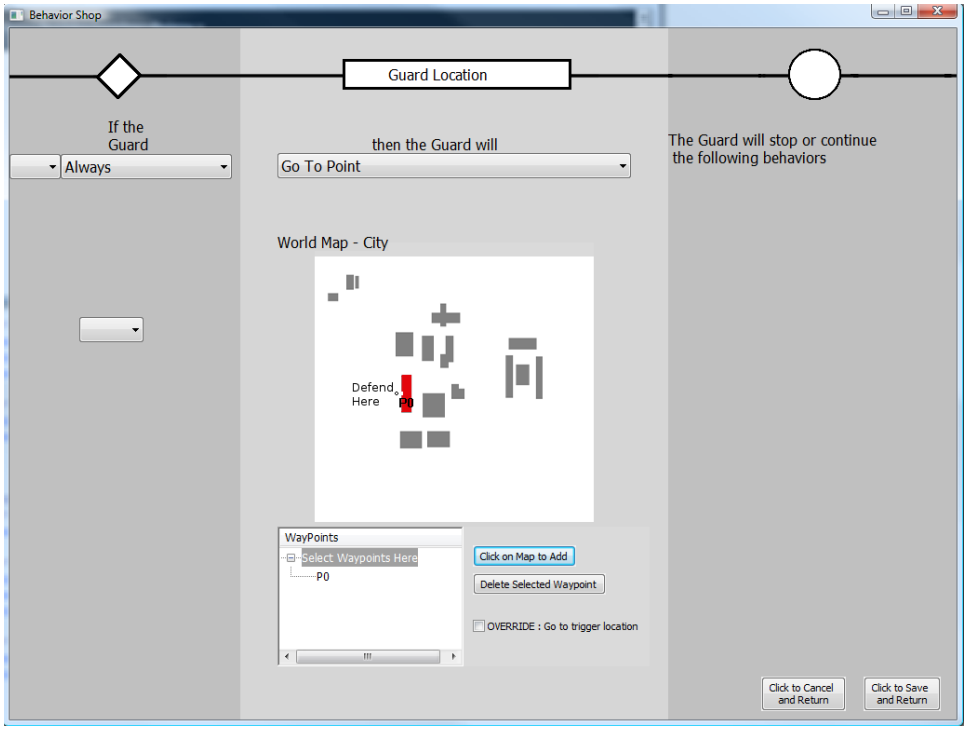


Figure A.4: The BehaviorShop specific behavior builder screen. This screen allows the user to define a specific layer in the subsumption agent. Note the map of the environment which is received from CGUL, and the menus which are propagated by querying BEHAVEngine.

world.

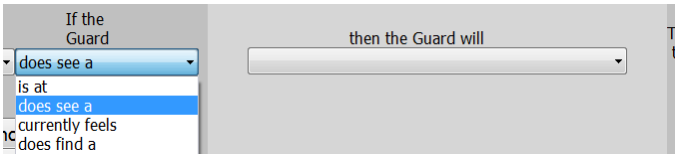


Figure A.5: behavior layers are specified as English language sentences

Note that, as seen in Figure A.5, behaviors are created as simple sentences: *if the [role] [triggering condition], then the [role] will [behavior response].* We have found that providing this style of interface makes the interface far more intuitive than simple lists of trigger and behavior names.

In addition to basic *if/then* sentences, trigger conditions can be modified through negation as well as combined as conjunctions or disjunctions. Figure A.6 shows the

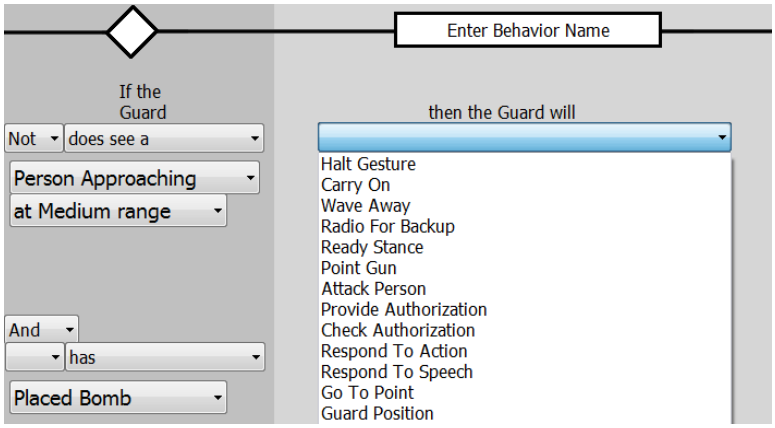


Figure A.6: Triggers can be negated using the "not" dropdown and combined with the "and" (conjunction/disjunction) dropdown. Behaviors available to the user are displayed using a drop down list.

trigger options as well as the list of behaviors. These behaviors are obtained by querying the BEHAVEngine. This allows BehaviorShop to provide access only to the behaviors available to the user in the particular scenario.

