A Framework for Reactive Intelligence through Agile Component-Based Behaviors

submitted by Andy Kwong

for the degree of MSc in Computer Science
of the University of Bath

September 2003

Abstract

This dissertation introduces PyPOSH, a reactive agent architecture with loadable behavioral modules based on Bryson's Parallel-Rooted Ordered Slip-Stack Hierarchical action selection model. The framework utilizes a modular and object-oriented interface to behaviors built utilizing agile and component-based methods.

Contents

1 Introduction **3**
1.1 The Need for Better Behaviors 4
1.2 Engineering Complex Scripts 4
1.3 Agile Methodologies and Rapid Development 5

2 Artificial Intelligence and Autonomous Agents **8**
2.1 Creating Complex Agents 9
2.2 Planning and Searching for the Next Action 9
2.2.1 Symbolic AI and Production Systems 10
2.2.2 Connectionist AI and Neural Networks 11
2.2.3 Behavioral-Based AI and Reactive, Multi-Layered Architectures . . . 12
2.2.4 Other Approaches 13
2.3 A brief survey of reactive architectures 13
2.3.1 Subsumption Architecture 13
2.3.2 Society of Mind 14
2.3.3 Spreading Activation Networks 15
2.4 BOD and POSH Reactive Plans 16

3 Requirements **17**
3.1 Programming Language Criteria 18
3.1.1 LISP 19
3.1.2 Python 20
3.2 Incorporating Remote Invocation: Is the Framework Language Important? . 21
3.3 Cross Platform Issues 22
3.4 Other Requirements 22
3.5 Goal Specification 23
3.6 Problems Not Addressed 24

4 PyPOSH: Implementing the POSH framework in Python **25**
4.1 An Overview of the PyPOSH Implementation 25
4.2 The PyPOSH Action Selection Engine 27
4.3 The Agent Class 28
4.4 The Behaviors System 29
4.5 The Base Class 31
4.6 Agent Services: Blackboard, Schedule and Debugboard 31

4.6.1 The Blackboard . . . 31
4.6.2 The Schedule . . . 32
4.6.3 The Debugboard . . . 32
4.7 POSH Plan Representations . . . 33
4.7.1 Generic Objects and Element Objects . . . 33
4.7.2 Message Objects: . . . 37
4.8 Execution of Action Selection and the Drives System . . . 39
4.9 Controlling and Launching Agents: An Overview of The PyPOSH Agent Manager . . . 40

5 Defining Behavior: Building PyPOSH Agents 44
5.1 Cookietest: Basic behaviors . . . 45
5.2 Poshbot: Complex behaviors . . . 49

6 Testing and Optimization 53
6.1 Unit Testing . . . 53
6.2 Profiling and Optimization . . . 53

7 Analysis 57
7.1 The agility of PyPOSH . . . 58
7.2 Improving Performance and Response . . . 58
7.3 Improving Parsing and Changing Planfile Format . . . 59
7.4 Future Work . . . 59
7.4.1 Game and Story Agents . . . 59
7.4.2 Robot Control . . . 60

8 Conclusion 62

A Code Listing 68

Chapter 1

Introduction

This dissertation presents PyPOSH, an reactive agent framework with particular emphasis on facilitating behavior creation. Behaviors are the interface between the agent and its environment. The agent decides on what to do through a process called action selection. Action selection is the task of choosing the right thing to do next out of all of the possible choices that an agent can make. This activity depends on sensing stimuli from the environment, a service provided by behaviors. When the agent is to take an action (for example, activate an actuator), similarly this is achieved by behaviors. These sensing and acting mechanisms are tightly coupled. Agents in this paradigm are autonomous and complex, they arbitrate between conflicting goals internally and require no human intervention while doing so.

PyPOSH is based on Behavior-Oriented Design (BOD) (Bryson, 2001), an agent architecture that utilizes Parallel-Rooted Ordered Slip-Slack Hierarchical (POSH) action selection. POSH agents organize primitive behaviors into elements supporting hierarchy and sequence. PyPOSH is an implementation of POSH in Python, and allows development of these behavior using agile and component-based techniques. Behaviors are loaded into PyPOSH as modules. Each behavior module provides all of the behaviors primitives that one type of agent requires to interact with a specific environment.

The aim of the project is to create a system in which behaviors can be constructed in a component-based manner using agile methodologies. This architecture lowers the barrier to programming intelligent agents for new environments by making the process of programming behaviors easier while keeping the general purpose nature of the framework. Two behavior modules that demonstrate the abilities of the system are included with the framework. One module is used strictly for demonstrating the composition of behaviors, while the other module connects to an virtual environment by a interface to a commercial game server and shows promise as a platform for research involving team strategies and human interaction.

1.1 The Need for Better Behaviors

Marvin Minsky recently voiced his disappointment with the field of Artificial Intelligence (AI) (McHugh, 2003). His main concern is the lack of progress to wards human-like cognitive abilities for robots. AI has been preoccupied with building things from the bottom up for the last two decades. Advances have been made in reactive control (Brooks, 1986), memory representation (Kanerva, 1988), pattern recognition (Ritter and Kohonen, 1989), and many others. With so much exciting work happening, what is keeping AI from developing human-like intelligence that is practical?

Our lack of fundamental understanding of human consciousness and failure in representing commonsense knowledge has certainly hindered the development of the cognitive robot. The other side of the problem is the lack of effort integrating currently available cognitive mechanisms onto the control architectures we already have. Although the current breed of reactive control architectures may never model the complexities of a human equivalent, by making them interface with different systems, we may actually produce cognitive systems that while may not have human-like abilities are sufficiently complex enough for us to gain new insight into this problem.

If we do need better behavior integration, what do we need to do in order to make it happen? Building behaviors for agents is a time consuming task and will likely remain that way until agents are able to build their own. The question becomes - How do we make it easier and more productive for the programmer to do the job? Software engineering has provided the answer in the form of agile methodologies and component based programming. Together, they improve programmer productivity and software quality. PyPOSH utilizes these concepts to provide a framework in which behaviors can be introduced easily, resulting in making intelligent agents utilizing POSH more feasible.

1.2 Engineering Complex Scripts

The object revolution brought us data structures with type bound methods and a variety of different mechanisms in which class of these objects relate to each other. One major benefit of Object-Oriented (OO) programing is the emphasis that it places on code reuse. The techniques that OO programming languages employ to simplify design and encourage code re-use include polymorphism, dynamic binding, inheritance and the separation of class and object instances.

These technologies are in widespread use today and Ousterhout (1998) suggested that from observation, the average level of code reuse are in the range of 20-30%. He then proceeds to explain the benefits of developing software with scripting languages. Scripting languages are high-level, dynamically typed and dynamically compiled or interpreted. Compared with system programming languages such as C, C++ and Java, these languages are very high-level (in terms of the number machine instruction per language statement) and typically execute from 100-1000 instructions/statement compared to 5-10 for system programming languages.

Ousterhout developed a set of criteria to determine the suitability of a project for using scripting:

Does the software involve -

- Gluing together components?
- Working with diverse components and data types?
- Implementing a Graphical User Interface (GUI)?
- Requiring an extensible architecture?

Answering yes to the above questions indicate that the project would be more suitable to implement in a scripting language. Conversely, if the software involves -

- Implementing complex algorithms and data structures.
- Manipulation of large data sets with application speed as a priority.
- Functions which are well-defined and slow-changing.

Then such projects may be better off utilizing a traditional, system programming language.

Considering that Ousterhout developed the widely used scripting language TCL, it is no surprise that the paper argued for increased proliferation of scripting.

Schneider and Nierstrasz (1999) suggested that the use of OO technologies is not yielding the desired increase in productivity and code reuse due to a variety of factors. The main factor identified is the emphasis on class hierarchies and not on the interaction between objects. They communicated the need for compositional features in languages to support object interaction and introduced the experimental language PICOLA (derived from π-calculus) to facilitate research. There are no general purpose compositional languages yet, however the component paradigm need not be restricted to an implementation at that level. The authors touted the advantages of using scripting languages to allow the binding of components on a higher-level. This is the exactly the type of application that we are looking at.

1.3 Agile Methodologies and Rapid Development

Modern software development methodologies emphasize the importance of rapid prototyping in order to create better quality software faster. Agile techniques such as Extreme Programming (XP) (Beck, 1999) requires the use of prototyping to formulate ideas and takes an iterative approach to development.

Proponents argue that by cutting down on the lengthy requirement specification and design stage and moving these activities to the implementation stage yields significant increases in efficiency. This is largely due to the potential to over design software components, neglecting the knowledge that would be gained in the actual implementation process.

The core ideas in XP are to defer design work until it is absolutely necessary, to start implementation of core features early, to build additional features through an iterative process, to use agile/lightweight modeling and to employ unconventional programming concepts such as pair programming.

Programmers creating software utilizing these methods would naturally perform better with a complete set of tools that encourages development in such an environment. Ousterhout (1998) suggested that the answer lies in using high-level scripting languages as the glue between components. The premise of his argument is that traditional systems programming languages such as C, C++ and Java are too verbose. As the number of lines of code that a programmer produces each year is roughly equal irregardless of the language used, programmers are more productive in high-level languages that lets the programmer focus on connecting together components rather than creating them from scratch. These advantages are provided by some of the very high-level language features. Dynamic typing and through this improved code re-use is important as it allows the programmer for focus on the overall design rather then getting dragged into implementation details. Concise syntax and easy to understand code are also important features. The self documenting attribute makes it easier for maintenance and review.

Boehm (1999) introduced a categorization system of the different types of rapid application development (RAD) methods from a software engineering viewpoint. Of special interest to us is Generator RAD (GRAD) and Composition RAD (CRAD).

GRAD is facilitated by very high-level languages or tools that are likely domain-specific. An example of a GRAD tool is Agentsheets (Repenning, 1993). GRAD tools are highly specialized and narrowly focused. Taken out of context, the system would be very inefficient. Subsequently flexibility is low if extensions are not available (due to narrow audiences) or hard to create (due to architectural issues). Similarly, scalability is another important consideration due to particular implementations of such high-level tools.

CRAD describes a process in which a small team assembles a project with the general purpose tools, connecting together ready-made components into a complete package. Components can be be off-the-self products, middle-ware layers, web services, anything that can be abstracted and componentized. The important piece of the puzzle here is the glue or framework that ties these individual parts together. This type of development is much more flexible compared to GRAD, as it allows disparate components to interface with the system. Scalability is another issue however, which depends heavily on the bottleneck, or the weakest component in the chain.

Both Ousterhout and Boehm argues that an important prerequisite in rapid development of flexible systems is a powerful glue mechanism for the components. An empirical study on scripting languages conducted by Prechelt (2000a) found that the programming time for scripting systems which offered these advantages is approximately one-half to one-third of programming languages such as C and C++. He took measurements on the number of hours that the programmer worked on and the number of lines of code that the resulting program is in. Prechelt also stated Ousterhout's assertion that the number of lines of code produced per programmer per year is relatively constant and referenced it to Boehm (1981). If this is sufficiently true, then we can estimate programming efficiency by looking at the source code. If a program that accomplishes a equivalent task is shorter in a scripting language, then we can reasonable assume that this programming method is more productive empirically. With less lines needed for any particular program, the programmer has more lines left for other tasks.

Chapter 2

Artificial Intelligence and Autonomous Agents

Artificial intelligence (AI) research covers a broad range of topics, including fields such as pattern recognition, linguistics, robotics, bio-informatics and more. Thus, the nature of AI itself, is rarely agreed upon. Winston (1992) describes it as "The study of the computations that make it possible to perceive, reason, act". McCarthy (2003) states that "It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable." Every researcher seems to have their own unique answer on what AI is, and what the goals of AI research are. If the aim of Artificial Intelligence is to create Intelligent machines, then what is intelligence, and how do we identify it? Turing (1950) gave his answer in the form of the Turing test. Intelligence on the other hand is much harder to define. Common sense tells us that humans are intelligent. Can we describe animals as intelligent? What about plants?

To this, McCarthy (2003) answers, "Intelligence is the computational part of the ability to achieve goals in the world. Varying kinds and degrees of intelligence occur in people, many animals and some machines."

The important concept to keep in mind is that intelligence not absolute. Is a rock boulder intelligent? Probably not. Inanimate object do not have goals nor desires. Most lifeforms, however, do have a goal. This goal being the propagation of genetic material through life itself. Assuming the mindset of a evolutionary biologist, the body can be considered the vessel that protects and ensures that survival of its genes. Intelligence, then, would become a facilitating factor in which genes survives, and which do not. From this perspective, one can position humans on one extreme of the spectrum, but we are by no means the only beings possessing intelligence.

When we have a agent that has its own goals, the question is naturally how to make the agent achieve it. The computational part of the ability to achieve goals can be viewed as the search for the decision on what to do next. For any given agent that operates autonomously in its environment, how does it decide on the action which would take further along the path to the goal? Practically, the problem is how to model this process. This process being Action Selection.

2.1 Creating Complex Agents

A complex agent can be thought of as an autonomous and high-level abstraction; one which can decide on when to execute actions to achieve particular goals, and hence possess some degree of intelligence. This decision process is a search, and how it is carried out determines the behavior and intrinsic properties of the architecture. There are many ways to build a complex agent, or more specifically the *mind* of one, and a summary of the many architectures can be found in later on in this chapter. These are often called Agent architectures, Action Selection Architectures, Cognitive Architectures or Robotic Control Architectures depending on whether from the perspective of AI, cognitive science or multi-agent systems although they may have slightly different definitions. We will take a view which is centered around AI and Robotics as opposed to one which is centered around Autonomous Agents or Multi-Agent Systems (MAS) (Wooldridge and Jennings, 1994). The agent itself is implemented in software, but it may be also be the software that controls physical systems such as actuators or sensors, as in a robot. Conversely, an agent may not interact with any external environment at all, if the agent is to run as closed simulation.

We need to build complex agents for many reasons. The simulation of animal behavior is essential for our understanding of natural intelligence and large set of related problems such as modeling natural environments. Autonomous agents can be used to to complete tasks that contains a substantial amount of complexity and requires adaptivity. We need these agents to help us search for the best answer to a question from billions of documents, or to explore extra-terrestrial environments in a robotic body, or to populate virtual worlds and interact with humans.

The following parts provide an introduction on the different approaches in how complex agents are designed and constructed. This would be accompanied by an overview on the major architectures and methodologies.

2.2 Planning and Searching for the Next Action

Planning is the process of searching for the next action that will take us closer to an specific goal. In essence planning is *search* leading to intelligence. It was once assumed in AI that planning can be done in the same way as proving theorems - When given a full representation of the environment in its initial and final state, we would be able to sequence the actions necessary to achieve the goal. Furthermore, the process of planning such actions were thought to be mathematical, in the sense that a particular problem would have a particular formal solution. This approach is now referred to as *conventional planning* and *Symbolic AI* which is explored in more detail in Section 2.2.1.

Action Selection is the process that choses which action to take according to the plan. In reactive systems this is done at run-time as compared to traditional planning which involved selecting actions and placing it into a sequence. A variety of action selection mechanisms exist, each having different properties and approaching the problem from different directions. One can argue that the term *planning* is obsolete, as what was originally done by planners are now in the domain of *action selection mechanisms* (ASMs). However, the term reactive planning has become almost synonymous with the action selection problem.

The primary task of action selection is to identify possible action and arbitrate between conflicting ones. The agent or robot can only perform a limited number of operations at any one time and has finite resources. When to focus attention on a particular action and when to release this focus is that main determining factor of a good action selection mechanism and a mediocre one.

2.2.1 Symbolic AI and Production Systems

With the goal being to recreate intelligence, we need to conceptualize a model for it. Symbolic AI is the first solution to this challenge. Many of the classic successes of AI (Newell and Simon, 1963) (Winograd, 1972), utilized high-level domain specific knowledge and reasoning. This top-down approach rests on the notion of representation. Representation in this form is said to be explicit. Each state, attribute or decision is internalized, categorized and stored as a symbol.

Symbolic AI agents often have deliberative inference mechanism as they require to a internal representation of the external environment before it can apply the relevant heuristics. This is modeled on the work of Post (1943) which involved *Production Systems*. In a production system, the set of all possible states is the *state space*. The state that a system is currently at is the *current state*. The desired outcome of the system is represented as the goal state. The central premise of production systems is that in order to achieve the goal, the system would be required to search the state space for a path from the initial state to the goal state.

