MGLAIR: A Multimodal Cognitive Agent Architecture

by

Jonathan P. Bona

August 14, 2013

A dissertation submitted to the
Faculty of the Graduate School of the
State University of New York at Buffalo
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering

Acknowledgements

Many thanks to my advisor, Dr. Stuart C. Shapiro, for his guidance, support, and patience over the years. His vast knowledge and experience have been tremendous assets to me. It is an honor to be part of the SNePS project. I will evaluate my efforts in any undertaking with the the question "What would Dr. Shapiro say?"

Dr. William J. Rapaport is an extremely careful reader and thinker with an encyclopedic knowledge, whose red pen has taught me much about writing and thinking clearly. I thank him for his advice on my dissertation and on many other topics.

I very much enjoyed interdisciplinary work on various projects with Prof. Josephine Anstey, and thank her for her input throughout the dissertation process.

I've received much support and encouragement from Alan Ruttenberg, for which I am grateful. He's very persistent at inspiring people.

Debra Burhans has given me a lot of good advice since I first took her Introductory Artificial Intelligence course at Canisius College over a decade ago. Ken Regan gave me a B+ in Theory of Computation, but it was worth it. I've enjoyed every interaction I've ever had with John Corcoran.

It's impossible to thank every friend and acquaintance who's helped in some way over the last several years. It was a pleasure working with my friends and fellow SNeRG members, including Mike Kandefer, Dan Schlegel, and Patrice Seyed.

Albert Goldfain has been a friend, mentor, and inspiration to me. I was very sorry when Mike Prentice packed up his *joie de vivre* and took it to Boston. I will miss meeting for weekly "disser-date" work sessions with Paul Heider.

Thanks to my family for believing I could do it – whatever it is they thought I was doing.

I'm very lucky to have Marta Cieślak in my life. Thanks for all the love, patience, and encouragement.

Contents

Acknowledgements **iii**

Abstract **xi**

1 Introduction **1**

2 Background **7**

2.1 Cognitive Architectures . . . 7

2.1.1 Defining *Cognitive Architecture* . . . 7

2.1.2 Overview . . . 9

2.1.3 Architectures . . . 10

2.1.4 Cognitive Architecture Properties and Evaluation Criteria . . . 13

2.2 Previous Work on GLAIR and GLAIR Agents . . . 15

3 Modality in MGLAIR **17**

3.1 Introduction . . . 17

3.2 Modality Properties . . . 19

3.2.1 Modality Name . . . 19

3.2.2 Modality Type . . . 19

3.2.3 KL Predicates . . . 20

3.2.4 Data Channel . . . 20

3.2.5 Modality Buffer . . . 20

3.2.6 Conscious Access Permissions . . . 21

3.2.7 Focus Permissions . . . 21

3.2.8 Conflict Handling Specification . . . 21

3.2.9 Description . . . 21

3.2.10 Relations to Other Modalities . . . 22

3.3 Modality Definitions . . . 22

3.4 Sensing . . . 23

3.5 Acting . . . 24

3.6 Single-use . . . 24

3.7 Modality Buffers . . . 25

3.8 Modality Focus . . . 26

3.9 Conscious Access to Modalities . . . 27

3.10 Relations between Modalities . . . 28

3.11 Platform Independence . . . 29

3.12 Modality Granularity . . . 29

4 Modality Mechanisms **31**
4.1 Becoming Aware of Percepts: Integration with the KL . . . 31
4.1.1 Modality and Architecture Overview . . . 31
4.1.2 Single Modality Example . . . 32
4.2 Perceptual Functions . . . 34
4.2.1 Linking Knowledge to Modalities . . . 35
4.3 Perceptual Buffers . . . 36
Adding PML structures to the perceptual buffer . . . 37
Processing PML structures from the perceptual buffer . . . 37
4.4 Act impulses in efferent modalities . . . 38
4.4.1 Efferent Modality Buffers . . . 39
4.5 Regulating Modality Focus . . . 40
4.5.1 Motivation . . . 41
4.5.2 Focus . . . 42
4.5.3 Effects of Changing Focus . . . 43
4.5.4 Measuring Effects of Changing Focus . . . 44
4.5.5 Focus Future Work and Conclusions . . . 47
Focus without Explicit Knowledge of Modalities . . . 47
Focus and Fatiguing . . . 47
Conclusions . . . 48