After creating several subsumption layers for a character, such as that seen in Figure A.7, the user can test the character using the *Save and Test* button (which can be used at any time during creation), or if satisfied, can save the complete character with the *Save* button to finish the task.

The specified subsumption is translated to a simple behavior description language which is understood by BEHAVE. This language lists all of the behaviors for the character, coupled with one or more triggers per behavior.

Evaluation

During the initial development of BehaviorShop, we created several different versions to explore different types of agent architectures. The initial interface was a FSM builder, but most users had difficulty with this due to the complexity of the transitions. A HFSM version also proved unintuitive, and so FSM-based approaches

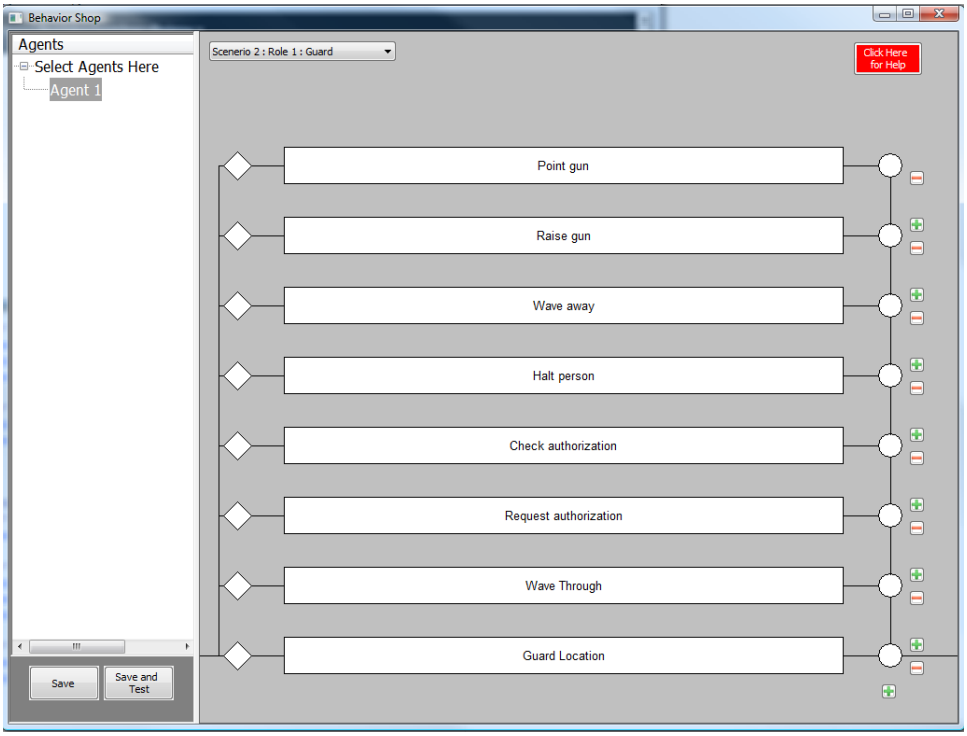


Figure A.7: The BehaviorShop subsumption screen. This shows the full layered behaviors for a guard character.

were quickly abandoned. Surprisingly, the layered builder was immediately found more intuitive, and most users were able to use it more easily.

During a pilot study at a public library, an earlier subsumption based version of BehaviorShop proved successful. Fifteen subjects participated in this pilot. Each subject was given the task of creating a game character for one of five possible described roles, which involved some complex behaviors (e.g., security for a location, gathering information from other characters, and other military-type scenarios). A single subject quit in frustration, citing basic unfamiliarity with computers. Eleven subjects created characters that behaved correctly and accomplished all key behaviors for the assigned role (as compared with characters created by experts from our lab), and three others created partially correct characters. The pool of subjects was 53%

female and covered an age range from just over 18 to mid-50s.

Online Study

Before running full user studies with BehaviorShop, we first created an online study. In the online study, subjects were asked to write instructions for a person filling a specific role in a scenario. One of thirteen possible roles in five different scenarios was chosen, and a high-level description of the role was given. The subject was then asked to write a more precise description of the tasks for a person executing that role. After completing this portion, the subject was shown a video of actors demonstrating another scenario. Following the video, the subject was asked to write a precise description of how one of the actors performed their role.