The control architecture of these agents utilizing this mindset separates the process into three components - sensing, planning and execution. The approach is to build a model of the environment through the sensory system, figure out what to do next through the planning system, and execute the action accordingly. This approach is commonly known as the *sense-plan-act* (SPA) approach and sometimes described as *conventional planning* or *deliberative planning*. Given a description of the initial state, it will try to deduce a list of the steps needed to reach the final state. Such a list is often called an *action sequence*. As the study of the organization of behaviors and processes pertaining to intelligence at the time focused on hierarchical and sequential states, many conventional planners based their search of action on hierarchical and ordered models. However, not all agents which utilizes symbolic representation use deliberative planning on hierarchal models.

Symbolic representation may be a crucial mechanism for specific types of intelligence, and maybe even essential as a general mechanism of intelligence for humans. However, when one considers the practicality of representing the astronomical number of states and decisions available to any single agent in a complex environment, there is no doubt that utilizing explicit representation is most likely a sub-optimal method, even when the search involved in planning is bounded. In extreme circumstances and when agent is in possible danger, there may be no time to internalize environment state and search for a good-enough solution. These systems would have to micromanage and integrate a variety of different components and processes including actuators, sensors, environmental state, modeling, planning and processing. Trying to plan through all of the different states within a moderately complex environment is near impossible; as soon as a good-enough plan sequence has been generated, the environmental state might already have changed.

Even with its list of failings, we can not discount the value of explicit representation. What symbolic representation excels at is the solving of high-level or formal problems. However, not all intelligence operate on those levels. Let us say that a bicycle is out of control and heading toward a person. What does she do? The instinctive reaction from the victim is to evade as quickly as possible. The thought probably never got on mind. This "reaction" is a form of intelligence. However, if the same strategy was used in playing a game of chess, it would not provide much of a challenge.

Strict conventional planning strategies has to make certain assumptions due to the way it solves problems. In order to find the sequence of actions leading from the initial to final state, it is assumed that the number of states within the system from the start to the end of a planning session is finite and manageable. Action effects are assumed to be predictable to allow planning to occur with congruency. The world state is assumed to consistent; if the state is modified unexpectedly, re-planning would be needed. Shakey the robot (Nilsson, 1984) is the embodiment of this. The robot would sense the environment through its different sensors and would spend long periods formulating plans. In the middle of executing the plan, however, Shakey would most likely discover that the environment has changed. It will then have to painstakingly start all over again.

2.2.2 Connectionist AI and Neural Networks

Connectionist AI takes the opposite approach to production systems by modeling biological process instead of modeling their effect. Lloyd (1989) stated, "The central idea of connectionism is the cognition can be modeled as the simultaneous interaction of many highly interconnected neuron-like units". Aleksander (1996) furthered this by adding that consciousness can be measured in degrees, and that complex artificial neural networks (ANN) systems have their own type of "Artificial Consciousness".

Neural Networks are very capable pattern recognition tools, and there is much ongoing researching that area. These networks do not do away with representation altogether, but instead changes the nature of this representation. Instead of a special symbol that represents a particular concept (explicit representation), the concept is represented by the interaction between these neuron-like nodes (distributed representation). With a number of sufficiently complex neural networks arranged properly, it is believed that consciousness or intelligence can result as a emergent property. As a paradigm for intelligence, studies have been focused on modeling the behavior of the Basal Ganglia (Baldassarre, 2001a), thought to be the action selection mechanism in nature. Recent research from (Baldassarre, 2001b) shows the use of ANNs in performing these tasks in a environment with multiple goals.

2.2.3 Behavioral-Based AI and Reactive, Multi-Layered Architectures

A new type of architecture dominated the field from the mid 80's onward, commonly known as *reactive planning*. *Reactive planning* takes a distributed and concurrent approach to the planning problem by localizing the scope of planning. A reactive system does not form comprehensive action plans to achieve goals. Instead, actions are executed due to individual conditions being triggered. Rather than working with the agent in its entirety, we deconstruct it into components that has plans to react to specific states, should they arise. Linked together, these components perform better than a monolithic system. The essence of reactive planning is to relieve to need to form explicit symbolic representations of the environment internally, as we only sense the information that we need for the specific action. A variety of architectures are grouped as reactive, including Subsumption Architecture (Brooks, 1986), Spreading Activation Networks (Maes, 1990), Extended Rosenblatt and Payton Architecture (Tyrrell, 1993), W-learning (Humphrys, 1997) and Teleo-Reactive Plans (Nilsson, 1994). Collectively termed as *Behavior-Based AI*, members of this group have at least one property in common - Intelligence is decentralized into individual behavior modules that reacts to some sort of trigger and connected together by some means. How there modules are organized and connected is the defines the architecture. For example, Brooks used layers for the Subsumption Architecture while Maes utilized non-hierarchical behavior nodes connected together for Spreading Activation Networks.

Reactive and Behavior-oriented architectures tend to represent latent knowledge as nodes associated with triggers, modifiers and output. Knowledge is associated to a certain situation though one or more of these nodes. This is what constitutes the *plan* in the behavioral approach, which is not necessary a comprehensive plan to achieve any particular goal.

Reactive planning architectures excel at situations when it is impossible to calculate in advanced all possible states of the environment, as the intelligence that handles each event is localized to the event. An example of a practical application would be in spacecraft control. Due to the large expanses of space and the time delay for control signals to reach the craft, deep space probes require a certain amount of intelligence to control and maintain itself in absence of direct communication from mission control.

Other influences in the field include ethology related theories (drives, time based selection), integrated systems and pattern recognition/repetition system such as Pandemonium (Selfridge, 1959), together with Cellular Automata (CA) and Artificial Neural Networks (ANN). These techniques are often used within or above the underlying architectures. Kim and Cho (2001) present one example of such use.

Most current agent architectures in AI research combine different methodologies into separate components or layers. Typically these systems contains two or three layers and are subsequently called two-layer or three-layer architectures. These architectures, in which the bottom layers are typically reactive, posses the advantage of being task-able and reusable, compared to earlier architectures where all control structures are expressed in static programs.

Gat (1997) described the three-layer ATLANTIS architecture as-

- **The Controller** layer defines multiple actions that interfaces with sensors and actuators. These are sometimes termed *behaviors*. These behaviors run separate to each other interfacing with their respective sensors and actuators.
- **The Sequencer** is tasked with selecting the right behavior to activate at a given time and manage the execution by supplying details immediately related to the execution. This layer performs the function commonly known as Action Selection (AS).
- **The Deliberator** runs separately and above the other two layers performing complex tasks such as planning, search and other compute-intensive operations. This layer interfaces with the sequencer by producing new plans by learning or answering queries.

2.2.4 Other Approaches

Reactive/Behavior-based and Layered architectures are by no means the only ones competing in the field of control systems for complex agents. The field of Autonomous and Multi-Agent-Systems is dominated by several theories such as PRS and DMARS (d'Inverno et al., 1997). SOAR (Newell, 1990) is another example which uses symbolic representation as a basis for creating a unified theory of cognition. Systems such as PRODIGY (Veloso et al., 1995) contributed to the field by utilizing an integrated approach to planning and learning.

2.3 A brief survey of reactive architectures

2.3.1 Subsumption Architecture

Brook's Subsumption Architecture (Brooks, 1986) began the transformation from traditional AI architectures into the reactive architectures that are prevalent today. In the paper, Brooks communicated the need for decomposing competence into different groups that would run together in building a control architecture for mobile robots. These groups are then layered and run in parallel, whereas higher-level layers would have the capability to suppress the activity of lower layers. This attribute of allowing higher-level layers to preempt lower layers is known as subsumption. In order for subsumption to work properly, each layer would have to have the ability to see the state of a lower layer and operate individually and unaware of higher-level layers.

Subsumption works through two means, inhibition and suppression. Inhibition prevents signals from traveling to and from the sensors and actuators. Suppression works similarly to inhibition but replaces the signals and messages with a surrogate one.

The architecture was designed so that each layer is responsible for keeping its own perception of the environment. Modules that interact with a specific component of the environment is tied to the affected sensors or actuators. There are no system-wide structures to keep state of the system or the environment, in fact, it is argued that the less state kept within the system the better. The state of the environment should be sensed and reacted upon as frequently as possible.

Brooks used the Subsumption Architecture in robotic systems successfully on many robots. The process of decomposing intelligence into individual layers proved to be a more natural and efficient way to represent and process knowledge when compared with traditional architectures at least in the field of robotics.

2.3.2 Society of Mind

Minsky's Society of Mind (SOM) (Minsky, 1986) proposes that the human mind is the aggregate of multiple small components (or agents) that are limited in scope, that when connected together, results in intelligence. Each of these components, although complex in itself, can perform only relatively simple tasks. Intelligence therefore forms as emergent behavior from the complex interaction of simple agents.

There are many components to SOM including agents, agencies and K-lines. Different specialized agents exists to process different types information. For example memory is handled through K-lines connecting polynemes and isonemes. An explicit topology for these components exists but is not defined rigidly leaving the implementation of such concepts open. Minsky's latest work (Minsky, 2001) continues this train of thought by exploring emotions in detail and how they relate to the SOM.

Unsurprisingly, this idea became hugely influential together with the reactive and behavior-based school of thought. Although the book does not discuss much on implementation, the introduction of *how* such tasks may be performed by the mind is enough to spark interest in many researchers. A large collection of research is subsequently based on implementing, extending or refuting Minsky's ideas.

2.3.3 Spreading Activation Networks

Maes (1990) proposed an action selection architecture based on nodes which form a series of links. Some of these nodes are tied to actuator/sensor loops and can be activated by external stimuli. The links could be predecessor/successor links, conflictor links or inhibition links. The structure of these links are non-hierarchical. Activation, an abstract quantity much like ether, is spread among all of the nodes in the network. Due to stimuli, activation is spread and channeled to other nodes within the network due through the predecessor and successors links. When choosing what the current task should be, we would just pick out the node containing the most amount of activation. The problem with this type of system is the setting of the activation trigger and output is mangled by hand. It is hard to tune the network to behave in a desired way.

Tyrrell's Simulated Environment and Rosenbatt and Payton's Architecture

Tyrell (1993) performed a comparison and analysis of Rosenblatt and Payton's architecture (presented as an alternative to the Subsumption Architecture) (Rosenblatt and Payton, 1989) and traditional hierarchies. Tyrrell argues that Rosenblatt and Payton's free-flow hierarchical organization of behaviors is a better model and supports this with evidence by creating and running simulations of agents in his Simulation Environment (SE) (Tyrrell, 1993) that tested the performance of Action Selection Mechanisms in complex situations.

The argument by Maes that a top-down level of control exhibited by traditional hierarchies are bad for action selection is shared by Tyrrell. Maes approach was naturally to create a mechanism that did not employ hierarchies at all. Tyrrell pointed out that by utilizing a modified hierarchy similar and maybe superior result may be achieved. Rosenblatt and Payton's original architecture used free-flow hierarchies to address the shortcomings of the top-down hierarchical approach termed as Hierarchical Decision Structures (HDS) through allowing the hierarchical components to have flexibility in choosing which candidate to defer control to.

(Tyrrell, 1993) contains an extensive comparison of action selection mechanisms by testing each of them in the SE. Surprisingly, all four of the architectures tested faired relatively poorly at first, but Rosenblatt and Payton's mechanism fared the best after slight modification. A noteworthy point is that the Edmund architecture (Bryson, 2000b) fared the best overall in action selection mechanisms tested in the SE in a followup study.

W-learning

W-learning, an architecture proposed by Humphrys (1997) is based on Reinforcement Learning (RL), a type of distributed learning. RL operates through a training process that rewards positive action and punishes negative or undesired actions. W-learning is hierarchical in that agents can be split into sub-agents that handle a specific part of a problem. Each sub-agent is a separate entity that handles it own learning. Individual agents then suggest a W-value depending on the state of the system and their previous actions. The parent just picks the sub-agent with the highest W value. Each time this cycle is complete, agents reevaluate the choices that the system made and form a new w-value with regard to their own goals to prepare for the next cycle.

2.4 BOD and POSH Reactive Plans

Bryson (2000b) argues that some degree of hierarchy exists in intelligent search and proposes that fully reactive planning would benefit from introducing a limited degree of hierarchical control. This approach is similar to the organization of behaviors advocated by Brooks (1986) and Tyrrell (1993). Partially based on her previous thesis, she developed Behavior Oriented Design (BOD) (Bryson, 2001), an architecture and methodology that combines hierarchical reactive planning and modular semi-autonomous behaviors to produce a layered system with emphasis on sequencing and action selection. The methodology portions are concerned with the process of behavioral decomposition and iterative development of the behaviors.

In the BOD model, parallel behaviors control how a BOD agent behaves, and when the behavior is expressed is controlled by the action selection mechanism. POSH (Parallel-rooted, Ordered Slip-Stack Hierarchical) reactive plans convey the information necessary for action selection to operate. Here are a list of POSH element types -

- Action patterns are simple ordered sequences of actions that perform a task, but also allow parallel ordering of elements.
- Competences are a collection of unordered prioritized actions (as Competence Elements) related to performing a task. These child actions have their own trigger for activation. These elements are derived from basic reactive plans (BRP).
- The Drive Collection and Drives are another variation if the BRP and operates in a similar manner to the Competence/Competence Elements. The key difference being that each drive have a value α, which contains the currently active element under a Drive. This feature implements the slip stack, which allow action selection to ignore other elements within the drive until the active element has terminated. The introduction of the slip stack is a performance measure that relieves the need for the whole drive hierarchy need to be traversed for each cycle. The traversal of the drive tree only happens when the active element of a drive has terminated.

Agents created under this architecture performed well in a variety of conditions. POSH agents ran in Tyrrell's (1993) Simulated Environment (Bryson, 2000a) outperformed all of other agent architecture testing in that environment (see Section 2.3.3). Its use of Hierarchy and Sequence helps it maintain focus in circumstances where fully parallel architectures can not.

Chapter 3

Requirements

Creating an BOD agent utilizing POSH involves programming behavior and creating reactive plans which uses them. Realistically however, the behaviors needs to be linked with and developed in a framework that handles action selection and interpretation of plans. The current POSH implementation supplied by Bryson (2001) is programmed in Common LISP (CLOS). Any agent created using that implementation would have to be at least partially implemented in the same environment.

LISP is a functional language which has a long history and a devoted following, mainly in AI and Cognitive Psychology. Many of the techniques and features in modern scripting languages were first pioneered in LISP. Dynamic typing is one example of this. Due to different factors, however, the uptake of LISP was slow. LISP grew in many directions and resulted in the splintering of efforts which lead to fragmentation of the language into multiple dialects such as Common Lisp, Standard Lisp, euLISP, etc. The limited use of LISP as a general programming language meant that no central repository of extension modules and libraries were available for it. This severely limits the power of using LISP as a component framework for integration.

Scripting languages such as Python, Ruby or Perl provides imperative/object-oriented syntax and a large selection of components and modules. They have attracted a large base of users with their strengths, bringing momentum and much needed support.

Decisions would have to be made with respect to the language and approach used for the implementation. The implementation language we consider as the best candidate at the moment is Python. Python has a strict emphasis on clear and readable code and the language itself has a strong object-oriented foundation. With a large number of modules and classes available for connecting components, Python is a mature scripting language and for components which do not have a Python interface, creating new wrappers for existing libraries in C is trivial.

Another important consideration in development is accessibility of the components. Any problems that the programmer has in searching for ready made components (such as those for data storage or for interfacing with hardware) is a hindrance to development. Languages with a central repository of available modules, that are freely available with the minimal amount of usage restriction is preferable. Both Python and Perl have fairly extensive libraries of modules, although Perl's CPAN repository is bigger by far. The number of Ruby modules is growing but selection is more limited due to it being youngest language.

Of particular note to this discussion is the performance of scripting languages. Bangley (2003) finds that Python is on average 20 times slower than C++ and up to 100 times as slow in common operations. In contrast, LISP is less 2 times as slow on average when the code was compiled. However, Prechelt (2000a) reports that in implementations of string processing routines, the run times and other performance metrics for scripting languages such as Python and Perl is less than a factor of 2 over similar implementations in C.

3.1 Programming Language Criteria

The language choice of the framework determines which language the behaviors need to be implemented. This is true even with remote invocation of methods and procedures (see Section 3.2) unless the plan representations, module organization and messages are stored and exchanged in a data format that is totally independent from the implementation. In the latter case, specific wrapper or translation components must be used to interface with the standard formats, adding additional complexity to the agent.

Specifically we will consider the following language criteria in selecting the implementation language:

Clear and concise syntax is very important. Overly verbose syntax may be more powerful and allow programmers more control over the program during execution, however, the trade-off is more development time used in repeating common tasks. On the other hand, overly expressive syntax leads to the problem of too much complexity in the code itself, resulting in code that is not readable or maintainable. The use of regular expression as a syntactic construct in Perl is one such example.