5 Sensing and Acting in Modalities **49**
5.1 Effectiveness of Actions . . . 49
5.1.1 Effects of Acts . . . 50
5.1.2 Act Completion . . . 51
5.2 Connecting Modalities . . . 52
5.2.1 Proprioceptive Sense . . . 53
5.2.2 Vestibular Sense . . . 53
5.2.3 Knowledge of Modalities' Connections . . . 53
Ignore the Connection at the Knowledge Layer . . . 54
Making Connections Explicit at Knowledge Layer . . . 55
Inferring Connections from Background Knowledge . . . 55
5.2.4 Interrupting Actions . . . 56
5.3 Passive and Active Sensing . . . 57
5.4 Modality Registers . . . 58
5.4.1 Managing Modality Registers . . . 59
5.5 Modality Examples . . . 60
5.5.1 NXT Agent . . . 60
5.5.2 Afferent Range Sensor Modality . . . 61
5.5.3 Efferent Grabber Modality . . . 64
5.5.4 Locomotion & Proprioception . . . 67
5.5.5 Active Vision in Delivery Agent . . . 67
5.6 Conclusions . . . 68

6 MGLAIR Implementation **71**
6.1 Introduction . . . 71
6.1.1 Across Layers . . . 71
6.1.2 Typical Use . . . 72
Using Efferent Modalities . . . 73
Using Afferent Modalities . . . 73
6.1.3 Typical Use Summary . . . 74
6.2 Modality Specifications . . . 75

6.2.1 Modality Classes . . . 75
Direct Subclasses of `Modality` . . . 77
Modality Subclasses vs Instances . . . 78
6.2.2 Defining Modalities . . . 79
6.2.3 Storing and Retrieving Modalities . . . 80
Modalities Hash . . . 80
Modality Functions . . . 81
6.3 Knowledge Layer . . . 81
6.4 PMLa . . . 82
PMLa Primitive Actions . . . 82
Focus Actions . . . 83
6.5 PMLs . . . 83
PMLs:believe . . . 83
6.6 PMLb . . . 84
6.6.1 PMLb Sense Handler Functions . . . 84
6.6.2 PMLb Action Functions . . . 85
6.6.3 PMLb:execute . . . 85
Blocking Act Impulses . . . 85
6.6.4 Modality Data Channels . . . 87
6.6.5 Modality Buffers . . . 88
6.6.6 Buffer CLOS Classes . . . 89
6.6.7 Afferent Sense Handlers . . . 89
6.6.8 Modality Registers . . . 90
Afferent . . . 90
Efferent . . . 91
6.6.9 Focus . . . 91
6.7 PMLc . . . 92
6.7.1 Connecting and Managing Modality Data Connections . . . 93
`ModalityManager` . . . 93
`ModalityConnector` . . . 93
6.7.2 PMLc Interface . . . 94
6.8 Persisting Modalities . . . 95

7 MGLAIR Agents **97**
7.1 NXT Agent . . . 97
7.1.1 Platform Details . . . 97
7.1.2 NXT Agent Embodiment . . . 98
7.1.3 Agent Modalities . . . 98
Modality Definitions . . . 100
7.1.4 PMLa . . . 101
7.1.5 PMLs . . . 102
7.1.6 PMLb . . . 103
Efferent . . . 103
Afferent . . . 104
7.1.7 PMLc . . . 104
7.1.8 SAL . . . 106
Acting in The SAL . . . 106
Sensing in the SAL . . . 107
Sensing Acting: Proprioception in the SAL . . . 107
Webcam-based Vision . . . 109
Simulated Vision . . . 109
7.1.9 Knowledge Layer . . . 110

7.2 CREATE Agent 111
7.2.1 Vestibular Sense 112
7.3 Greenfoot Agent 113
7.3.1 Agent Overview 114
7.3.2 Agent Modalities 114
7.3.3 Active and Passive Vision 115
7.3.4 Modality Definitions 115
7.3.5 PMLa 116
7.3.6 PMLs 117
7.3.7 PMLb 117
7.3.8 PMLc 118
7.3.9 Agent SAL 120
7.4 Conclusions 121

8 Discussion **123**
8.1 Possibilities for Future Work 124
8.1.1 Connectionist Extensions 124
8.1.2 Focus & Attention 125
8.1.3 Integration with SNePS Concurrent Reasoning 125
8.1.4 Inter-layer Communication 125
8.1.5 Decay of PML structures 126
8.1.6 Conclusion 126