Descriptions from this study were analyzed for word frequency to find the most common language used to describe behaviors for each role. The results of frequency analysis based on 147 people are being used to adjust the presentation of behaviors and triggers for the interface, but the current results are preliminary. Once we have completed the analysis, it will be used to develop a more complete vocabulary of behaviors and language specific to user profiles for use with BehaviorShop and BEHAVEngine.

User Study

Our first full user study with the complete system (BehaviorShop, BEHAVEngine, and FI3RST) included fourteen subjects, but data from four subjects was unusable. Each subject was asked to build a character filling a guard role. Users were given the following description of the scenario and guard role:

> This is a normal day outside a secure office building. A security guard (Role 1) armed with a gun and a communication radio to the main guard dispatch office stands guard at the entrance to the facility. Authorized personnel must show their badges to the guard to gain entry. An authorized person (Role 2) will enter the building through the doors after showing their badge to the security guard. An unauthorized person (Role 3) is trying to make several attempts to enter the building. They try to run past the guard, trail an authorized person, and other such activity. The unauthorized person does carry a knife. The security guard is authorized to use deadly force if threatened with any type of lethal force.

After being given the description and a 1-minute tool and subsumption technique tutorial, the user was asked to create the guard agent in BehaviorShop.

Before the study, subjects were given a demographic survey. The majority of subjects had substantial experience playing video games, and 7 were male. All of the subjects were in the 18-24 age range. Of the subjects, 6 had no AI experience, while 3 more described themselves as having had a low level of experience; 6 of 10 subjects had either a bachelor's degree or were currently in a bachelor's program, 3 had advanced degrees, and 1 high school education.

Results

Of the 10 users with valid data, 8 successfully created agents that filled the guard role. To determine success, the characters generated by the users were compared with a set of reference agents created by members of the research team. 3 users created

agents which exactly matched one of the reference agents, while 5 more created agents which were acceptable—they did not match exactly, but used all of the critical behaviors as defined by the AI experts. These measures were also verified by calculating a graph edit distance between the reference agents and created agents. Subsumption controllers were transformed into a graph where each layer of the subsumption (both trigger and behavior together) was treated as a vertex in the graph, and an edge was created between adjacent layers (ie, the third layer had edges to the second and fourth layers). If the behavior and trigger of a layer matched the reference character, no change was made (distance of zero). If only one of the behavior or trigger was correct, it was considered a distance of 1. If the entire vertex had to be removed and replaced, that counted as a distance of 2. The average graph edit distance for successful agents was 1.25, while the average distance for unsuccessful agents was 7.25. The difference between the two groups was found to be statistically significant at $p < .05$.

We learned several important lessons from this user study. First, contrary to our expectations, the most confusing aspect of the user interface was the negation modifier for the trigger conditions, rather than the conjunction/disjunction feature. No users used the help button, suggesting that a new paradigm for providing support information is needed; this is especially important given that users were sometimes confused by what different triggers and behaviors actually do. Four users attempted to click and drag layers of the subsumption around to rearrange their priorities. Two users actually requested the ability to make more complex behaviors, by linking multiple behaviors to a single trigger, either in series or parallel.

One of the most important discoveries based on user feedback was that no users had difficulty with the basic subsumption concept of triggers and actions. Only brief instruction was given about the subsumption architecture itself, yet it was still a very intuitive paradigm. The *Save and Test* button was extremely important, as users would frequently test out behaviors to see what they would do. This reinforces the notion that some mechanism for providing immediate feedback is needed.

Certain triggers and behaviors were confusing to users. BEHAVE includes a simple mental state model for characters, allowing them to be *scared*, *alert*, or *nervous*. These were chosen for specific scenarios, but, unsurprisingly, they caused confusion, as it is difficult to precisely define what these moods mean. Users had different expectations than the developers about what events would trigger a change in these states. For our guard scenario, initially the character authorization required two steps: *request authorization* and *confirm authorization*. Consolidating these two behaviors into a single more complex behavior greatly reduced the problem.

In the post-survey, subjects were also asked to rate two statements on a 5 point Likert scale: "Creating simulation characters is easy with the DASSIEs Creation Tool" and "Making simulation characters is fun". The average for ease of use was 3.38, with a standard deviation of 0.916, while fun was rated higher—average of 4.5, with a standard deviation of 0.756. While users found creating the characters moderately difficult, no user rated fun lower than a neutral "neither" value, with most users rating fun as "agree" or "strongly agree".

APPENDIX B: BEHAVENGINE

BEHAVEngine (Behavior Emulating Hierarchical Agent Vending Engine) is a full C++-based AI engine for hierarchical subsumption architecture with reactive teaming. This appendix provides an overview of the individual components of BEHAVEngine and basic usage information.