A related problem is the familiarity of the language constructs. A simple and concise language also be familiar to a majority of programmers. Most programmers are at least familiar with one major imperative language. Our choice should at least cater those this style or programming.

Dynamic typing is a important part of building rapid prototypes. In languages that support dynamic typing the system determines how variables are interpreted depending on in which context those variables are used. It increases productivity by eliminating the need to associate a type with a variable at compile time. For the programmer, this process (static typing), is tedious and time consuming. Furthermore, in OO based scripting languages, the use of dynamic types allow more code reuse by allowing child methods to overload parent method without the concern for the type signature of the method. For example, we can inherit a boat-container object from a container object that returns boat objects which can be immediately used without typecasting.

Interpreted and dynamically compiled languages offer more flexibility in development due to the absence of a lengthy compile process, which allows the programmer to run tests more frequently. These languages also offer better cross-platform compatibility in many cases, as the software runs on top of virtual machine or byte-code interpreter, the need to recompile code for different architectures is non-existent, greatly reducing the number of problems that can occur due to the peculiarities of different architectures.

The availability of components ready to be used is another important factor in the decision. On of the method to achieve the goal is to allow the agent to use the framework to tie together disparate components. The license and quality of these components are also an important factor in consideration, as these attributes affect the use of the components.

3.1.1 LISP

LISP is considered as the first general purpose functional programming language. Compared with imperative programming styles which encourages iteration and variable use, functional programming involve defining functions and passing parameters to these functions, with recursion being favored over iteration. However, all LISP dialects derived from the the first version of LISP (apart from the original version itself) have standard features of imperative languages, such as variables, assignments and iteration.

LISP was used extensively in the early development of AI. It had a several key advantages to the its functional nature and also due to it being typeless. Most variants of LISP handles compilation and execution in their own method, some operate on a purely interpreted level, while others have the option of compiling to native machine code. A excellent example of the power of LISP can be found in the GNU/Emacs editor.

Unfortunately, there were several problems with LISP that hindered the uptake of the language outside a few specialist fields (such as AI). Firstly, the functional and recursive nature of LISP and the extensive use of lists did not map well performance-wise to the von-Neumann architecture. Additionally, the proliferation of imperative languages (such as ALGOL, FORTRAN and subsequently C) meant that the syntax of and use of LISP is largely unfamiliar to many programmers.

Due to the fragmented nature of LISP, there are many dialects of LISP that are incompatible with each other. Scheme is one such dialect that has gained a sizable following. One of the major features is that Scheme treats function as first-class entities, allowing them to be assigned to variables and to values of expressions.

3.1.2 Python

Python van Rossum (2003) is a general purpose, very high-level, open-source programming language that has gained a relatively large following since its introduction in 1990. Python has a large list of features, many of which are targeted toward increasing the productivity of the programmer. Some of the these features include -

- Rapid development cycles due to the absence of costly compilation and linking operations.
- The dynamically typed nature of the language allows programs to be more concise and easier to read.
- Supports Object Orient Programming, class hierarchies brings benefits such as code re-use.
- Cross platform support due to the ability to compile to byte-code for Virtual Machines(VMs) (Hugunin, 1997) and the multiple platforms that the interpreter runs on.
- Garbage collection and memory management comes together with the language.
- Python supports LISP like constructs such as λ functions.
- Functions, classes and modules are first class objects.
- A large collection of available components to interface with existing systems software.
- Powerful threads support, allowing components to run concurrently and separately.
- Open-source with a relaxed policy in use and distribution.
- Creating a GUI is relatively easy with high-level bindings for a variety of graphical toolkits (GTK, qt and tk).
- Clear and familiar syntax that requires whitespace in formatting, making code easier to read.

Python has become one of the most popular complex scripting language in use today. Its designers have focused on readability over the development of the language. The result is that the language maintained clear syntax even as additional features were added to the language. This leads to code that is easier to understand and maintain. Agile methods have been tested extensively in Python. PyUnit, the Python unit testing framework is a standard module in the Python distribution. PyUnit is used to build unit tests which is a part of XP.

Georgatos (2002) conducted a study on the suitability of using Python for education in a pre-university setting and raised two important points; compared with traditional imperative languages, Python code achieves the same result in significantly less code and brings faster development. More importantly, Python attract students attention, and raised their enthusiasm for programming. This may be attributed to the ease of use of the language, and the huge improvement over the existing languages taught such as Pascal.

Python is a powerful language underneath the easy-to-use syntax and supports many advanced language features. Almost everything in Python is a first-class object, methods, object, classes, modules can all be passed around and stored in variables. A robust run-time exceptions model allow Python programs to handle unexpected data gracefully. By utilizing tools such as SWIG (Beazley, 1996), wrapping around and providing access to external libraries is quick and painless. An report on the use of Python (and SWIG) in a large-scale scientific application is detailed by Beazley and Lomdahl (1997).

Python fits the criteria we defined in Section 3.1. However, since Python is a scripting language, the rich functionality of the language results in a performance trade-off.

Prechelt (2000b) performed an empirical study on several scripting and systems programming languages based on string operations. Volunteers were asked to solve a particular programming problem in a specific programing language. The results and related measurements were collected for each of these volunteers in the sample population. The results were surprising on two fronts. First, the report established that in the average implementation, the programs in Perl and Python lag their counterparts in C and C++, but are only slightly slower (which is acceptable nowadays due to the abundance of processing resources). Second, the report indicated a very substantial increase in performance when analyzing the amount of time each programmer spent on the task, and the number of lines of resulting code. The inclusion of built-in list and dictionary types that are implemented natively no doubt contributed to the strong show as certain operations in Python, such as comparisons are much slower than their C or C++ counterparts.

3.2 Incorporating Remote Invocation: Is the Framework Language Important?

There are a number of competing component technologies that offer a framework for developing distributed and remote applications. Examples include Java-RMI, CORBA, DCOM, and SOAP/XML-RPC. Each technology provides a different approach the the problem, including the choice of programming language, method of remote invocation, etc. Which one to choose depends directly on the needs of the particular project and the choice of programming language.

As we have preliminarily settled upon the Python, we can focus specifically on the choices that are available on the platform -

Corba (through Fnorb, an object request broker implemented in Python) is a standards based distributed component architecture that allows language independence through passing messages between objects through brokers.

Python Remote Objects and DOPY are the Python versions of Java-RMI, allowing remote Python objects to be utilized.

SOAP and XML-RPC are protocols for the development of RPC mechanisms through XML message passing, originally built for web services, but adopted for use in many MAS systems.

After reviewing these mechanisms, an interesting question is raised. By using a language neutral remote invocation mechanism, can we ignore how we implement the framework altogether? I would answer no due to several factors. Choice in programming languages is a good thing, however, not all projects would call for the use of several different languages. Adding a remote invocation layer would increase the complexity of the project and when utilized introduces a non-negligible performance degradation (Davis and Parashar, 2002), even when assuming best-case scenarios. It is a better strategy to assume that most programming would be done in the native language of the framework and to treat remote access as a special case. Therefore, we should optimize the system architecture for native operation.

3.3 Cross Platform Issues

There are many issues to consider for software that is to be intended to run on different platforms. Different operating systems implement functions and features differently, and this fact has to be taken into account even when the language environment chosen is guaranteed to be cross platform. Issues such as timing granularity is especially important in our case as real-time operation of action selection is required. Other differences result from the difference in the storage of file attributes and filenames. The resulting software would have to take into account all of these differences and provide a consistent interface for the user and programmer.

3.4 Other Requirements

The base system must preform to a minimum set of performance requirements. Reactive action selection should remain reactive, and would require the frequency of action selection cycles to remain above a reasonable value. This is an especially challenging problem in very high-level languages as data operations tend to take much longer. The other side of this issue is the amount of computing resources that the system utilized. The aim is to keep this number as low as possible to allow other processes to run alongside the agents.

A graphical user interface to launch and control agents is important. It would allow developers to focus on the design of behaviors rather than constructing a user interface, and it facilitates debugging by showing the internal structure of the agent at run-time. The blackboard and schedule are parts of the system that should be shown at any given time during an execution.

3.5 Goal Specification

The goal of the software system is to provide an complete agent framework utilizing POSH action selection to enable development of agile behaviors. The software system must be implemented in a programming language that meets the criteria defined in Section 3.1. For example, dynamic typing, a large repertoire of existing components easily accessible, clear and familiar syntax and easy code maintenance are some of the considerations for the selection of the language used. The framework itself would then be tied to the different components with the aid of the programming language and its features. This language that is decided upon is Python.

Emphasis must be placed on the creation of agile behaviors. The behavior interface itself should have structure and be well specified. The aim is to allow rapid development and prototyping of these essential components.

The software must be able to read and load in planfiles in their current format. POSH plan representations must be implemented in an object-oriented manner.

The Python distribution targeted is version 2.2 and above. The decision to standardize on 2.2 is due to the inclusion of new features that were not included or less mature in previous versions. Generators and lexical closures are examples of such features.

The resulting software must be usable and perform reasonably, whereas reasonable can be defined as capable of operating a real-time agent or robot. The resulting action cycle must make decisions at a frequency of 30 cycles per second or above on modern hardware.

The software must run on at least two major platforms and perform near identically.

The resulting software must also be maintainable and readable and the innards of the system must be made clear and understandable. Documentation explaining the basic use of the framework is a requirement.

The system should include a graphical user interface to control the creation of agents and controlling them while in execution. Control of the agent would allow the internal state of the system to be accessible to the user.

Remote invocation may be implemented but is not required.

In addition to the features in the reference implementation, the resulting software may contain addition modules to aid in the development and deployment of agents built upon the framework. Such work will allow the connection of agents built upon the framework with components such as voice recognition (Huang et al., 1993). The final target of the system would be to build a agent simulator in a virtual world integrating projects such as Alice (Conway et al., 1994), robotics environments such as Pyro (Blank et al., 2003) and the Gamebots Server (Adobbati et al., 2001).

3.6 Problems Not Addressed

Outside the scope of the project is the graphical user interface (GUI) to create the plans interpreted by the POSH action selection mechanism. (Bryson, 2001) included code that provides a basic interface for creating and editing plans. A separate project of implementing a full interface for reading and writing POSH plans is under development at the time of writing.

The resulting software in itself would not be an complete and integrated environment for simulating agents, although the integration of the result with simulation, virtual reality or robotics frameworks would bring it closer to this goal. The behavior programmer would be provided with tools that aide the creating of the agent but those should not be the only ones used. The programmer is expected to build upon the existing components and also build new ones new one when the need arises.

Chapter 4

PyPOSH: Implementing the POSH framework in Python

PyPOSH is a agent framework built in Python which supplies POSH action selection mechanism for agents. To create new types of agents in this framework, we need to create the primitives that enable an agent to interface with that environment. These behavior primitives are packaged into behavior modules, implemented in PyPOSH as Python packages. These modules are loaded on the core system and provides a interface to an environment, which can be real, simulated, virtual or otherwise.

An agent in PyPOSH consists of two parts, behaviors and plan representation. The behaviors tell the agent *how* to do something, the plan representation tells the agent *what* to do in specific circumstances. The action selection provided by PyPOSH provides the mechanism to determine *when* to do something. Together, they provide the necessary components to make complex and autonomous agents. An explanation of how agents are built utilizing PyPOSH can be found in Section 5.

We will go into detail on the design of PyPOSH in this chapter. Let us start with an overview.

4.1 An Overview of the PyPOSH Implementation

PyPOSH provides an action selection engine augmented by dynamically loadable behavior libraries to allow agents to interact with the environment. In addition to the basic action selection engine, the package also includes a GUI Agent Manager that create agents and take care of systems functions such as loading and initializing the behavior modules, and providing a graphical interface to control and monitor these agents. It allows the user to view the internal data structure of the agents and serves as a debugging tool. This part of the software is described in Section 4.9.

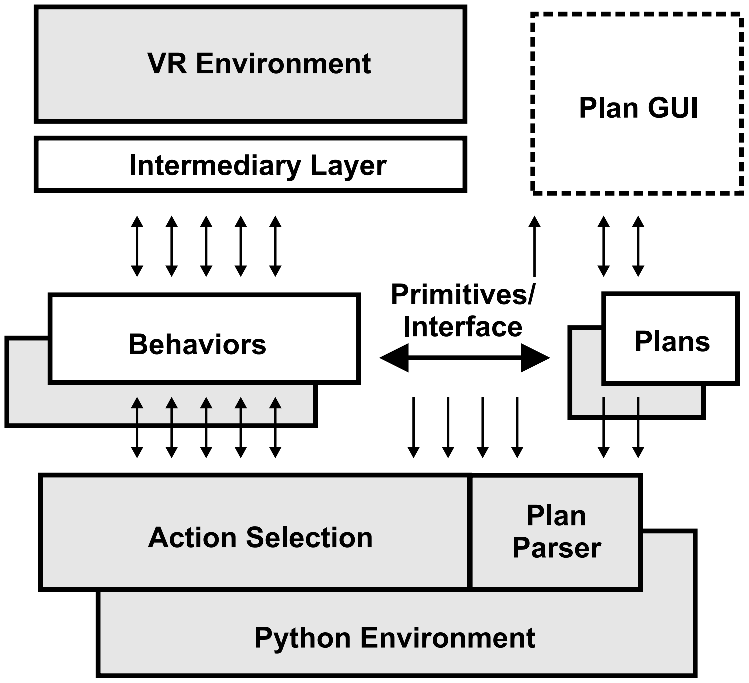


Figure 4.1: This diagram shows PyPOSH components for an agent which interacts thorough a VR environment.

Figure 4.1 shows the high-level configuration of the system for an PyPOSH agent that interacts through a VR environment. PyPOSH runs on top of the Python environment which is available on many popular platforms. The agent itself consists of the action selection mechanism, the loaded behavior module and the POSH plan representations supplied by the plan file. The behaviors interact with the VR environment through their own means.

The PyPOSH action selection paradigm is centered around a real or virtual actor, such as a robot or a player in a game. In PyPOSH, this actor is represented by an object instance of the Agent Class. The instance provides basic services that allow communication between different subsystems and act as the consolidator of different subsystems within the agent by providing a wrapper around them.

An agent is of not much use if it has no interaction with the environment. This environment is provided a dynamically loadable behavior module. This module includes code that may define or simulate a environment, as well as procedures that specifies what the agent can sense, and what actions it can take. For each instance of PyPOSH, only one behavior module can be loaded. For multi-agent simulations, the use of the module allows Agents to communicate with one another and to share a common world. Each Behavior module would define a behavior class. Instances of this class are attached to Agents, and these instances provides bound methods for use by the agent. Additionally, temporary state information can be stored within these instances for use by senses and actuators. Section 4.4 visits the topic in more detail.

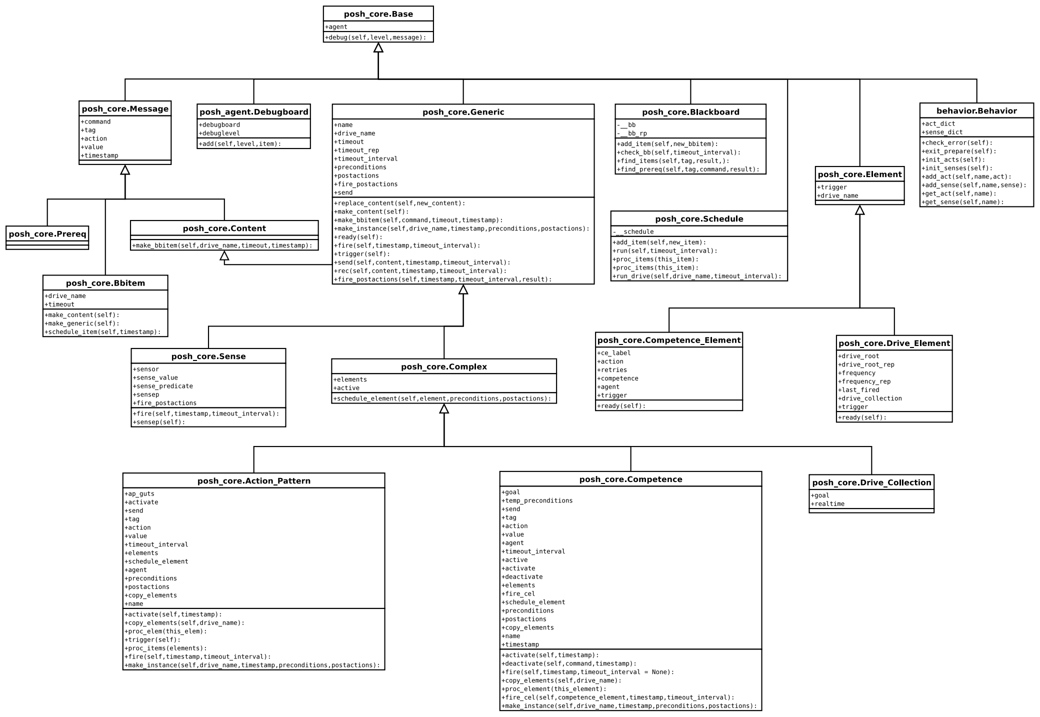


Figure 4.2: An overview of generalization relationships between classes in PyPOSH

Now that we have agents and the primitives needed for them to reflect upon the environment, they need to make the agent do things. The question of when to take actions and what actions to take for an Agent is determined by the plan. This plan is supplied to agents in the form of a planfile, which is read in at the creation of a specific agent. A representation of this plan is stored as objects in a tree data structure within the agent itself. The plan components are linked to the actions and senses provided by the behavior module in this arrangement. The agent make action selection decisions by iterating over the hierarchy. The plan structure, along with what the agent senses determine the actions which should be executed in each particular cycle.