References **129**

A Implementation Details of MGLAIR Modalities and Mechanisms **135**
A.1 MGLAIR in Common Lisp 135
A.1.1 `mods.cl` 136
A.1.2 `modality/modality-classes.cl` 141
A.1.3 `modality/modality-data-channel.cl` 143
A.1.4 `modality/modality-focus.cl` 145
A.1.5 `modality/modality-buffer.cl` 147
A.2 Python-based PMLc Implementation 150
A.2.1 `PMLc.py` 150
A.2.2 `ConfigReaders.py` 152
A.3 Java PMLc Implementation 154
A.3.1 `ModalityConnector` 154
A.3.2 `ModalityManager` 157
A.3.3 `Modality Configuration Readers` 158

B Implementation Details for Python-NXT Agent **161**
B.1 Knowledge Layer 161
B.2 Modality Definitions 164
B.3 PMLa 166
B.4 PMLb 167
B.5 PMLs 168
B.6 Python PMLc and SAL 170
B.6.1 NXTBot Class 170
B.6.2 NXT Sensor & Motor Classes 173

C Implementation Details for Greenfoot Delivery Agent **177**
C.1 Knowledge Layer Excerpt 177
C.2 Modality Definitions 179
C.3 PMLa 181
C.4 PMLb 182
C.5 PMLs 183
C.6 Agent PMLc 184
C.7 SAL 187

Abstract

MGLAIR (multimodal, grounded, layered, architecture with integrated reasoning), is a cognitive architecture for embodied computational cognitive agents that includes a model of modality and an account of multimodal perception and action. It provides these capabilities as part of a cognitively plausible architecture without necessarily mimicking the implementation of the human brain. MGLAIR can be viewed as a cognitive science investigation into the structures and processes sufficient for this type of cognitive functionality. It is also an engineering project aimed at producing embodied computational agents that effectively carry out tasks involving concurrent multimodal perception and action.

As a theory of multimodal perception and action, MGLAIR presents a model for modalities and mechanisms governing their use in concurrent multimodal sensing and acting.

MGLAIR is a layered architecture. Conscious acting, planning, and reasoning take place within the Knowledge Layer (KL), which is comprised of SNePS and its subsystems. The Sensori-Actuator Layer (SAL) is embodiment-specific and includes low-level controls for the agents sensory and motor capabilities. The Perceptuo-Motor Layer (PML) connects the mind (KL) to the body (SAL), grounding conscious symbolic representations through perceptual structures. The PML is itself divided into sub-layers. Perception involves the flow of sense data up through the layers from the SAL through the PML, and its transformation via a gradation of abstractions into consciously available percepts at the knowledge layer. Acting involves the flow of impulses down from the conscious knowledge layer through the PML and into the SAL. Actions are decomposed along the way into low-level motor commands.

A modality in MGLAIR is a limited resource corresponding to a single afferent or efferent capability of the agent's embodiment. Each modality implements only a limited number of related activities. Modality mechanisms govern the use of modalities and their integration with the rest of the agent: reasoning, planning, etc. These mechanisms determine the behavior of afferent modalities when an event or state of affairs in the world impinges on the parts of an agent's body that can detect it. The raw sense data is processed within the relevant modality and converted into percepts that are made consciously accessible to the agent. How this is achieved depends on the nature of the modality — what properties it possesses — as well as on configurable mechanisms shared by all afferent modalities. Corresponding mechanisms for efferent modalities determine how conscious actions performed by the agent result in the operation of its effectors.

This work includes a software implementation of the architecture, which has been used to instantiate embodied agents that sense and act simultaneously using MGLAIR modalities.

Chapter 1

Introduction

Humans and other animals — naturally occurring cognitive agents — continually sense and act using multiple afferent and efferent modalities concurrently, often without any conscious effort to do so. That is, we talk and walk and listen and look about, etc. without thinking about how we're managing to perform these tasks simultaneously. Arguably, this is possible because the separate perceptual and motor modalities, and the mechanisms to manage their resources and integrate their use with our conscious experience, are parts of the underlying architecture of our minds – our cognitive architecture.