BEHAVEngine Usage

BEHAVEngine requires FI3RST to run. For information on running FI3RST, see the FI3RSTManual, distributed along with FI3RST. Once FI3RST and MemStore are running, BEHAVEngine can be started from the command line by specifying an agent configuration file. All of BEHAVEngine's configuration files should be stored under `irrFI3RST/environments/<environment-name>/`. Characters must be stored in the `characters` directory under the given environment.

BEHAVEngine is started with the command:

```
>BEHAVEngine.exe -c <config>
```

Where $< config >$ is the name of a configuration file stored under the environment currently being run in FI3RST. Other command line options are also available:

```
BEHAVEngine version 1.5


Usage:
  -h [ --help ]           Display this help message
  -c [ --config ] arg     Set BEHAVE config file (from the directory
```

```
 containing the environment)
-q [ --query ] arg      Run in query mode (dumps behavior and
 trigger information to file)
-t [ --time ] arg       Use time multiplier value to slow down or
 speed up BEHAVE
-o [ --outputtree ] arg Dump the trigger output tree to a file.
-l [ --logtofile ]      Send logs to file
-m [ --mute ]           Mute stderr output
-x [ --logdata ]        Enable data logging.
-v [ --verbose ] arg    Set output level: debug, info, notice,
                        warning, error, fatal.
```

BEHAVEngine Structure

BEHAVEngine is divided into three major parts: BEHAVEngine, BEHAVEControl, and SubsumptionBehaviorLibrary. BEHAVEngine is the base application runtime. It handles loading agents, communicating with FI3RST, managing the thread pool, and serves as a runtime container for all agents. Base classes for behaviors, triggers, perceptual models, and action models are included in the main runtime.

BEHAVEControl is a command line utility for controlling BEHAVEngine. It can send commands to check the status, pause, unpause, stop, or reset BEHAVEngine.

The SubsumptionBehaviorLibary contains the code necessary to run hiearchical subsumption agents. This library defines 19 triggers and 31 behaviors, along with the subsumption-specific action and perception models.

Additional libraries can be added to BEHAVEngine to define additional triggers and behaviors or entirely new architectures.

BEHAVEngine Configuration Files

BEHAVEngine uses multiple configuration files. It provides information about agent capabilities to through the use of BehaviorInfo files and receives lists of agents to instantiate from the BEHAVEngine configuration file. Agent specifications are read from agent description files. When using FI3RST, information about the scenario is received from the FI3RST scenario description file, which is largely transferred via shared memory. Object and agent information is received from the FI3RST object info and model info files. The FI3RST files are not described here, but the two BEHAVEngine file types are described in the following sections.

BEHAVEngine Capabilities

Because components of BEHAVEngine can be exchanged to create different types of controllers, and because it is designed to work with a variety of simulations, its capabilities may be different depending on how it is configured. It is critical to specify the behaviors and triggers available to BEHAVEngine, so this is done with the BehaviorInfo query file. Each behavior and trigger must include a method that provides information about its inputs and outputs.

Normally, the capability file should never be used directly. BehaviorShop provides an additional utility, the LanguageBuilder, which consumes the capability file and can be used to create a language file as input to BehaviorShop.

This information is produced according to a grammar:

behavior *name* **modality** { **visual** | **aural** | **self** | **historical** | **other** | **none** }

option* *option name* of **semantictype** { **agent** | **object** | **location** | **region** | **distance** | **regiondistance** | **boolean** | **relation** | **state** | **user** }

syntactictype { **string** | **numeric** | **point** | **boolean** }

multiplicity {**nomin** | **min** *value* } {**nomax** | **max** *value* }

with **limits**

rangemin? *value*

rangemax? *value*

source? { **any** | **trigger** | **auto** | **description** }

inventory? {**true** | **false** }

integer?

relation* { **hostile** | **neutral** | **friendly** }

action* *value* **slot** { **object** | **tool** | **target** }

enum* *value*

conflict*

option name

option name

produces* action *actiontype*

requires* *option name* **value** *value*

Keywords are in **bold**, and user-specified values are in *italics*. The * symbol indicates that zero or more instances of this option may be present, a ? symbol indicates that zero or one instances of the option may be present.

- **modality** gives a hint as to the type of behavior. This is useful for categorizing behaviors in an interface as well as creating natural language sentence representations of the behavior.

- Multiple **options** may be present for a behavior.

- **semantictype** describes the content of this option. If the option may be received via a trigger, this specifies the percept type. Note that the **boolean**, **distance**, and **user** types do not unambiguously describe a percept type. Also, a **location** value may be extracted from any location-dependent percept (so this could be a location, agent, or object percept if received from a trigger). **region** values can be numeric (particular region) or a string (symbolic region name).

- **syntatictype** describes how the option will be written in a description file.