Figure 4.2 illustrates the generalization relationships between the classes in the action selection engine. We will take an in-depth look at each of these components in the following sections.

4.2 The PyPOSH Action Selection Engine

The Action Selection Engine or ASE provides the core mechanism to make decisions. The entire system is implemented through classes. Object instances of these classes are used at run-time. They can be grouped into the following categories -

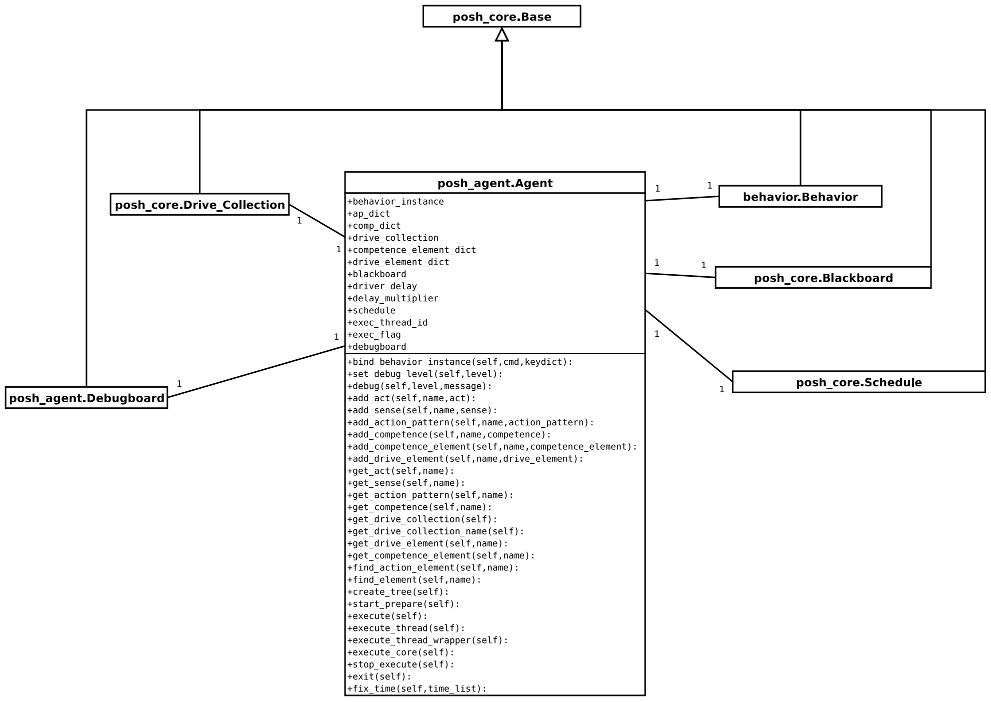

posh_core.Base
posh_core.Drive_Collection
posh_agent.Agent
+behavior_instance
+ap_dict
+comp_dict
+drive_collection
+competence_element_dict
+drive_element_dict
+blackboard
+driver_delay
+delay_multiplier
+schedule
+exec_thread_id
+exec_flag
+debugboard
+bind_behavior_instance(self,cmd,keydict):
+set_debug_level(self,level):
+debug(self,level,message):
+add_act(self,name,act):
+add_sense(self,name,sense):
+add_action_pattern(self,name,action_pattern):
+add_competence(self,name,competence):
+add_competence_element(self,name,competence_element):
+add_drive_element(self,name,drive_element):
+get_act(self,name):
+get_sense(self,name):
+get_action_pattern(self,name):
+get_competence(self,name):
+get_drive_collection(self):
+get_drive_collection_name(self):
+get_drive_element(self,name):
+get_competence_element(self,name):
+find_action_element(self,name):
+find_element(self,name):
+create_tree(self):
+start_prepare(self):
+execute(self):
+execute_thread(self):
+execute_thread_wrapper(self):
+execute_core(self):
+stop_execute(self):
+exit(self):
+fix_time(self,time_list):
behavior.Behavior
posh_core.Blackboard
posh_core.Schedule
posh_agent.Debugboard
1
1
1
1
1
1
1
1
1
1

Figure 4.3: Instances of the agent class collaborate with other class instances to provide services. These relationships are shown on the graph

- The Agent Class
- Agent Services (Debugboard, Blackboard, Schedule)
- POSH Plan Representations (Generic, Sense, Action Pattern, Competence, Competence Elements, etc.)
- POSH Messages (Content, Bbitem, Prereq)
- The Behaviors System (package *modules* and the behavior modules)

The following subsections describes the use of these objects within the ASE and details their relation to the system as a whole.

4.3 The Agent Class

The agent serves as the focal point for all the activities related to a single agent. Services in the agent are typically provided by components which are object instances of other classes attached to the agent at certain attributes. These services are created during agent initialization and include message logging (instance of Debugboard() in Agent.debugboard), a blackboard to assist in action selection (instance of Blackboard() in Agent.blackboard) and the schedule of tasks for selected actions (instance Schedule() in Agent.schedule). Additionally, a special object called the *behavior instance* is attached after the agent object is created. The purpose of this object and how it is attached to the agent is covered in Section 4.4.

After the the *behavior instance* has been attached to the agent, the agent loads the representation of the POSH plan located in the planfile by calling the method *read_file()*. The parser that reads in the planfile is implemented inside that method. In memory, objects representing the plan are created and stored in the agent temporarily. The method *create_tree()* arranges the temporary objects into a hierarchy by connecting together objects and functions with the names used in the planfile representations The final result is a drive collection that serves as the root of the decision hierarchy. All of the objects created to represent the plan have references back to the agent which allow objects to share the blackboard, schedule and debugboard. Our implementation is not the only way to share the data structures of the agent. Dynamic scoping, for example, can be used to provide a similar effect in languages such as Common LISP.

At this point the agent object is ready to execute action selection. Two methods are provided to run start execution. We can run the action selection in the current thread by calling *execute()*. Calling *execute_agent()* starts the action selection in a separate thread.

When the agent is executing the decision cycle in a separate thread, we can use some of the methods bound to the agent instance to control execution. Method *stop_execute()* stops the action selection cycle and *exit()* does the same with the additional side effect of issuing a exit command to the *behavior instance*, effectively killing the representation of the agent in the environment.

4.4 The Behaviors System

Behavior modules are extensions to allow the agent to interact with different external environments and are implemented as loadable Python sub-packages under the package *modules* under the root directory of the PyPOSH distribution. For each execution instance of PyPOSH, only one behavior module can be loaded in memory at any given time (a limitation of the current implementation). However, the user can run separate instances of PyPOSH for different agents as different processes.

The base distribution has two implementation of behaviors, *cookietest* and *poshbot*. Chapter 5 provides detailed information on how they work and the process involved in creating additional behaviors.

The behavior module itself can be thought of as the interface between the agent mind (provided by the ASE) and the environment. Sometimes, the module also provides the simulated environment; the ASE does not differentiate between these uses. To the ASE, the behaviors module is a black box that is used to interact with the environment.

The agent in fact does not interface with the behavior module at the module level. It interfaces with instances of a class present within the module called Behavior. This class provides the essential mechanisms for the ASE to tie in the primitive action and senses provided by the behavior module. Thus, when an agent object is created, an object instance of this class (called a *behavior instance*) must first be bound to the agent before the agent can load in the plan representation from the planfile. This topic is further explored in 5.

Understanding the relationship between the *agent*, the *behavior instance*, and the *behavior module* is very important. The agent object contains the behavior instance, which is provided by the behavior module. The behavior instance is in effect a package of all of the primitives available to the agent. It also stores the temporary state information for use by the senses and actuators if necessary.

The *behavior module* exists as a sub-package or module of *modules*. This module is imported before the creation of agents in the following manner -

```
import modules.some_behavior as behavior
```

However, the Agent Manager GUI does not use the above method to import the behavior since it dynamically detects the installed modules and allows selection of the module to load. The functions that enable this are provided by the package *modules*. This package can return the modules installed in the the system, and also return the full pathname to the plans directory (where the planfile exists) for a specific module.

From now on, all references to *behavior* are references to this particular behavior module. The function *init_world()* is called which in thurn may call other initialization routines. These routines are normally instructions for the creation of remote connections or shared environments.

At this point, the agent can be created. The agent is only a shell after instantiation, with no senses or actions and no plan representation. To populate it, we bind it to a *behavior instance* which is produced by calling *make_behavior*, passing in options if necessary. Now, the agent is ready to load the plan representation and the routine continues as described in Section 4.3.

The behavior instance is in essence a representation of the agent in its environment. The agent can call *get_sense()* and *get_act()* to retrieve primitives for interaction. The use of an agent instance allows state to be stored for use by the primitives. Reactive agents should minimize internal state, and one can see this as a violation of that rule (Brooks, 1991). However, the use of state is necessary in some simulations (*poshbot* for example, needs somewhere to keep information from the server) and is optional in the architecture.

The primitives themselves can be located anywhere. In the included behavior module, all of the senses and acts are methods of the Behaviors class, so that they are bound to a particular instance of Behaviors. Although useful for illustration, this does not need to be the case; behaviors are free to encapsulate additional services as long as the actual sense and act methods are added to the dictionaries. The method for initializing primitives is the methods *init_act()* and *init_senses()*, which uses the methods *add_sense()* and *add_act()* to insert functions into a dictionary object that is attached to the behavior instance. Python treats all functions and methods as first class objects, and this allows easy storage and manipulation of functions in a LISP-like manner.

When a particular action is decided upon by the agent mind, the object instance representing the action would already have a attribute containing the reference to the particular function or method implementing the behavior primitive as defined in the behavior instance. This attribute can be called directly without the need for run-time lookup of primitives.

4.5 The Base Class

Everything which is stored in or kept tracked of by an Agent is derived from the Base class. This class manages the most basic functions of object instances, handling common functions such as initializing the link back to the parent Agent instance, and posting debug messages back to the agent.

Currently the base class fills two roles. First, any derivative class from Base requires a reference back to an Agent instance. Second, utilizing this link, it provides a debug method that allows the these objects to send back messages to the debugboard of the Agent.

4.6 Agent Services: Blackboard, Schedule and Debugboard

As explained above, the Agent instance serves as mostly a container for everything related to that particular agent. It provide services to child objects which are linked back to the parent agent. Most of these services are in turn provided by object instances attached to a specific agent. The following section describes the three main services provided by the agent.

4.6.1 The Blackboard

The backboard is the central log for all decisions made by the action selection process. More details on the workings of the action selection is detailed in Section 4.8. Each item in the blackboard is an instance of the Bbitem class, derived from the Message class described in Section 4.7.2.

Serving as the repository for state during decision making in the agent, elements can check the blackboard for specific items via the tag or command attributes. This proceedure is normally used to check for dependency information for action patterns. The method *find_prereq()* is used for this purpose exclusively, returning a True when the first instance of a match is found.

Performance is of great concern to the blackboard. The blackboard grows depending on two factors - the default timeout, and activity (which depends heavily on the frequency of decision cycles). The time required to search in the list and to trim it are also dependent on the length of the blackboard. Section 6.2 details the efforts in optimizing the blackboard through profiling and other techniques.

4.6.2 The Schedule

The schedule keeps track of the actions to take for the current and the following cycles. An item on the schedule is an instance of the Content class which derives from the Message class. It has functions similar to an Bbitem instance.

Each instance of the schedule contains the Scheduler method. This method runs every cycle and is the workhorse of the action selection mechanism. It processes actions that have all of its preconditions met in addition to canceling requests that have timed out. There are two versions of this scheduler, one runs on all elements, the other runs on only one drive. The latter version is used predominantly in the system. When given a drive name, it runs all scheduled actions related to that drive and trims off elements that have successfully executed. It then returns the number of elements found in that drive. This information is used for adding drives which are ready to run, but are not active and the mechanism for this is described in 4.8.

4.6.3 The Debugboard

The debugboard is the message collection facility for an Agent instance. Messages are emitted by processes that call the debug method on instances derived from the Base class, or by calling the debug board directly. The board also has the concept of debug level, which is an integer that acts as the threshold of acceptance for messages. Processes that wish to send a message to the debugboard would supply a priority level. If the current debuglevel is below the message priority, the message would be ignored. Through this mechanism, we are able to control the detail level of the messages.

All object instances from classes derived from the Base class have reverse references back to the Agent that they are associated with. The Base class also provides a debug method, which calls the *add_debug* method of the debugboard. This allows object instances a unified and simple interface to send logs, results and errors to.

4.7 POSH Plan Representations

The POSH plan is a hierarchy represented by the drive collection at the root. Typically, when the action selection engine is executed, this hierarchy is visited once through a cycle. Each cycle lasts for only a fraction of a second, and the higher the number of cycles run per second, the more reactive the agent is to the environment.

When an Agent instance is created, the POSH plan is loaded from disk through the *read_file()* method. Subsequently, this generates the objects that represent the plan into memory, first as object lists sorted by the type, and finally combined together as complete tree with a root.

There are basically two types of objects needed to represent a POSH plan. As we touched upon in section 2.4, POSH plans consists of three types of complex structures - Competences, Action Patterns and the Drive Collection, which can be thought of a modified competence (although implemented as a separate class). Action Pattern elements are simple, as they do not have additional attributes other than the order in which the elements are stored in. However, Competences and Drive elements have special needs in that each element has its own trigger and other attributes. Subsequently, each of these elements would require its own object instances. These belong to the Elements branch of the class tree.

The Drive Collection contains drives, the top level elements in the heirarcy, which are in turn linked to the competence, action pattern, or primitive they represent. Drives are important in POSH action selection as it provides the Slip-Stack Hierarcy (the SH in POSH). The action selection would only descend into a drive only when its trigger returns true. This allows the system to restrict the complexity of evaluating the hierarcy to a single branch.

Messages are special objects that convey information within the system. For instance, when an action is completed successfully, a 'done' message is sent to the blackboard. This message is stored as a Bbitem. However, the creation of the message involves the use of a Content Object. Both of these objects are Message objects, and they represent state information that is necessary for the operation of the action selection engine. A third type of Message is a Prereq, which is used to store information in the preconditions list of a Generic object. Together, they regulate the firing of action with respect to preconditions. The Prereq instances specifies what needs to be done first, and the Bbitems on the blackboard specifies what has been done already. The result is that for each prerequisite, we check the blackboard to see if the it has been met, if so, fire the action.