MGLAIR (multimodal, grounded, layered, architecture with integrated reasoning), the subject of this dissertation, is a cognitive architecture for embodied computational cognitive agents that provides an account of multimodal perception and action. It is worth noting at the outset that MGLAIR is concerned with providing these capabilities and functionality as part of a cognitively plausible architecture, rather than with mimicking their implementation in nature. Though this work takes inspiration from human psychology and biology, I make no claims about, or attempts to replicate, the structure of the human brain. This approach is consistent with previous work on the GLAIR architecture and cognitive agents motivated by what is called "Computational Philosophy" in [Shapiro, 1992]: the computational understanding and implementation of human-level intelligent behavior.

This work can be viewed as a cognitive science investigation into the structures and processes sufficient for this type of cognitive functionality, seeking to understand multimodal perception and action, and to produce a model and a testable implementation that reflects that understanding. On the other hand, this effort can also be viewed as a pragmatically-motivated engineering project aimed at producing useful artifacts: robots and other computational agents that are more effective at carrying out their intended tasks than are similar

agents that lack such multimodal capabilities. These two major goals are not in conflict. Rather, they are complementary: we possess these capabilities because they are useful, and many of the useful tasks we would like computational cognitive agents to perform on our behalf are things that humans can do; instantiating and using agents on an architecture that claims to have a cognitively plausible model of some phenomenon is a convenient way of testing that model.

MGLAIR is a theory about multimodal perception and action. As such, it presents a model of modality, and gives an account of mechanisms that govern their use and facilitate concurrent multimodal sensing and acting by an embodied cognitive computational agent. MGLAIR is *also* an implementation in software of the theoretical architecture, which allows one to build, test and use MGLAIR agents. I try to keep descriptions of the theory and the implementation as separate as possible. Some elements of the theory are, of course, influenced by computational considerations. The software implementation can be viewed as another way of expressing the theory, as well as being an artifact that allows us to make direct use of it.

Chapter 2 has more to say about what I and others mean by ”cognitive architecture,“ but for now it is sufficient to say that MGLAIR is a cognitive architecture by means of its providing a specification of a system on which models of cognition - cognitive computational agents - may be instantiated.

Chapter 2 also has more to say about what is meant by ”embodied,“ and why this is an important feature in cognitive agents. For now, suffice it to say that embodiment facilitates grounding agents' conceptual knowledge and other abstractions in *a* world (not necessarily *the* world: I include agents embodied in simulated bodies within simulated worlds).

As an architecture for embodied agents, MGLAIR can also be considered as a cognitive robotics project. Cognitive robotics aims to bridge the gap between high-level reasoning, planning, etc, and the low-level issues that have typically been the subject of work in robotics [Lakemeyer et al., 2010].

There are many other cognitive/agent architectures, some of which are discussed in Chapter 2. MGLAIR differs from existing architectures, including its predecessor, GLAIR [Hexmoor et al., 1993b] [Shapiro and Bona, 2010], by providing a unique model of *modality*, and mechanisms to integrate multimodal perception and action with conscious thought, reasoning, and acting.

That is, the MGLAIR architecture provides a specification for agents that can act and sense in, and reason about, multiple modalities concurrently. It provides a general model of modality that specifies what a modality is, what its properties are, and how those properties influence its use within an agent.

As discussed in more detail in Chapter 3, an MGLAIR modality is a limited resource corresponding to a single afferent or efferent capability of the agent's embodiment. Each modality implements only a limited

number of related activities, and each can be engaged in doing only one thing at a time – where "doing a thing“ is used loosely here for both actions in efferent modalities and sensation/perception in afferent. Each modality possesses a data channel through which sensory data and act impulses pass between an agent's body and its mind.

The architecture also specifies modality mechanisms, separate from the modalities themselves, that govern the use of modalities and their integration with the rest of the agent. These mechanisms determine the behavior of afferent modalities when occurrences or states of affairs in the world impinge on the parts of an agent's body that can sense them. The raw sense data is processed within the relevant modality and converted into percepts that are made consciously accessible to the agent. How this is achieved depends on the nature of the modality - what properties it possesses - as well as on modality mechanisms, some of which are general and shared by all afferent modalities, and some of which may be specific to the particular modality. Corresponding mechanisms for efferent modalities determine how conscious actions undertaken by the agent are converted into, e.g. movement of its body.