- **multiplicity** describes how many instances of the option may be present.

- **limits** provides range and attribute limits for the option.

- **rangemin** allows a minimum value for numeric option types (such as distance).

- **rangemax** allows a maximum value for numeric option types

- **source** restricts how option values can be received by the behavior. The default is **any**. **trigger** means that the option must be received as a percept from the trigger which activates the behavior. **auto** means that the behavior may receive it as a percept from the trigger, or search the percept stream for an appopriate value. **description** means that this option can only be specified in the agent description file.

- **inventory** is useful if the option is an **object** semantic type, by declaring that the object must or must not be in the inventory.

- **integer** forces a numeric option to be an integer value

- **relation** specifies the relationship between this agent and an **agent** semantic type option.

- **action** specifies that an **object** semantic type option must be filled by an object which can be acted upon with the given action, as either the target of the action, an object involved in the action, or a tool to execute the action. This must match with the object description from FI3RST.

- **enum** is used with user/string types to provide a list of possible options

- **conflict** clauses specify that two options conflict with one another

- **produces** clauses specify the action produced by a behavior

- **action** gives the name of the action produced, which must match an action from FI3RST (for the agent to instantiate correctly, this action must be available in its action model)

- **requires** specifies an option which must be present for this action to be produced. The optional **value** gives a required value for the option.

Note that unless specifically marked as a conflict, any combination of options is allowable. Note also that *produces* clauses specify only the minimum option requirements for a particular action (or with triggers, perception) to be produced– other options may be present, so long as they do not conflict with the required options.

Trigger specification is very similar:

trigger *name* **modality** {**visual** | **aural** | **self** | **historical** | **other** }

option* *option name* of **semantictype** { **agent** | **object** | **location** | **region** | **distance** | **regiondistance** | **boolean** | **relation** | **state** | **user** | **action** }

syntactictype { **string** | **numeric** | **point** | **boolean** }

multiplicity {**nomin** | **min** *value* } {**nomax** | **max** *value* }

with **limits**

rangemin? *value*

rangemax? *value*

source? { **any** | **trigger** | **auto** | **description** }

inventory? {**true** | **false** }

relation* { **hostile** | **neutral** | **friendly** }

action* *value* **slot** { **object** | **tool** | **target** }

conflict*

option name

option name

produces* percept { **agent** | **object** | **distance** | **boolean** | **relation** | **null** }

requires* *option name* **value** *value*

The key difference between the trigger and behavior specifications is the **produces** clause. Instead of producing an action, triggers produce percepts. The **produces** clause for a trigger specifies the percept type, and also has **requires** clauses which describe which options must be present for the percept to be produced. **null** percept types are used to specify that a valid trigger will not produce percepts. BehaviorInfo files are produced as XML versions of this specification. See the BehaviorInfo.xsd schema file for the exact format.

Agent Description File

Agent descriptions are passed into BEHAVEngine as XML files with the following abstract grammar:

Agent

model *model name*

agentname *agent name*

team *team name*

scenario? *scenario number*

role?*role number*

startlocation {*x*,*y*,*z*}

startorientation? *yaw value*

allowtransfers? {**true** | **false** }

xferRequestStrategy? *value*

xferTransmitStrategy? *value*

agentoption* *name* **value** { **point** {*x*,*y*,*z*} | *value* }

inventory?

item* type *value* **slot** *value*

spatial

behavior+ **id** *unique id number*

priority *priority value*

type *behavior type*

subtype? *subtype number*

transferrable? { **true** | **false** }

allowChildTransfers? { **true** | **false** }

maxTransfers? *transfer limit*

displayInfo *value* **show** { **true** | **false** }

trigger type *trigger type*

subtype *subtype id number*

displayInfo *value* **show** { **true** | **false** }

option* *name* **value** { **point** {*x*,*y*,*z*} | *value* }

subtrigger

[uses same trigger structure as above]

option* *name* **value** { **point** {*x*,*y*,*z*} | *value* }

subsume

layer *id number* **status** { **true** | **false** }

Keywords are in **bold**, and user-specified values are in *italics*. The * symbol indicates that zero or more instances of this element may be present, a ? symbol indicates that zero or one instances of the element may be present. The + symbol indicates that at least one of the entity must be present, but more are allowed. See the Agent.xsd schema for exact format.

- **model** gives the name of the character model to use for this agent.
- **agentname** gives a string name for this agent.
- **team** gives a numeric team id for this agent.
- **scenario** specifies the scenario number for this agent, and is primarily useful when testing with the DARPA scenarios for BehaviorShop.
- **role** specifies the role number for this agent, and is primarily useful when testing with the DARPA scenarios for BehaviorShop.
- **startlocation** specifies a start location in the environment for this agent. This may be overridden by the BEHAVEngine configuration file.
- **startorientation** specifies a starting yaw/heading for the agent.