4.7.1 Generic Objects and Element Objects

Generic derives from Base and also from the Content class. Instances of the Generic Class acts represents the basic and atomic plan elements within the PyPOSH environment. Objects from this and derived classes are used to wrap around action functions, or used as a representation of an Action Pattern, Competence, Sense or Drive Collection (see Figure

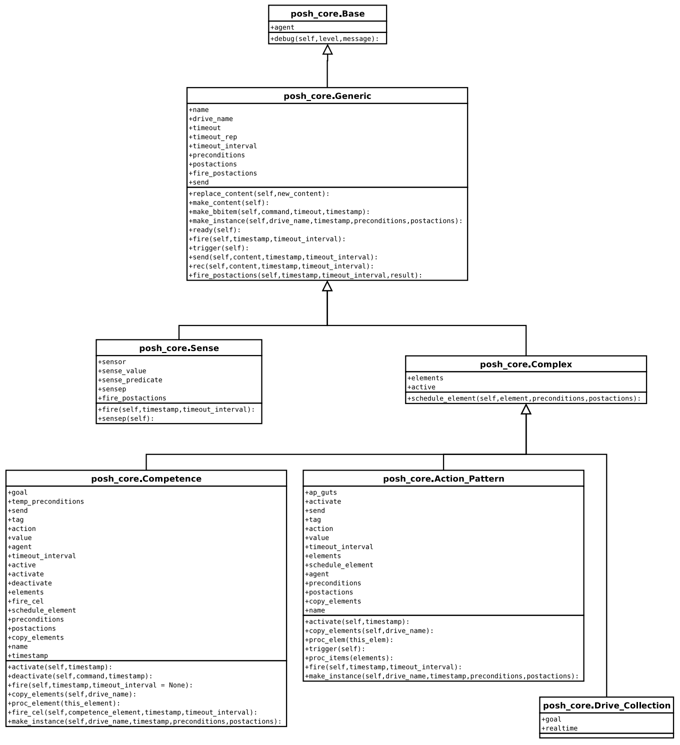

posh_core.Base
+agent
+debug(self,level,message):
posh_core.Generic
+name
+drive_name
+timeout
+timeout_rep
+timeout_interval
+preconditions
+postactions
+fire_postactions
+send
+replace_content(self,new_content):
+make_content(self):
+make_bbitem(self,command,timeout,timestamp):
+make_instance(self,drive_name,timestamp,preconditions,postactions):
+ready(self):
+fire(self,timestamp,timeout_interval):
+trigger(self):
+send(self,content,timestamp,timeout_interval):
+rec(self,content,timestamp,timeout_interval):
+fire_postactions(self,timestamp,timeout_interval,result):
posh_core.Sense
+sensor
+sense_value
+sense_predicate
+sensep
+fire_postactions
+fire(self,timestamp,timeout_interval):
+sensep(self):
posh_core.Complex
+elements
+active
+schedule_element(self,element,preconditions,postactions):
posh_core.Competence
+goal
+temp_preconditions
+send
+tag
+action
+value
+agent
+timeout_interval
+active
+activate
+deactivate
+elements
+fire_cel
+schedule_element
+preconditions
+postactions
+copy_elements
+name
+timestamp
+activate(self,timestamp):
+deactivate(self,command,timestamp):
+fire(self,timestamp,timeout_interval = None):
+copy_elements(self,drive_name):
+proc_element(this_element):
+fire_cel(self,competence_element,timestamp,timeout_interval):
+make_instance(self,drive_name,timestamp,preconditions,postactions):
posh_core.Action_Pattern
+ap_guts
+activate
+send
+tag
+action
+value
+timeout_interval
+elements
+schedule_element
+agent
+preconditions
+postactions
+copy_elements
+name
+activate(self,timestamp):
+copy_elements(self,drive_name):
+proc_elem(this_elem):
+trigger(self):
+proc_items(elements):
+fire(self,timestamp,timeout_interval):
+make_instance(self,drive_name,timestamp,preconditions,postactions):
posh_core.Drive_Collection
+goal
+realtime

Figure 4.4: Generic class inheritance relationships

4.4). The class provides a collection of base attributes and methods that are common to the derived classes and may be used or overridden.

Generic also serves as a wrapper around actions and aggregates when they are put on the schedule. For example each Drive Element contains a Generic instance in its *drive_root* attribute. This Generic instance serves no purpose other than being the wrapper, allowing the scheduling of the intended action or aggregate.

Sense

Sense object are wrappers around a specific sense function or method (the *sensor* / sense primitive) provided by the Behaviors module. Senses are normally invoked as a object within a trigger sequence. The reason that a wrapper is needed is because triggers can contain predicates and values in addition to sensor specified. This allows the output form the trigger function to be compared against a *value* with a predicate. Sense allows the object to store this predicate and value as an attribute in the instance until the trigger method is called. When it is called, it evaluates the sensor and compares it against the value. If the Test is successful, a true is returned. Otherwise the trigger evaluates to false. In case where the value is specified but the predicate is missing, then the predicate is assumed to be equality comparison or ==.

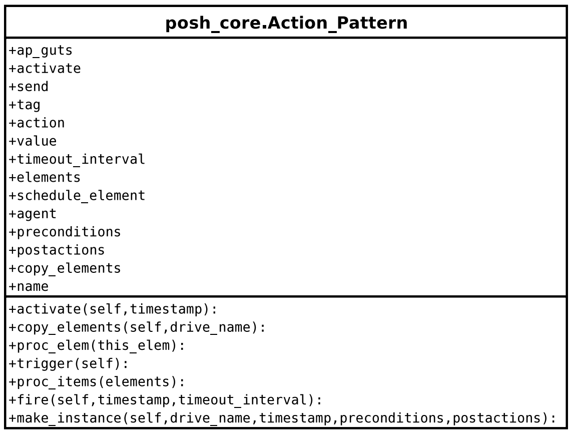


Figure 4.5: The Action Pattern Class

There are triggers which do not contain a value to compare against and the Sense object that wraps around it would not contain value and predicate information. In this case, the exact result from calling the sensor is returned, without and tests performed.

As senses normally wraps around sensors and acts as a trigger for an element, it is called when the element is being traversed and should not end up on the schedule.

Complex

Complex is a virtual class that ties together aggregates in the POSH system. There should be no object instances of this class. Aggregates are objects which can contain multiple child objects. Instances of classes derived from Complex keeps these object in the *elements* attribute of the class, in the form of a list.

The Complex class defines one important method, *schedule_element()*, which is used to put an child object on the schedule. This object can either be a reference to an action, a Competence Element, or an object of any class derived from Generic. When an action (a reference to a behavior method) is passed in to be scheduled, the action is wrapped around a Generic object before it is sent to the schedule. Otherwise, if a Competence Element object is passed in, the object within the action attribute of the Competence Element (Which must be derived from Generic) is copied (by using *make_instance()*) and put on the schedule. If the object passed in is a normal Generic object, then *make_instance()* is called on it, and the result placed on the schedule.

Action Pattern

Action patterns (AP) (Figure 4.5) are simple sequences and one type of an aggregate in the PyPOSH system. They can be thought of as a list of things to do, and each item on the list

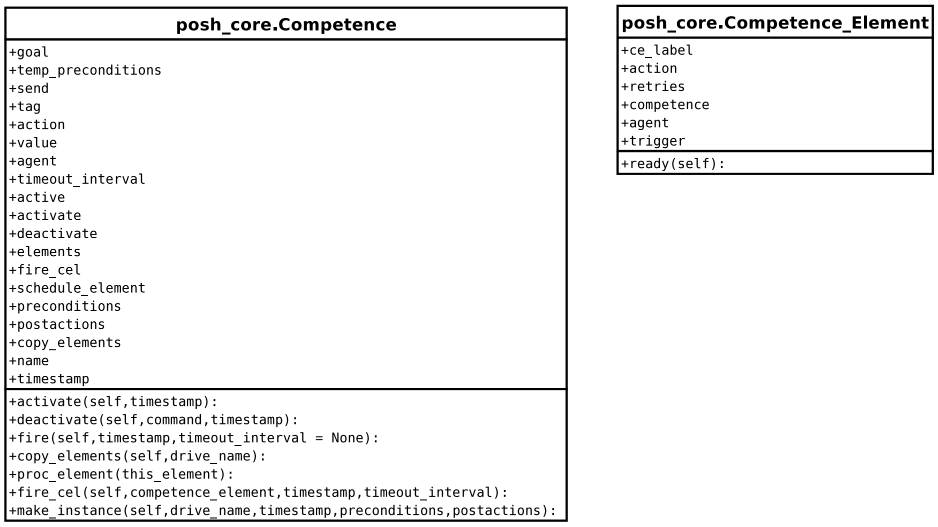

posh_core.Competence
+goal
+temp_preconditions
+send
+tag
+action
+value
+agent
+timeout_interval
+active
+activate
+deactivate
+elements
+fire_cel
+schedule_element
+preconditions
+postactions
+copy_elements
+name
+timestamp
+activate(self,timestamp):
+deactivate(self,command,timestamp):
+fire(self,timestamp,timeout_interval = None):
+copy_elements(self,drive_name):
+proc_element(this_element):
+fire_cel(self,competence_element,timestamp,timeout_interval):
+make_instance(self,drive_name,timestamp,preconditions,postactions):
posh_core.Competence_Element
+ce_label
+action
+retries
+competence
+agent
+trigger
+ready(self):

Figure 4.6: The Competence and Competence Element Classes

can only fire when the previous item has finished. Child elements are kept in the *elements* attribute in the form of a list. Slots in this list can also contain lists which allow items of the same priority to run parallelized.

There are two types of actions which could be applied - *trigger()* and *fire()*. When an AP is used as a trigger sequence (ie. attached to the trigger attribute of a Drive Element or Competence Element), trigger() is called each time the parent is traversed. Objects in the elements attribute are in turn triggered and the combined result combined (with AND logic) and returned. As a result, in any trigger sequence, the failure of any single trigger will return negative for the trigger() call.

The method *fire()* is called with the AP is used as sequence of actions that needs to be put on the schedule. The call puts each object form the elements attribute of the AP on the schedule individually, together with the specific dependency requirements. For example, in a AP with 3 elements α, β, and γ in a list one level deep, the precondition for β is α and the precondition of γ is β. Parallel elements would share the same preconditions, and the following elements would have all of the parallel elements as preconditions.

Competence and Competence Element

Competences are more complex aggregates with child objects (from the class Competence_Element) that are not invoked in sequence, unlike an Action Pattern. Instead, elements are ordered by priority and invoked when a trigger sequence of that child returns True.

The Competence instance itself has a goal, which is represented by a single object of class Competence Element which contains an Action Pattern object in its trigger attribute. This action pattern is used as a trigger sequence, which for identification purposes we will

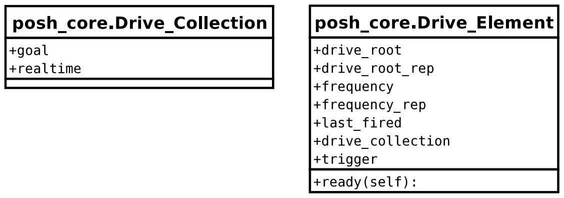


Figure 4.7: The Drive Collection and Drive Element Classes

call the goal trigger. When the *fire()* method is called on the Competence instance, it check this goal first. If all of the elements in the goal trigger return True, then this Competence has achieved its goal and need not be activated. Otherwise, it will descend into its list of children checking that the trigger of the each child individually until one returns True. The competence will then run that child through fire_cel()

The method *fire_cel()* effectively inserts the action attribute of the selected competence element into the schedule and update the status of itself and return "preserve" keeping itself on the schedule. The effect of this is that the competence will be kept on the schedule as its child competence element is on it at the same time. However, the competence instance will sleep until the child has finished or timed out.

Drive Collection and Drive Element

Drives are elements of the Drive Collection. Only one drive collection exists for each agent. The drives system is implemented as a strip down version of the Competence/Competence_Element system. The drive collection itself does not handle the triggering and firing of its own children. Currently this task is done by the *driver()* method of the agent for more control. Subsequently, the Drive_Collection class is mainly a data structure to keep the drives in.

Drives are represented by instances of the Drive_Element class. Each drive has its own trigger as with Competence_Element instances. The difference is that during each cycle, if drive is ready (by its trigger sequence returning True) but has nothing on the schedule, the *driver()* function of the agent notices this and puts the root of the drive on the scheduler.

For more information how the drives system cycles through the object hierarchy, see Section 4.8.

4.7.2 Message Objects:

The Message Class defines structures which carry information on a particular process or event and has three subclasses, Content, Bbitem and Prereq. Each of the three subclasses fills a role in the system, but Content and Bbitem provides similar interfaces while Prereq has a stripped down attribute list for its lesser requirements.

The command attribute indicates the type of message to be sent.

- request
- active
- done
- cancel

The tag is used to identify messages which are sent out by specific operations. This is especially important in determining if a prerequisite has been completed. The action is run if the command is a "request". Normally this attribute will contain a instance of the Generic Object.

Content

Content instances are informational tokens used throughout the system. When an action has been completed for example, a "done" token would be posted on the blackboard of the agent. This is represented by a Bbitem with the command "done" and a tag that is unique to the specific operation.. A Content instance can be return a Bbitem image of itself through make_bbitem() and the reverse is also possible.

Instances of Generic and its derived classes also require these attributes as they require a template when creating separate Content instances to pass around or to generate messages to the blackboard.

Bbitem

The practical difference between a Content instance and Bbitem different deals with the different requirements the system places on the information on the blackboard. First, Bbitem information must be augmented with *drive_name* to identify which drives the specific message is associated with. Secondly, since the Bbitem instance has a finite lifespan on the blackboard, it need to contain a timeout.

Apart from the addition of the two attributes, one important method was added to allow the addition of a Bbitem with the command of "request" directly onto the schedule.

Prereq

A Prereq instance is used exclusively to attach dependency information to the preconditions attribute of object from Generic or any derive class. Used as such, there is only two attributes necessary, which is the command and tag. This information is fed into the *find_prereq()* method of the blackboard to identify preconditions that have been satisfied.

4.8 Execution of Action Selection and the Drives System

When the Agent has all of the POSH plan elements loaded and ready to go, we are ready to start the loop which runs the decision cycle. There are two ways in which the loop is executed. The method *execute()* runs the loop in the current thread of execution. The method *execute_thread()* starts a new light-weight process and runs the loop in a separate thread. This is especially useful when using the Python shell as we can peer into the data structures of the agent in real-time.

What both of these methods do is to call *execute_core()*. This is the center of PyPOSH and where all the different elements come together. The *execute_core()* method has two loops for two different time representations. Real-time agents use the current system time as the basic for timekeeping, while incremented time agents use a counter that is incremented for each decision cycle. However, the difference only affects timestamps, the timeout of messages always use real-time.

Suppose that we have a real-time agent which is ready to run. We call *execute()* on it and the agent starts running. We see the debug messages and notice that the system is running at several hundred cycles per second. Short of running a debugger on it and setting breakpoint everywhere, the system is complex enough that is is almost impossible see the mechanism it used to reach the decision. An attempt is made to better explain the operation in this section.

When *execute_core()* is called. First it checks to see if the agent has the *real_time* attribute set. It is set to True (and this is done by the parse when it sees a RDC in the plan file), so the real-time loop is selected. Now it does several things. First, it make a note of what the current time is. This timestamp is then fed in to check_bb() method of the blackboard. *check_bb()* is a housekeeping function that cleans Bbitem that have timed-out, and also processes requests from the last decision cycle. Once this has been completed, the decision cycle is ready to run.

The decision cycle is executed by the driver() method of the agent. It directly descends into the *elements* list of the drive collection. These drive elements or drives are then tested to see if they are ready by running the trigger sequence. If one is ready, we make a note of the drive name and pass it onto the *run_drive()*method of the schedule.

run_drive() runs all of the items on the schedule that matches the name passed in. It returns the number of items that it found. If it returned 0, then when control is passed back to *driver()*, it will add the drive root to the schedule for the next cycle.

When one decision run is completed, basically we are ready to update to a new timestamp, clean the BB and execute the decision run again. One complete iteration of this is a decision cycle or action selection cycle.

Optionally, by setting the *driver_delay* and *delay_multiplier* attributes of the agent we can instruct the *driver()* method to sleep x amount of time every y cycles. This has the effect of setting hard limits on the number of cycle that can be achieved in each loop.

4.9 Controlling and Launching Agents: An Overview of The PyPOSH Agent Manager

Running on top of the Python VM, PyPOSH uses facilities provided by Python to create light-weight parallel processes allowing multiple agents running concurrently. *Wax*, a thin wrapper around the cross platform wxPython (Dunn, 2003) toolkit was used. The wax release used is distributed by the author under an open source license. wxPython is a Python interface to wxWindows (Smart, 2003), a cross-platform native widget toolkit built in C++. The use of this interface allows the GUI applications to operate as native applications with native interfaces across multiple platforms including different varieties of Unix/X, Microsoft Windows (9x, 2000 and XP) and Macintosh Operating System (8, 9 and X). The only prerequisites for using the Agent Manager is Python and wxPython for the target system. Wax Nowak (2003) is included with the PyPOSH distribution and does not need to be installed globally.

The main purpose of the GUI manager is to provide a generic interface for launching and controlling agents. Developers utilizing the PyPOSH ASE can also create custom applications that launch agents and simulations directly if they so chose to. In this respect, the Agent Manager GUI is not a requirement for using PyPOSH. However, it provides many common features dealing with agent management that also help illustrate the process of interfacing with a Agent instance. New users are encouraged to test the system using the Gui first.