- **allowTransfers** specifies whether or not this agent is allowed to receive transferred layers from other agents (reactive teaming).
- **xferRequestStrategy** specifies the strategy used by an agent in a reactive team to decide when it should request a new layer. Default is to request a new layer when agent has only one layer or an existing layer is failing.
- **xferTransferStrategy** specifies the strategy used by an agent in a reactive team to decide which layer to transmit when a layer is requested by another agent.
- **agentoption** allows arbitrary options to be specified for an agent.
- **inventory** provides a list of **items** of a given **type** that should be preloaded in the agent's inventory. **slot** is currently not used, but is meant to specify that an object should be held or worn by an agent.
- **spatial** gives a list of behaviors to be instantiated by this agent
- **behavior** specifies a single behavior with a given unique **id** value
- **priority** specifies the priority of this agent; higher priority behaviors are executed with preference
- **type** gives the name of the behavior type to be instantiated
- **subtype** is the numeric subtype of the behavior if behavior trees are being used
- **transferrable** specifies whether this behavior may be transferred to other agents in a reactive team.

- **allowChildTransfers** specifies whether this layer can be transferred by agents which are not the source agent in a reactive team.

- **maxTransfers** specifies how many agents may have this behavior at one time

- **displayInfo** specifies a display string for this behavior in debugging interfaces or BehaviorShop, and whether the behavior should be shown.

- **trigger** specifies the trigger to be used with this behavior. Trigger elements are similar to behavior elements, and the individual options will not described separately.

- **option** specifies options for the trigger, and have the same form as **agentoptions**.

- **subsume** elements provide a list of **behaviors** that are subsumed or not subsumed by this layer. If the layer should be deactivated by this layer, **status** should be false. Note that behaviors may only subsume behaviors of lower priority, and for behaviors that have child behaviors, only the parent should be specified. Parent subsumption policies automatically apply to any children, as well.

BEHAVEngine Configuration File

The BEHAVEngine configuration file gives a list of agents to be instantiated in the scenario.

agentcore-config

agent-file+ *agent description file*

number? *number of agents*

red? *agent color red channel*

green? *agent color green channel*

blue? *agent color blue channel*

See the agentcore-schema.xsd for the exact XML format of this file.

- **agent-file** is the name of an agent description file that should be used. This file should be under the path *environmentname*/characters
- **number** is how many of these agents should be instantiated; if more than one, colors will be chosen randomly, and the name of the agent will be auto-generated.
- **red**, **green**, and **blue** can be used to specify a color for the agent. These values should be in the range 0-255.

APPENDIX C: SPLAT

This appendix describes the basics of using Splat to create simple non-graphical game simulations. The major components that may need to be modified to create a new simulation in splat include the Simulator, Conditionals, Deciders, and Actions. We describe each component here, referencing the basic Splat Python distribution.

To obtain a copy of Splat, visit http://frederickheckel.com/. This appendix only provides basic information in setting up and running a Splat simulation. For more detailed information, such as how to build new objects and behaviors, see the Splat distribution.

Creating a Splat Simulation

To start creating a new Splat simulation, see ExampleSimulator.py. The first method in the simulator is `buildWorld`. It takes in a world object, from the World library. The world is constructed with successive calls to `world.addRoom()`, which takes a Room object, also from the World library. A `Room` has three constructor arguments: `xsize`,`ysize`,`id`. The first two arguments define the size of the room, and the third argument gives the room a unique ID.

```
def buildWorld(world):
    # room 1. entrance. exits to room 2
    world.addRoom(Room(1,1,1))
    world.addTwoObjects(Objects.makeLinkedDoors('door',1,2,
       True,False,(0,0),(0,0)))
    # room 2
    world.addRoom(Room(1,1,2))
    world.addTwoObjects(Objects.makeLinkedDoors('door',2,3,
       True,False,(0,0),(0,0)))
```

The next line after the first `addRoom` creates a door between rooms 1 and 2. Note

that a single door is represented as two linked objects: one of the linked doors exists in the source room, and the other in the destination room. Because they are linked, all actions on one object are performed on the second object, so they should be treated as a single object when building the world. The `makeLinkedDoors` method from the Objects library simplifies the process of creating doors between rooms. It takes seven arguments: `name`, `location`, `destination`, `closed`, `locked`, `subpos`, `subpos2`. `name` provides the base name for the door, and will be combined with the location and destination to create the names for each of the two doors created. `location` and `destination` give the two rooms that are being connected. A door into/from a room can be created before that room is created. The arguments `closed` and `locked` are boolean values, specifying whether the door is closed and locked. The final two arguments specify the local coordinates for the door in each room. For 1x1 size rooms, this should always be (0,0). For larger rooms, the coordinates must be inside the room.

More advanced rooms can be created, such as rooms 8, 9, and 12. When rooms 8 and 9 are created, the doors are saved in a list to be used later.

When room 12 is created, these doors are referenced for creating key objects. The `makeKey` method from the Objects library takes four arguments: `name`, `location`, `doors`, `subpos`. `name` provides a unique name for the key. The `location` argument gives the room the key is located in, and `doors` is a Python list of doors (or boxes or windows that the key will open. These must be references to the original objects, and not just names. Finally, the `subpos` argument sets the position within the target room for the key. Note that in this case, the key is not immediately added to the

```
# room 8
world.addRoom(Room(1,1,8))
morelocked=Objects.makeLinkedDoors('door',8,9,True,True
   ,(0,0),(0,0))
world.addTwoObjects(morelocked)
world.addTwoObjects(Objects.makeLinkedDoors('door',8,12,
   True,False,(0,0),(0,0)))
# room 9
world.addRoom(Room(1,1,9))
world.addTwoObjects(Objects.makeLinkedDoors('door',9,10,
   True,False,(0,0),(0,0)))
lockeddoor=Objects.makeLinkedDoors('door',9,13,True,True
   ,(0,0),(0,0))
world.addTwoObjects(lockeddoor)
```

```
# room 12. key room
world.addRoom(Room(1,1,12))
doors=[]
for d in lockeddoor: doors.append(d)
for d in connectdoor: doors.append(d)
for d in morelocked: doors.append(d)
key1=Objects.makeKey('skeletonkey',12,doors,(0,0))
safe1=Objects.makeSafe('keysafe',12,[key1],(0,0))
```

room. Instead, we make a safe object in which to store the key. The only argument that is different for the safe is the third, which is the `inventory` argument. This takes a list of references to objects that are to be stored in the safe. *Safes* are objects that can store other objects, but start out locked and closed. They can only be opened by using linked objects (for storage objects that can be opened normally (and even broken), try a *box* object instead).

Now taht we have the key in a safe, we need some way to access the safe, and maybe some traps to help protect it. We create a *treasure*, another *safe* to contain the treasure, a *trap*, and a *badger*. The treasure and the trap each have just three arguments, a `name`, a `location`, and a `subpos`. The trap is a type of object that,

when activated, immediately bonks any characters standing in the room. Depending on the simulation rules, the character may then respawn (ExampleSimulator allows this).

The *badger* is a special type of character from the Agents library. So long as the badger is calm, it will not attack other characters, and remains hidden in its layer. If the badger is activated, it will attack other characters until it is bonked or no characters remain. It will not leave the room it is in, and when bonked, returns to its lair to recover. The third argument for the badger is a list of states. This list should usually be 'hidden' and 'calm'.

```
    treasure=Objects.makeTreasure('treasure',12,(0,0))
    safe2=Objects.makeSafe('treasuresafe',12,[treasure],(0,0))
    trap1 = Objects.makeTrap('trap1',12,(0,0))
    badger = DungeonAgents.makeBadger('badger1',12,['hidden','
       calm'])
    lever1,paff1 = Objects.makeLever('lever1',12,safe1,'open'
       ,(0,0),'world')
    lever2,paff2 = Objects.makeLever('lever2',12,trap1,'death'
       ,(0,0),'user')
    lever3,paff3 = Objects.makeLever('lever3',12,badger,'spawn
       ',(0,0),'spawn')
    lever4,paff4 = Objects.makeLever('lever4',12,safe2,'open'
       ,(0,0),'world')
    Affordances.linkAffordances([paff1,paff2,paff3,paff4])
    world.addObject(lever1)
    world.addObject(lever2)
    world.addObject(lever3)
    world.addObject(lever4)
    world.addObject(safe1)
    world.addObject(safe2)
    world.addAgent(badger)
```

The next four lines create levers and probabilistic affordances for the trap, the badger, and the safes. `makeLever` takes six arguments: `name`,`location`, `linkedobj`, `operation`, `subpos`, `type`. The `linkedobj` argument provides a reference to an object

that should be triggered when the lever is pulled. `operation` signals what type of action the lever triggers. Finally, `type` gives additional information about the action: whether it is an action that affects an object in the world, an action that directly affects the user, or an action that spawns a new character (such as the badger).

In this case, we create four linked levers, and the `Affordances.linkAffordances` line takes care of linking the affordances together. Finally, we add all of the objects to the world. Note that the treasure and the skeletonkey do not get added to the world directly, as they are contained in the safes. The rest of the `makeWorld` method is similar. Note that, because the treasure defines a win condition, it is returned by the method.