On a proper Python and wxPython installation, executing the **pyposh.py** file in the PyPOSH distribution starts the Agent Manager. When the application has finished loading, the agent launcher window is displayed. Figure 4.8 is an illustration of the main launcher screen. The system at this point has no behavior module loaded. Behavior modules are located as subdirectories of the *modules* directory and are automatically detected by the system. The combination box near the top of the launcher allows the user to select the behavior modules desired. Only one behavior module can be loaded by Agent Manager in one session currently. This is due to behavior modules can also serve as the state foundation for a multi-agent environment, and this introduced a certain amount of complexity in the process. However, there is no theoretical constraint on the ability for the system to flush a behavior module, although every agent executing in the system would have to be killed and *destroy_world()* of the current behavior module called before the new module is loaded.

After a behavior module is selected, the user will load it as the active behavior by pressing the *Load Behavior* button. This action activates the *init_world()* function in the behavior. The main purpose is to allow custom initialization code from the behavior to run. One possible application is to provide environment initialization here. For example, this function

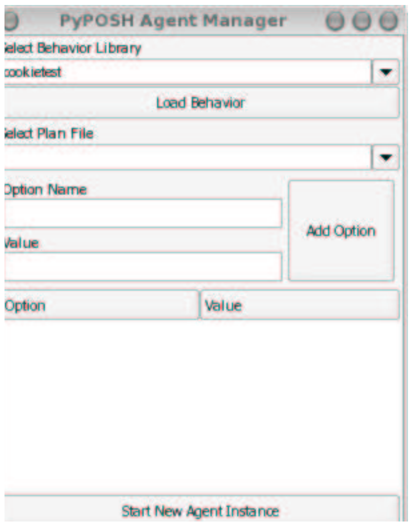


Figure 4.8: The Agent Manger showing the launcher window.

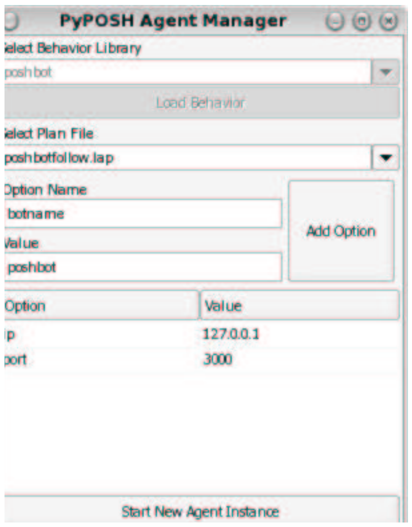
may set up an draw a board on the screen to allow subsequent agents to connect to. Once the behavior is loaded, a list of the LAP plan files located in the *plans* subdirectory of the selected behavior module is listed in the next combination box. The user is required to select a plan from the box before it is possible to create a agent instance.

Some behavior modules may require extra information from the user before the agent can be created successfully. The poshbot module, for example, requires three options to connect to the Gamebots server. This information can be entered in the manner shown in Figure 4.9. Each option consists of a name/value pair and is entered in the respective text field. Clicking on the *Add Option* button creates the option and displays it in the list underneath. Double-clicking on a option stored in the list deletes it.

With the behavior module loaded, the plan file selected, and the options required to launch the agent have been entered, the Agent Manager is now ready to create agents. When new agent instances are created, they are attached to a separate window frame which allows the operator extensive control over the agent. This is done by pressing the *Start New Agent Instance* at the bottom of the agent launcher.

Figure 4.10 shows the agent control panel. This window allows the operator to control the agent through the use of the command buttons located of its top left corner. When this window is first displayed the attached agent has just been instantiated, but nothing else has been done to it. In order to make it run, we need to take a few more steps and instruct the agent to start making decisions. Pressing the *Start Agent* button connects the agent to a new behavior_instance (See Section 4.4), then process the LAP plan file into the in-memory object tree ready for action selection, and finally starts the action selection process.

There are three buttons concerned with debug messages from the debugboard of the agent. *Show Debugboard* makes a copy of the debugboard and displays it in a separate



elect Behavior Library: posh bot

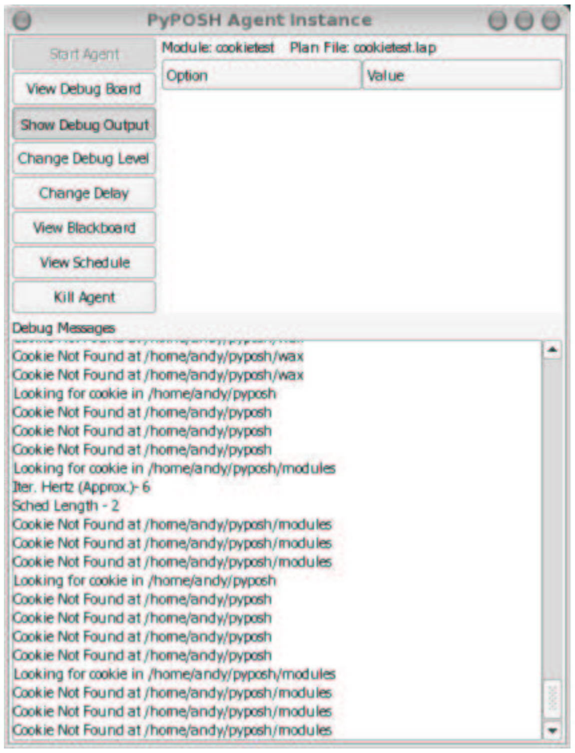
elect Plan File: poshbotfollow.lap

ption Name: botname

alue: poshbot

Option	Value
p	127.0.0.1
port	3000

Figure 4.9: Entering options in the launcher window.



Module: cookietest Plan File: cookietest.lap

Option	Value

Debug Messages

```
Cookie Not Found at /home/andy/pyposh/wax
Cookie Not Found at /home/andy/pyposh/wax
Looking for cookie in /home/andy/pyposh
Cookie Not Found at /home/andy/pyposh
Cookie Not Found at /home/andy/pyposh
Cookie Not Found at /home/andy/pyposh
Looking for cookie in /home/andy/pyposh/modules
Iter. Hertz (Approx.)- 6
Sched Length - 2
Cookie Not Found at /home/andy/pyposh/modules
Cookie Not Found at /home/andy/pyposh/modules
Cookie Not Found at /home/andy/pyposh/modules
Looking for cookie in /home/andy/pyposh
Cookie Not Found at /home/andy/pyposh
Cookie Not Found at /home/andy/pyposh
Cookie Not Found at /home/andy/pyposh
Cookie Not Found at /home/andy/pyposh
Looking for cookie in /home/andy/pyposh/modules
Cookie Not Found at /home/andy/pyposh/modules
Cookie Not Found at /home/andy/pyposh/modules
Cookie Not Found at /home/andy/pyposh/modules
```

Figure 4.10: The agent control panel.

window. The resulting text can be copied from this window to other applications. The *Show Debug Output* option is a toggle and is set to on by default. When on, new messages accepted by the debugboard of the agent are copied to the debug window in the agent control panel. This maybe have induce a serious performance penalty when the volume of debugboard messages are high which may be caused by setting the debug level to a high number, indicating that low-priority debug messages are accepted by the debugboard. In such cases, pressing the button will toggle it to the off state in which no debugboard messages are copied to the debug window.

Change Debug Level allows the user to increase or decrease the amount of messages sent to the debugboard and the debug window. It does this by showing a dialog and asking the user to enter a new debug level for the agent. This debug level is described in Section 4.6.3 and is a filter that throws aways debug messages that are of a lower priority (having a higher number) than the current debug level. This is a very useful mechanism that allows the user to see what the agent is doing. The default debug level is 5 (and is set at the top of **posh_agent.py** in the root directory of the PyPOSH distribution). Interesting levels of detail can be seen at debug levels 7 and 9.

Show Blackboard and *Show Schedule* shows the respective data structures from the agent. It does this in a similar manner as with showing the debugboard, by making a copy of the list and then printing out the string representation of the contents in a separate window.

The feature *Change Delay* allows the operator to set a hard limit on the number of cycles per second of the action selection loop. It achieves the feat by introducing a small amount of delay in each cycle, freeing up the system to do other tasks. The formula for calculating the hard limit on the frequency is $1/t$ where t is the delay in fractions of a second. However, there is a low bound limit on the smallest period in which the system can sleep for. Python on a Linux 2.4 kernel would not sleep for less than 0.01 seconds. Therefore, the delay multiplier was introduced to increase the range of frequency we can limit. The concept is to introduce the delay only every x cycles. This way, we can multiply the frequency obtained by using the delay with this value in the form of $\frac{1}{t}x = f$ where f is the frequency. As this system does not take in account the length of each decision cycle, the frequency approaches the processing limits of the hardware logarithmically. Changes to these value are reflected immediately on the agent and can be seen if the debug level is set to 7 or above.

Kill Agent sends an exit request to the agent, shutting down any connections or threads running in the background. At the default debug level, messages would have been emitted to either the agent debugboard and/or the debug window. The window can now be disposed by closing it through the window manager. Closing the window at any time would invoke the exit procedure and dispose of the window.

Chapter 5

Defining Behavior: Building PyPOSH Agents

The main aim of PyPOSH is to serve as a foundation for creating agile behaviors. In this section, we will learn how these behaviors are constructed. PyPOSH includes a behavior specification, which is essentially a empty behavior template containing all of the components necessary. The module *cookietest* is only a minimal extension of this and illustrate the basic techniques in building behavior modules.

A more complex example is included in the form of the module *poshbot* which involves networked communications with the Gamebots server running as a separate thread. In addition to examining the structure of the module, we will use *poshbot* as an example tutorial in building more complex plans.

Engineering agents in PyPOSH is a two step process. First, a behavior module is created in order to allow the agent to interact with a specific environment in the required capacity. Then, plans organizing these primitives into POSH elements need to be made in order to give direction to the agent. We will explore these topics in the next two section by disassembling the code for the two agents included in the PyPOSH distribution.

Before we begin, let us first review the structure in which behavior modules reside in the distribution.

```
pyposh/
pyposh/modules/
pyposh/modules/\_\_init\_\_.py
pyposh/modules/cookietest
pyposh/modules/cookietest/\_\_init\_\_.py
pyposh/modules/cookietest/cookie.py
pyposh/modules/cookietest/plans/
pyposh/modules/cookietest/plans/cookietest.lap
```

The above shows the directory *pyposh* as the root of the distribution. Underneath *pyposh* is the modules subdirectory. To Python this directory is special (because the presence of the __init__.py file). These are called packages and are loaded as modules. The *modules* package provides several methods that allow the query of packages underneath its own directory (pyposh/modules). These sub-packages are the behaviors available to the system. In the example above, cookietest is the behavior module, and the contents of module *cookietest* resides in pyposh/modules/cookietest/__init__.py. However, this file does not do anything other than import pyposh/modules/cookietest/cookie.py, where the real code for the behavior is at.

5.1 Cookietest: Basic behaviors

Cookietest is a behavior that provides basic functions to allow an agent to look for a file named "cookie" in directories. It provides one sense to tell the agent if it see it cookie in the directory that is being visited, and one act that tell the agent to randomly change to a subdirectory or parent directory. It will not however, change to a directory above the directory in which the agent is run in. In addition to these two primitives, it also provide a helper primitive named *fail*, which always return a false.

The file that contains all of the code for the cookietest behavior is located in pyposh/modules/cookietest/cookie.py. This is the only file imported by pyposh/modules/cookietest/__init__.p
Developers can create multi-file behaviors by making __init__.py import additional files.

Take a look at the source file, the first function is *init_world()* -

```
def init_world():
    pass
```

This function gets called when the module has been loaded successfully. In our cookietest behavior, there is no use for it. However, this might come in useful when we are creating multi-agent simulations in which calling this may initialize a simulation world.

```
def make_behavior(*args, **kw):
    return Behavior(*args, **kw)
```

This next function creates a behavior instance to attach to the agent (see Section 4.3). This function is called a the routine does the binding, typically by the agent launcher. Make sure you return a behavior instance here. Optionally, you can modify the function to produce side effects if necessary.

```
def destroy_world():
    pass
```

This function is called just before the system is about to exit. It is the final chance to cleanup anything left around. Now that we have looked at all the functions in the module, we can dissect the Behavior class which contains most of the interesting routines.

```
class Behavior(Base):
    def __init__(self, agent, *args, **kw):
        Base.__init__(self, *args, **kw)
        self.act_dict = {}
        self.sense_dict = {}
        self.init_acts()
        self.init_senses()
        # These are behavior varibles
        self.base_dir = os.getcwd()
        self.cwd = self.base_dir
```

This is the initialization code for creating the behavior instance. You can see that the constructor first called the *__init__()* of the Base class (which it inherits from). Because this class is derived from Base, we can use the *self.debug()* method, an example of which is available later in this section. It then proceeds to create two empty dictionaries, one for acts and one for senses. After the dictionaries are created, it calls the two init functions and are described in detail below. Right now all we need to know is that these function initialize the dictionaries with the primitives, allowing the agent to access them. The next two variables are specific to the cookietest behavior. The object gets the current working directory of this program and assigns it to two values. We will see how it is used later on.

```
    def init_acts(self):
        self.add_act("change_dir", self.change_dir)

    def init_senses(self):
        self.add_sense("see-cookie", self.see_cookie)
        self.add_sense("fail", lambda : False)

    def add_act(self, name, act):
        self.act_dict[name] = act

    def add_sense(self, name, sense):
        self.sense_dict[name] = sense

    def get_act(self, name):
        if self.act_dict.has_key(name):
            return self.act_dict[name]
        else:
            return None
    def get_sense(self, name):
        if self.sense_dict.has_key(name):
            return self.sense_dict[name]
        else:
            return None
```

These are the actual methods used to populate the dictionary. There is one action primitive in the dictionary with the name of "change_dir", this is bound to the method *self.change_dir()*, we will see later on what this method does. Two senses are then added to the dictionary, one of these is "see-cookie" which is bound to the *see_cookie()* method of the current class, the other is "fail". That latter sense is bound to a nameless (lambda) method that always returns False.

We now see that this behavior provides one action - "change_dir" and two senses - "see-cookie" and "fail". If you have already seen the test plan file included with the PyPOSH distribution, you may already understand how these actions are called.

```
    def check_error(self):
        return 0

    def exit_prepare(self):
        return True
```

The above methods are used by the agent to determine if the behavior is ready, and to tell the agent to prepare to exit. They are not populated in this module, but are useful when the behavior has to connect to external environment through sockets.

```
    def see_cookie(self):
        try:
            os.stat(self.cwd+"/cookie")
        except:
            self.debug(8, "Cookie Not Found at " + self.cwd)
            return False

        self.debug(8, "Found cookie at " + self.cwd)
        return True
```

Looking back at the *init_senses()* method above, we can see that this method is stored as "see-cookie" in the dictionary. Thus, this is the method that the agent will substitute for when it sees a "see-cookie" in the planfile. It is a simple sense that only returns True or False. When there is a file named "cookie" in the directory that we are visiting (self.cwd), then return True. Otherwise return False. Also notice the *self.debug()* method which allow messages to be sent back to the debugboard of the agent. The first argument to this method is a number which tells the debugboard the priority of this message. We suggest a number of 8 or above.

```
    def change_dir(self):
        tmplist = os.listdir(self.cwd)
        dirlist = []

        for x in tmplist:
            if os.path.isdir(x):
                dirlist.append(x)
        if self.cwd != self.base_dir:
            dirlist.append("..")

        # We are at the basedir, but there are no directories to change to.
        if not len(dirlist):
            return False

        result = dirlist[random.randrange(len(dirlist))]
        if result == "..":
            self.cwd = os.path.dirname(self.cwd)
        else:
            self.cwd += "/"+result

        self.debug(8, "Looking for cookie in " + self.cwd)
        return True
```

This is the method bound to the action primitive "change_dir" in the dictionary. This method does several things. First, it gets a list of filenames that exist under the directory we are visiting (self.cwd). Then, it checks to see if each of these entries are a directory. Directories get added to the list of directories. Now we add the parent (..) directory onto the directory list so that it will have a equal change of visiting the parent directory as well, but only if we are not at self.basedir (where we started in the beginning). These directories are to be randomly picked to see which directory we will visit next (ie. change self.cwd to). The method then changes self.cwd to reflect the new directory we are visiting.

Now that we have seen the structure of the module and primitives that it provides, let us take a look at the planfile accompanying this behavior -

```
(
  (C find-cookie (minutes 10) (goal((see-cookie)))
    (elements ((look-for-cookie (trigger ((see-cookie False))) change_dir)))
  )
  (DC life (goal ((see-cookie)))
      (drives
        ((get-cookie (trigger((see-cookie False))) find-cookie))
      )
  )
)
```

The drive collection for the agent is *life* and has the goal of "see-cookie". This goal is tested every cycle and as long it returns False, the agent will continue. This particular agent only has one drive. The drive is called *get-cookie* and the trigger for it to run is that "see-cookie" returning false. Every time this drive runs, it will test "see-cookie" (which in turn calls the *see_cookie()* method of the behavior instance), and if it returns False, then this drive will run.