Because Splat is used primarily for simulating large numbers of games, the next important method is `runExperiment`. For single-agent games, this takes three arguments: `agent`, `results`, `silent`. The single character to be simulated is passed in as the first argument, followed by a results dictionary used for storing agent information. The third argument will suppress simulation output if set to `True`.

```
def runExperiment(agent, results,silent):
    keyroom=12
    goalroom=15

    theworld = World()
    theworld.silent=silent
    treasure = buildWorld(theworld)

    redteam = Team('red','default')
    redstart=1
    red.position=redstart
    redteam.add(agent)

    theworld.addAgent(agent)
    failurethreshold=5
```

The first two lines of `runExperiment` store the locations of the key room and the goal room. The goal of our characters is to reach the goal, but they first must obtain the key from the key room. The locations of these two rooms must be recorded to monitor the progress of the characters and reset the traps in the key room. The next three lines take care of building the World and set World to have noisy output or be silent. All game actions are executed through the World, which maintains the rules governing the actionspace and character movement. After setting up the world, the character is prepared for the game by adding it to the default team. Characters must be in a team, even if the simulation is only for a single character. The character is then added to the world. The `failurethreshold` defines how many turns the simulation will run if there is no agent activity.

The following section of `runExperiment` will not be covered here. For non-escape simulations or team simulations, the main loop of this method must be modified to change the rules. See the Splat distribution for other sample simulations. If possible, avoid modifying runExperiment and the methods following it when creating new simulations.

One more section of the Simulator should be modified to create new simulations. At the bottom of the simulator file is the main run section of the script. The first several lines of this script define the characters to be evaluated. First, a list of available *actions* must be chosen for the simulation. This is the character action space, and any actions not listed here will not be available. The second line gives a list of initial *states* for the characters. The provided list, $['stand','unhurt','relaxed','visible']$, should usually be correct. Next, the characters are created. The simplest character type is a

```
if __name__ == '__main__':
    actions = ['open','close','break','exit','pick-up','search
       ','hide','push','lift','pull','unlock','swipe']
    states = ['stand','unhurt','relaxed','visible']

    alice = Agent.Agent('Alice',1,actions,states,False)
    alice.addInventoryR(Objects.makeSword('alicesword',1,(0,0)
       ))

    bob = Agent.Agent('Bob',1,actions,states,False)
    bob.addInventoryR(Objects.makeSword('bobsword',1,(0,0)))

    carl = Agent.Agent('Carl',1,actions,states,True)
    carl.addInventoryR(Objects.makeSword('carlsword',1,(0,0)))

    dee = Agent.Agent('Dee',1,actions,states,True)
    dee.addInventoryR(Objects.makeSword('deesword',1,(0,0)))

    quentin = DungeonAgents.buildHandCraftedAgent(1,'Quentin',
       actions,states)
    DungeonAgents.populateAgentLayers(quentin)
    quentin.addInventoryR(Objects.makeSword('quentinsword'
       ,1,(0,0)))

    n = 1000
    results = makeFullResultDict([alice,bob,carl,dee,quentin])
    runNExperiments(alice,n,results,True)
    runNExperiments(carl,n,results,True)
    runNExperiments(quentin,n,results,True)
```

random agent. The base Agent is part of the Agent library, and takes five constructors: `name`, `position`, `actions`, `states`, `contextagent`. The `actions` and `states` arrays should be passed in here. These values can be different for each character if some characters should have a restricted action space. The `contextagent` field, if set to *True*, allows the character to make use of contextual affordances. If it is *False*, then a purely random character will be generated.

When adding base inventory for character in the simulator, the `addInventoryR` method should be used. This method ensures that the inventory is reset correctly

after each game. If the `addInventory` method is used, the inventory will *not* be reset correctly.

Note that fifth agent, Quentin, is built differently. The DungeonAgents library includes several agent definitions. Quentin is created as a hand-crafted subsumption agent, and populated with several behavior layers defined in DungeonAgents. See the Splat distribution for more information on building and modifying subsumption controllers for more information.

Finally, the last five lines run the simulation. The number of experiments to run is set by `n`, and a result storage dictionary is created for the full set of characters. `runNExperiments` actually runs the experiments, and takes four arguments: `agent1`,`n`,`results`, `silent`. `agent1` is the character to be tested, `n` is the number of games to run, and `results` is the result dictionary. The final parameter controls whether the simulation should run with full output or in silent mode. In silent mode, the simulator will output the agent results after the full set of `n` games have been run for the character being tested, but all other messages will be suppressed.