When the drive does run, it calls *find-cookie*, which is a competence. A competence has its own goals and tests them in the same manner as the drive collection. So when the competence runs, "see-cookie" gets called again. It returns False and the competence continues as the goal has not been achieved.

The only competence element in find-cookie is *look-for-cookie*, which only runs when it tests "see-cookie" again and it returns False. When it does run, it runs "change_dir" which is chained onto the *self.change_dir()* command.

When this agent is executed, all it does is keep changing directories until it finds a file named "cookie". When it does the drive collection *life* ends and the agent will terminate.

5.2 Poshbot: Complex behaviors

Having been through the structure of a minimal agent in the previous section, this section take a change of focus and look at a more complex agent, although from further away. *Poshbot*'s behavior instance behaviors similar to *cookietest*'s although with many more senses and acts.

```
    def init_acts(self):
        self.add_act("stop-bot", self.stop_bot)
        self.add_act("rotate", self.rotate)
        self.add_act("move-player", self.move_player)
        self.add_act("pickup-item", self.pickup_item)
        self.add_act("walk", self.walk)

    def init_senses(self):
        self.add_sense("see-player", self.see_player)
        self.add_sense("see-item", self.see_item)
        self.add_sense("close-to-player", self.close_to_player)
        self.add_sense("hit-object", self.was_hit)
        self.add_sense("fail", lambda : False)
        self.add_sense("succeed", lambda : True)
        self.add_sense("is-rotating", self.is_rotating)
        self.add_sense("is-walking", self.is_walking)
        self.add_sense("is-stuck", self.is_stuck)
```

It is obvious from the initialization routines that this behavior class is much bigger than the previous one. When creating behavior modules, remember to add new primitives to the respective *init* section of the behavior class. It is very easy to make the mistake of adding sense primitives into the act dictionary and vice-versa.

There are more to this behavior than just the additional primitives. One of the things that creating a new behavior instance does in this module is that it creates a new object form a class called bot. This object is the representation of the bot that is connected to the Unreal Tournament sever using the Gamebots interface (Adobbati et al., 2001) and is beyond this scope of what we are doing here.. This object tries its best to get out of the way most of the time to the point that it runs its own communications thread to avoid blocking when waiting for data from the server.

```
def make_behavior(ip, port, botname, agent, *args, **kw):
    bot = Bot_Agent(ip, port, botname)
    b = Behavior(agent = agent)
    b.bind_bot(bot)
    b.bot.connect()
    return b
```

We can see that the bot is attached to the behavior in the sample above. Also notice that *make_behavior* trims off three of the received arguments and uses it to create a new bot instance connected to the server at ip and port supplied. Keep in mind that *make_behavior* can do almost anything as long as the correct behavior module is returned.

There is much more to the module than what is presented in this section including code that does communications with the Gamebots server and maintain local state. For the sake of simplicity, we will not visit them here.

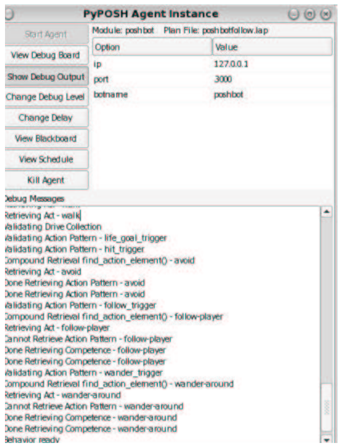
```
(
  (C follow-player (minutes 10) (goal((fail)))
    (elements
      ((close-enough (trigger ((close-to-player))) stop-bot))
      ((move (trigger ((see-player))) move-player))
    )
  )
  (C wander-around (minutes 10) (goal((see-player)))
    (elements
      ((stuck (trigger ((is-stuck))) avoid))
      ((pickup (trigger ((see-item))) pickup-item))
      ((walk-around (trigger ((is-rotating False))) walk))
    )
  )

  (AP avoid (minutes 10) (stop-bot rotate then-walk))

  (C then-walk (minutes 10) (goal((is-walking)))
    (elements
      ((try-walk (trigger ((is-rotating False))) walk))
    )
  )



  (RDC life (goal ((fail)))
      (drives
        ((hit (trigger((hit-object)(is-rotating False))) avoid))
        ((follow (trigger((see-player))) follow-player))
        ((wander (trigger((succeed))) wander-around))
      )
  )
)
```

This is the included planfile that specifies the actions that the robot would take in Unreal Tournament. There are three drives, with *hit* having the highest priority, then *follow* and *wander* having the lowest. Basically, one would expect the robot to wander only if it has not been hit or does not see any other players nearby. When it does see a player, the robot would follow the player. Note that the player moves faster than the bot and so that it is possible for the bot to lose the player. When the bot does lose the player, it would resume the *wander* drive and run around picking up items.

The drive *hit* is the top priority drive, and the result is that whenever the bot hits something, no matter what is is doing, it will initiate the action pattern *avoid* which stops the robot, makes it rotate, and calls the competence *then-walk* which makes sure the bot has finished rotating before it takes a test step.

The *follow-player* competence keeps following the player. When the bot gets too close, the *close-enough* element stops the bot. When it gets the player moves further away, the bot would see that the sense “close-to-player” returns false and it would select the element *move* and fire “move-player” which moves the bot toward the player again.

Figure 5.1: Unreal Tournament Window showing the Poshbot.



Module: poshbot   Plan File: poshbotfollow.lap

Option	Value
ip	127.0.0.1
port	3000
botname	poshbot

Debug Messages

```
Retrieving Act - walk
alidating Drive Collection
alidating Action Pattern - life_goal_trigger
alidating Action Pattern - hit_trigger
ompound Retrieval find_action_element() - avoid
Retrieving Act - avoid
one Retrieving Action Pattern - avoid
one Retrieving Action Pattern - avoid
alidating Action Pattern - follow_trigger
ompound Retrieval find_action_element() - follow-player
Retrieving Act - follow-player
annot Retrieve Action Pattern - follow-player
one Retrieving Competence - follow-player
one Retrieving Competence - follow-player
alidating Action Pattern - wander_trigger
ompound Retrieval find_action_element() - wander-around
Retrieving Act - wander-around
annot Retrieve Action Pattern - wander-around
one Retrieving Competence - wander-around
one Retrieving Competence - wander-around
ehavior ready
```

Figure 5.2: Poshbot Control Window.

The *wander-around* competence operates in a similar fashion. Notice that the goal to *wander-around* is "see-player". When that sense returns True (when a player is seen), the competence immediately stops clearing any stray children on the schedule and allowing *follow-player* to take over. Sometimes when wandering around, the bot will get stuck. When this happens the top priority element *(*stuck) gets triggered and fires the AP *avoid*, otherwise if it sees items that it can pickup, it picks them up. If it is not stuck and sees nothing to pickup, it walks straight hoping that some other element will trigger.

Figures 5.1 and 5.2 show the poshbot in operation.

Chapter 6

Testing and Optimization

6.1 Unit Testing

The PyUnit Unit testing framework included in the Python 2.2 distribution was used to test specific functions and object behaviors in the system. One problem is that the system is complex network of difference object in use, and the attachment of these object proved very difficult for a comprehensive test plan for every single object and object combination with the inclusion of the Agent in the testing of each POSH class introducing additional complexities into testing.

6.2 Profiling and Optimization

Profiling was an important part in constructing the action selection mechanism. The initial implementation performed poorly. Profiling allowed us to locate where the bottlenecks were, which led us to reimplementing some of the performance critical parts of the system. We used the Python's built-in profiler module to perform this task. One point to take note of is that the Python Profiler does not work well with threads. We executed the action selection in the main thread in order to profile the main loop.

The cumulative times showed which procedures were causing the most trouble. In response to the first profile, we changed the method in which the items were stored on the blackboard. Before, items were added onto the rear of the blackboard, and we noticed that when the method *find_prereq()* operated on the blackboard it traversed a large portion of it. This can be explained by the tendency of the method to be called to resolve recent prerequisites, and the traversal started with the first element of the list. By reversing insertions to the head of the blackboard, method performance increased depending on the length of the blackboard.

```
  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.000    0.000   41.980   41.980 profile:0(agent.execute())
       1    0.000    0.000   41.980   41.980 <string>:1(?)
       1    0.300    0.300   41.980   41.980 posh_agent.py:752(execute)
    5001   12.760    0.003   27.990    0.006 posh_core.py:767(check_bb)
 2180697   15.060    0.000   15.230    0.000 posh_core.py:775(proc_bbitems)
    5001    0.710    0.000   13.590    0.003 posh_agent.py:807(driver)
    5001    0.190    0.000    8.030    0.002 posh_core.py:837(run_drive)
   16279    0.310    0.000    7.800    0.000 posh_core.py:843(proc_items)
   31049    0.310    0.000    5.280    0.000 posh_core.py:580(trigger)
   31049    0.540    0.000    4.970    0.000 posh_core.py:583(proc_items)
   32996    0.500    0.000    4.430    0.000 posh_core.py:365(trigger)
   13277    0.220    0.000    3.880    0.000 posh_core.py:119(ready)
   32996    0.550    0.000    3.660    0.000 posh_core.py:389(fire)
   13306    0.120    0.000    3.380    0.000 posh_core.py:321(ready)
    9123    3.260    0.000    3.260    0.000 posh_core.py:757(find_prereq)
   32996    0.680    0.000    2.910    0.000 posh_core.py:401(sensep)
    4502    0.130    0.000    2.720    0.001 posh_core.py:648(fire)
   17772    0.360    0.000    1.980    0.000 posh_core.py:89(ready)
    5001    0.060    0.000    1.890    0.000 poshbot_load.py:437(was_hit)
    5001    1.040    0.000    1.830    0.000 poshbot_load.py:224(was_hit)
    4553    0.090    0.000    1.120    0.000 posh_core.py:330(fire)
    4003    0.260    0.000    1.110    0.000 posh_core.py:671(fire_cel)
```

One major bottleneck remains in the method *check_bb()*. This blackboard method performs several important functions and runs every action selection cycle (which runs many times each second). Typically items on the blackboard have a timeout and *check_bb()* removes these items. It also processes items with the "request" command and adds them to the schedule. Everything else it pretty much leaves alone. The previous code visited every item in the blackboard, checking for timeouts and requests. This was a major performance problem due to at least two comparisons per item with the operation, which is quick in Python, but is especially devastating due to the size of the blackboard (often hundreds or thousands of items) and the frequency that is runs at. However, with some manipulation of the way the data us stored, most of these comparisons are not necessary.

A partial solution was to use two blackboards; one for normal entries, and one for requests. All blackboards exist as lists under the object instance of the Blackboard class so the existing *__bb* list would be augmented by the *__bb__rp* list for requests. The blackboard method *add_bbitem()* would be modified to add the item to the respective list according to the command attribute. Other methods were modified to search and operate on both blackboards. The result was that now, there is no need for *check_bb()* to compare if the command attribute of each item is "request"; it directly processes each item in the *__bb__rp* and send them off to the schedule.

Now that we have solved one-half of the problem, we still have the timeout comparisons for each item which is still costly. The solution to this is achieved by recognizing that time is linear. Thus, we can sort the items by timeout and operate from one end of the list removing items which has timed out until the first item which has not timed out has been encountered. Then we just need to delete all items up to that point. For larger lists, this process can be accomplished even more quickly by implementing some sort of binary search. However, the problem is that sorting the code still requires processing each item and that the sorting itself is a performance bottleneck. There are two options to overcome this problem. First, we can sort the list when items are added on to it. Second, we can use a specific properly of Python with its extremely fast built-in sort mechanism implemented in C. However in order to do this, we need to employ a special technique named the Guttman-Rosler Transform (Guttman and Rosler, 2003) (a special case to the Schwartzian Transform) which presents the data in a format that the internal Python sort implementation can use. It turned out that using the built in search was faster and that route was chosen.

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   21.220   21.220 profile:0(agent.execute())
        1    0.000    0.000   21.220   21.220 <string>:1(?)
        1    0.650    0.650   21.220   21.220 posh_agent.py:805(execute_core)
        1    0.000    0.000   21.220   21.220 posh_agent.py:786(execute)
     5150    1.390    0.000   18.730    0.004 posh_agent.py:894(driver)
     5150    0.440    0.000    9.690    0.002 posh_core.py:946(run_drive)
    16725    0.480    0.000    9.210    0.001 posh_core.py:952(proc_items)
    32145    0.650    0.000    7.820    0.000 posh_core.py:605(trigger)
    32145    0.810    0.000    7.170    0.000 posh_core.py:608(proc_items)
    34055    0.750    0.000    6.360    0.000 posh_core.py:385(trigger)
    13924    0.470    0.000    5.310    0.000 posh_core.py:139(ready)
    34055    1.020    0.000    5.240    0.000 posh_core.py:409(fire)
     4482    0.350    0.000    5.110    0.001 posh_core.py:673(fire)
    34055    1.330    0.000    3.780    0.000 posh_core.py:421(sensep)
    18221    0.600    0.000    3.580    0.000 posh_core.py:109(ready)
     5150    0.170    0.000    1.980    0.000 modules/poshbot/poshbot.py:485(was_hit)
     4273    0.160    0.000    1.960    0.000 posh_core.py:350(fire)
     3834    0.490    0.000    1.860    0.000 posh_core.py:696(fire_cel)
     5150    1.040    0.000    1.810    0.000 modules/poshbot/poshbot.py:274(was_hit)
    38328    0.640    0.000    1.680    0.000 posh_core.py:362(fire_postactions)
     5150    1.080    0.000    1.580    0.000 posh_core.py:825(check_bb)
    12871    0.370    0.000    1.150    0.000 posh_core.py:341(ready)
    39013    0.740    0.000    1.140    0.000 posh_agent.py:101(debug)
     6519    0.230    0.000    1.120    0.000 posh_core.py:319(send)
     4500    0.230    0.000    1.090    0.000 posh_core.py:448(schedule_element)
    27210    0.810    0.000    1.060    0.000 posh_core.py:148(__init__)
```

The above profiler output shows significant improvements to check_bb, taking only 1.5 seconds of time for 18.7 seconds of running the action selection. Before, check_bb took 27.9

out of the 41.9 seconds of running action selection.

The end result is that we have taken out most of the serious performance bottlenecks in the system. There are additional minor improvements that can be made to the code base in order to further optimize the operation, however, these optimizations would yield diminishing returns. Substantial improvements might require a fundamental redesign of the system. Section 7.2 touches upon this topic.

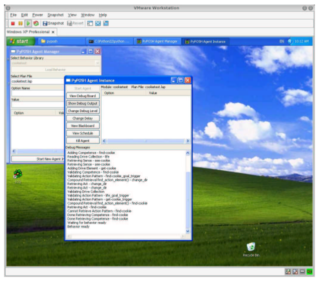
Chapter 7

Analysis

The previous chapters describe a framework for creating reactive agents that can be easily extended through building lightweight component-based behavior modules. The end result fits very well with in initial goals of the project. The process of programming the behaviors included with the PyPOSH distribution were problem-free. As with many data intensive programming projects in scripting, performance (as measured by the frequency of the decision cycle) is the main drawback; acceptable performance, but not stellar and with much room for improvement. We will explore the performance part of the equation in Section 7.2.

Note that the current implementation of PyPOSH as an action selection mechanism only implements the action schedule model as detailed by Bryson (2001). The control loop for running action selection without the use of scheduling has not been implemented yet.

Tested on both Linux and Microsoft Windows platforms (see Figure 7.1 for an illustration of PyPOSH Windows XP), the system performed identically with no apparent indication of any defects due to differences in platform. The stability and maturity of the Python implementations contributed greatly to this. Although there were inconsistencies, such as the different results for some operating system level functions, these were by and large documented clearly. In fact, since the development of the system was done in a Unix-like environment, the author was surprise that running the system on a Windows system worked



Select Plan File: cookietest.lap

Module: cookietest Plan File: cookietest.lap

```
Adding Competence - find-cookie
Reading Drive Collection - life
Retrieving Sense - see-cookie
Retrieving Sense - see-cookie
Adding Drive Element - get-cookie
Validating Competence - find-cookie
Validating Action Pattern - find-cookie_goal_trigger
Compound Retrieval find_action_element() - change_dir
Retrieving Act - change_dir
Retrieving Act - change_dir
Validating Drive Collection
Validating Action Pattern - life_goal_trigger
Validating Action Pattern - get-cookie_trigger
Compound Retrieval find_action_element() - find-cookie
Retrieving Act - find-cookie
Cannot Retrieve Action Pattern - find-cookie
Done Retrieving Competence - find-cookie
Done Retrieving Competence - find-cookie
Waiting for behavior ready
Behavior ready
```

Figure 7.1: Running PyPOSH on Windows XP

without any additional changes to the code. Plan loading has been tested on each platform with good results and no errors.

The development of the *poshbot* behavior module highlights the success of building agents within the architecture. There is centralized representation of the agent in the behavior interface through the behavior instance. This allows us to keep the necessary amount of state information required by the nature of the communications model between he bot and the server. It goes beyond just providing agile behaviors, to providing agile behavior objects that increases the clarity of the code.

Much of the complexity in building an agent resides in solving problems in the low-level behaviors. *Poshbot*, for example, required specific instructions on how to calculate real distance in three dimensional Cartesian space as the Gamebots interface did not provide it. The programming of this in python is trivial. In our implementation, a string containing the coordinate representation is directly converted to real coordinates and passed through a function to find distance. The ease at which this is done allows for prototyping new agents easily.

On the other hand, a fair amount of integration work that was suggested has not been completed. Speech and Story Agents holds much promise for integration with PyPOSH and are visited later on in this chapter.

7.1 The agility of PyPOSH

One of the big questions that is raised when evaluating a framework that is meant to bring agility is just how agile is it. Unfortunately, we do not have definitive answer. One of the aims of the dissertation is to show, through dissecting the procedure in creating the new behaviors, that PyPOSH facilitates agile development of behaviors. Schneider et al. (2001) provides a compelling argument for developing agents in component based scripting. This is an area, that I feel is especially suited to additional work in determine the role agile development in agent development. There have been recent attempts in dealing with this topic in MAS (Knublauch, 2002) and further research on it is warranted, especially from the perspective of AI.

7.2 Improving Performance and Response

The chief measure of performance is the number of decision cycles that the ASE makes in one second. To that measure, performance of the system is acceptable on the whole. When running an moderately complex *poshbot* agent connected to a Unreal Tournament server on the same machine and with the blackboard timeout set at the default 0.5 seconds, the agent averaged to 500 cycles per second on the main test system, running Red Hat Linux 9 on a AMD Athlon 2200+. The much simpler *cookietest* agent fares much better at around 700 to

800 hertz.

At this level, the robot would be practical for lightweight control of real-time agents. Improving this would most likely involve a rewrite of the classes in **posh_core()** in C++ while preserving the Python interface. Python has standard mechanisms for this as many computation intensive modules are built in lower level languages.

Rebuilding POSH while keeping interfaces to Python behavior may take a large amount of work, as the current Python classes and object depend on a large amount of functionality provided by the Python language. The real question is do we need more cycles? I believe that the response for is fine for now. In interactive sessions with *poshbot*, it is evident that the bot reacts smoothly. The only reason that might worth re-thinking implementation is to explore implications for biological plausibility.

7.3 Improving Parsing and Changing Planfile Format

Although the performance of the parser is acceptable, improvement can be made in the form of utilizing established methods such as parse trees. Beyond that, one idea is to have a flexible parser system that allows loading POSH plans in different formats, such as XML. Some insight can be gained from Aycock (1998) which describes a four layer system of parsing in Python. In addition to this, there are packaged parser for Python that can be used (Cotton, 2003) (Patel, 2003).

7.4 Future Work

Future work revolves around two areas, building behaviors that connect to external environments, and extending the PyPOSH models to incorporate new ideas potential improvements. Adapting the system for learning plans is a substantial challenge. Bryson (2001) touched upon learning and the different types of memory required for it to occur. PyPOSH at its current form is strictly an agent framework with action selection. Integration of memory representation in the system allows it to move toward the direction of a cognitive framework. Of particular interest are connectionist storage mechanisms (Pollack, 1990) (Levy, 2002) , and sparse memory models (Kanerva, 1988).

7.4.1 Game and Story Agents

Interaction with virtual characters is one of the areas in which hold a lot promise for complex agents. PyPOSH is already connected to the Gamebots interface to Unreal Tournament through the *poshbot* module. Currently, the behavior provides enough primitives for the bot to avoid objects, wander around and follow players. An likely extension would allow to bot to play competitive through team games such as *capture the flag* (CTF) and other game modes. Another prospective area to consider is the research of human behavior in limited virtual environments such as first person shooters or role playing games. Can we program a reactive agent perform as well as or identical to human players. If so, what is involved? The use of these virtual environments enable the theories to be tested in a setting that is on one hand vastly expanded when compared to limited simulation environments, and on the other hand in levels of complexity much less than that of the real world.

Team agent environments such as Robocup (Noda and Matsubara, 1996) is also prospective environment for PyPOSH. It would allow research into strictly artificial agent based teams strategies.

Alice (Conway et al., 1994) is a virtual world engine based on rapid prototyping ideas similar to those presented in this paper. Based on Python, the aim of Alice is to provide a integrated virtual reality environment where simulations can be created by rapid prototyping. The choice of Python is touted as method to achieve the goal of shortening the development time. Alice can be described as a 3D authoring system written in Python. It allows programmers and users to create interactive 3D actors, objects and environments.

An integration example of Alice and PyPOSH is the creation of a behavior library that allows agents in PyPOSH to interact through an Alice agent. Alice is meants as a general purpose environment for creating interactive story-oriented simulations. Example applications can range from developing simulations and modeling of agent behavior that deals with human interaction to researching spatial and sensory components in cognition within the simulated environment.

7.4.2 Robot Control

Pyro (Python Robotics) (Blank et al., 2003) is described as a library, environment, graphical user interface and set low-level drivers to explore AI and robotics using the Python language. It is a fairly complete open-source robotics development environment that can connect to real world commercial robots in addition to virtual simulations. The project provides a variety of components useful for implementing intelligent agents and for testing new ideas. These components include artificial neural nets (ANN), visual and image processing, a map representation module and evolutionary algorithms. It is a full simulation framework, and can incorporate multiple robot simulators.

Introducing POSH action selection to the framework seems feasible. A behavior-based control module already exists and caters for vertical and horizontal style behaviors. However, integrating the POSH mechanism with this module may not be directly achievable. PyPOSH is quite a large system compared to the other brain modules provided by Pyro. More research on integrating the behaviors is required.

The advantage of integration with Pyro is that it provides a complete robotics environment that allows rapid construction and testing of robots. The completion of a POSH module for Pyro would allow a additional option in the behavioral control in the environment.

Chapter 8

Conclusion

In this dissertation, I have introduced PyPOSH, a framework that allows the creation of agents based on POSH action selection and plan representation principles. PyPOSH allows the creation of reactive actors that interact with an environment through an interface supplied by a dynamically loadable Python module. Engineering new agents is a two step process involving the creation of a behavior module to handle interaction between the action selection mechanism and the environment, and creating POSH plans that utilizes this interface.

Design documentation for the system is found in Chapter 4, where the implementation is described component by component. A limited form of the user manual for running the Agent Manager is found in Section 4.9.

Secondary to the contribution of PyPOSH, this dissertation also includes a review on creating and controlling complex agents and a short survey of recent reactive control architectures in Chapter 2.

Included with the PyPOSH framework are two sets of behavior modules. One is a minimal test module that provides basic behaviors to allow testing of the framework. The other is a simulated bot that interfaces with a AI research platform built on a commercially available game engine. This behavior provides a possible starting point for research involving POSH agents and team strategy or human interaction in a virtual environment.

A design review of the included modules and information on the creation of new behaviors can be found in Chapter 5. The limited walk-through introduces the steps needed to create a new behavior module for the framework and serves as a introduction to writing POSH plans. Regrettably, additional information on behavior decomposition (see Bryson, 2001) is not included due to time and space constraints.

Chapter 7 contains an analysis of the project and lists out possible extensions to this dissertation in the form of building additional behaviors and extending POSH.

Agile methods and component based programming are helping programmers become more productive by changing the way how code is written. PyPOSH support behavior development through these techniques. Currently, designing behaviors is a fundamental part of creating practical reactive agents, and this is not likely to change until substantial further developments are made. By making the process more productive, this project hopes to increase the speed in which intelligence can be deployed.

Bibliography

Adobbati, R., Marshall, A. N., Scholer, A., and Tejada, S. (2001). Gamebots: A 3d virtual world test-bed for multi-agent research. In *In Proceedings of the Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, Montreal, Canada.

Aleksander, I. (1996). *Impossible Minds.* Imperial College Press, London.

Aycock, J. (1998). Compiling little languages in python.

Baldassarre, G. (2001a). A modular neural-network model of the basal ganglia's role in learning and selecting motor behaviours. In *Proceedings of the Fourth International Conference on Cognitive Modeling - ICCM-2001*, pages 37–42.

Baldassarre, G. (2001b). A planning modular neural-network robot for asynchronous multi-goal navigation tasks. In *Proceedings of the 2001 Fourth European Workshop on Advanced Mobile Robots - EUROBOT-2001*, pages 223–230.

Bangley, D. (2003). The great computer language shootout. http://www.bagley.org/~doug/shootout/.

Beazley, D. and Lomdahl, P. (1997). Feeding a large scale physics application to python.

Beazley, D. M. (1996). SWIG: an easy to use tool for integrating scripting languages with C and C++. In Association, U., editor, *4th Annual Tcl/Tk Workshop '96, July 10–13, 1996. Monterey, CA*, pages 129–139, Berkeley, CA, USA. USENIX.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change.* Addison Wesley.

Blank, D., Meeden, L., and Kumar, D. (2003). Python robotics: An environment for exploring robotics. In *SIGSCE03.*

Boehm, B. W. (1981). *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ.

Boehm, B. W. (1999). Making RAD work for your project. *IEEE Computer*, 32(3):113–114.

Brooks, R. (1986). A robust layered control system for a mobile robot.

Brooks, R. A. (1991). Intelligence without representation. Number 47 in Artificial Intelligence, pages 139–159.

Bryson, J. (2000a). Hierarchy and sequence vs. full parallelism in action selection. In *The Sixth International Conference on the Simulation of Adaptive Behavior (SAB2000).*

Bryson, J. J. (2000b). The study of sequential and hierarchical organisation of behaviour via artificial mechanisms of action selection. Master's thesis, University of Edinburgh.

Bryson, J. J. (2001). *Intelligence by Design: Principles of Modulatiry and Coordination for Engineering Complex Adaptive Agents.* PhD thesis, Massachusetts Institute of Technology.

Conway, M., Pausch, R., Gossweiler, R., and Burnette, T. (1994). Alice: A rapid prototyping system for building virtual environments. In *Proceedings of ACM CHI'94 Conference on Human Factors in Computing Systems*, volume 2, pages 295–296.

Cotton, S. (2003). Pylr: Fast lr parsing in python. http://starship.python.net/crew/scott/PyLR.html.

Davis, D. and Parashar, M. (2002). Latency performance of soap implementations. *IEEE Cluster Computing and the Grid.*

d'Inverno, M., Kinny, D., Luck, M., and Wooldridge, M. (1997). A formal specification of dMARS. In *Agent Theories, Architectures, and Languages*, pages 155–176.

Dunn, R. (2003). wxpython: wxwindows for python. http://www.wxpython.org.

Gat, E. (1997). On three-layer architectures.

Georgatos, F. (2002). *How applicable is Python as first computer language for teaching programming in a pre-university educational environment, from a teacher's point of view.* PhD thesis, Universiteit van Amsterdam.

Guttman, U. and Rosler, L. (2003). A fresh look at efficient perl sorting. http://www.sysarch.com/perl/sort_paper.html.

Huang, X., Alleva, F., Hon, H.-W., Hwang, M.-Y., and Rosenfeld, R. (1993). The SPHINX-II speech recognition system: an overview. *Computer Speech and Language*, 7(2):137–148.

Hugunin, J. (1997). Python and Java: The best of both worlds.

Humphrys, M. (1997). *Action selection methods using reinforcement learning.* PhD thesis, University of Cambridge.

Kanerva, P. (1988). Sparse distributed memory.

Kim, K.-J. and Cho, S.-B. (2001). Robot action selection for higher behaviors with cam-brain modules. In *Proceedings of the 32nd ISR (International Symposium on Robotics).*

Knublauch, H. (2002). Extreme programming of multi-agent systems.

Levy, S. D. (2002). *Infinite RAAM: Initial Explorations into a Fractal Basis for Cognition.* PhD thesis, Brandeis University, Waltham, Massachusetts.

Lloyd, D. (1989). *Simple Minds*. MIT Press, Cambridge, MA.

Maes, P. (1990). How to do the right thing. *Connection Science Journal, Special Issue on Hybrid Systems*, 1.

McCarthy, J. (2003). What is artificial intelligence? http://www-formal.stanford.edu/jmc/whatisai/whatisai.html.

McHugh, J. (2003). Why a.i. is brain-dead. *Wired*, 11.08.

Minsky, M. (1986). *The Society of Mind*. Simon and Schuster, New York. A loosley-coupled theory of intelligence based on the model of a distributed society of interacting 'experts'.

Minsky, M. (2001). The emotion machine (draft).

Newell, A. (1990). *Unified theories of cognition*. Harvard University Press, Cambridge, MA.

Newell, A. and Simon, H. (1963). GPS: A program that simulates human thought. In Feigenbaum, E. and Feldman, J., editors, *Computers and Thought*. McGraw-Hill, New York. Probably the most influential paper establishing the "physical symbol system" computational paradigm as the underlying model of intelligence used by AI.

Nilsson, N. (1984). Shakey the robot. Tech Note 323, AI Center, SRI International.

Nilsson, N. J. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158.

Noda, I. and Matsubara, H. (1996). Soccer server and researches on multi-agent systems.

Nowak, H. R. (2003). Wax: A high-level gui layer sitting on top of wxpython. http://zephyrfalcon.org/labs/.

Ousterhout, J. K. (1998). Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30.

Patel, A. (2003). Yapps: Yet another python parser system. http://theory.stanford.edu/˜amitp/Yapps/.

Pollack, J. (1990). Recursive distributed representations. Technical report, Ohio State University.

Post, E. (1943). Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65:197–215.

Prechelt, L. (2000a). An empirical comparison of c, c++, java, perl, python, rexx, and tcl. *IEEE Computer*, 33(10):23–29.

Prechelt, L. (2000b). An empirical comparison of c, c++, java, perl, python, rexx, and tcl for a search/string-processing program.

Repenning, A. (1993). Agentsheets: A tool for building domain-oriented dynamic.

Ritter, H. and Kohonen, T. (1989). Self-organizing semantic maps. *Biol. Cybern.*, 61:241–254.

Rosenblatt, J. K. and Payton, D. W. (1989). A fine-grained alternative to the subsumption architecture for mobile robot control. In *Proc of the IEEE Int. Conf. on Neural Networks*, volume 2, pages 317–324, Washington, DC. IEEE Press.

Schneider, J.-G., Lumpe, M., and Nierstrasz, O. (2001). Agent coordination via scripting languages. In Omicini, A., Zambonelli, F., Klusch, M., and Tolksdorf, R., editors, *Coordination of Internet Agents*, pages 153–175. Springer-Verlag.

Schneider, J.-G. and Nierstrasz, O. (1999). Components, scripts and glue. In Barroca, L., Hall, J., and Hall, P., editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer-Verlag.

Selfridge, O. (1959). Pandemonium: A paradigm for learning. In *Proceedings of Symposium on the Mechanization of Thought Processes.*

Smart, J. (2003). wxwindows: Cross platform gui library. http://www.wxwindows.org.

Turing, A. (1950). Computing machinery and intelligence. *Mind*, 49:433–460.

Tyrell, T. (1993). The use of hierarchies for action selection. In *Proceedings of the second international conference on From animals to animats 2 : simulation of adaptive behavior*, pages 138–147. MIT Press.

Tyrrell, T. (1993). *Computational Mechanisms for Action Selection.* PhD thesis, University of Edinburgh.

van Rossum, G. (2003). Python programming language. http://www.python.org.

Veloso, M., Carbonell, J., Pérez, A., Borrajo, D., Fink, E., and Blythe, J. (1995). Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120.

Winograd, T. (1972). *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language.* PhD thesis, Massachusetts Institute of Technology.

Winston, P. (1992). *Artificial Intelligence.* Addison-Wesley, Reading, MA. 3rd ed.

Wooldridge, M. and Jennings, N. R. (1994). Intelligent agents: Theory and practice.

Appendix A

Code Listing