MGLAIR is a layered architecture: conscious acting, planning, and reasoning take place within the Knowledge Layer (KL) and its subsystems. The Sensori-Actuator Layer (SAL) is embodiment-specific and includes low-level controls for the agent's sensory and motor capabilities. The Perceptuo-Motor Layer (PML) connects the mind (KL) to the body (SAL), grounding conscious symbolic representations through perceptual structures. The PML is itself divided into sub-layers. Perception involves the flow of sense data up through the layers from the SAL through the PML, and its transformation via a gradation of abstractions into consciously-available percepts at the knowledge layer. Acting involves the flow of impulses down from the conscious knowledge layer through the PML and into the SAL. Acts are decomposed along the way into low-level motor commands, etc. **Figure 1.1** shows the layers of the architecture, with vertical arrows representing efferent and afferent modalities spanning the layers.

In addition to mechanisms for integrating modality percepts and actions with agents' conscious minds, MGLAIR also provides flexible mechanisms for regulating the flow of sense data and act impulses within modalities. This is accomplished through the interplay of two separate components of the model: what I call modality focus, which determines for each modality how much of the agent's available load that modality will consume relative to the others; and modality buffers, which can be adjusted to determine what happens to rapidly-arriving sense data to prevent it from "backing up" – which, if not handled in some way, would result in very strange perceptual experiences for the agent. Modality focus also allows agents to consciously act to prioritize certain types percepts over others, which is of particular use when an agent is engaged in an

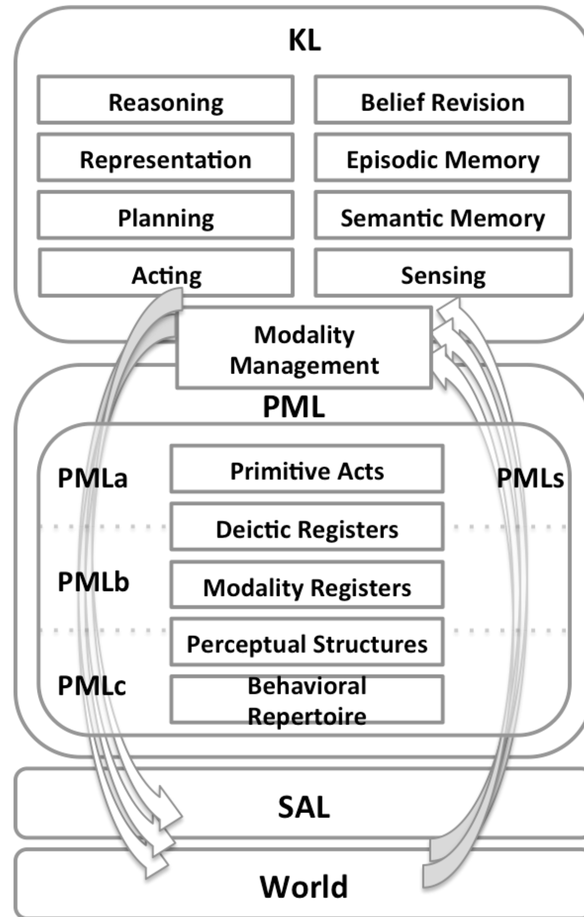

Figure 1.1: MGLAIR

activity for which certain objects or occurrences in the world are of greater relevance than others. Chapter 4 describes these components and other issues related to regulating the flow of data within modalities.

This abstract model and specification are accompanied by a software framework for MGLAIR, discussed more in Chapter 6, on which actual MGLAIR agents are instantiated. MGLAIR's knowledge layer is implemented using the SNePS knowledge representation, reasoning, and acting system. The conscious contents of an agent's mind - its beliefs, policies, plans, goals, etc - take the form of symbolic representations operated on by SNePS' subsystems, include its inference and acting systems.

The implementation of an agent's SAL is necessarily dependent on its embodiment. The PML is independent of the implementation of both the agent's body and its mind. For some MGLAIR agents, the software parts of the KL, PML, and SAL are implemented using different programming languages, and even executed on different physical machines. The architecture facilitates this sort of heterogeneity by specifying interfaces between the layers that establish how each layer communicates with those adjacent to it.

Modalities in MGLAIR are separate modules, handled individually and concurrently by processes operating on each of them to effect their integration with the agent's conscious mind. These include processes to manage modality buffers, convert sense data into percepts, decompose act impulses, and so on, without interfering internally with each other. These modality mechanisms are discussed in detail in Chapter 4.

Two MGLAIR modalities may have a relation between them that captures a causal, logical, or other type of relation that holds between them in the world. A simple example is the relation between an efferent locomotive modality and an afferent proprioceptive modality that allows the agent to sense when its locomotive capacity is in use. Relations between modalities are discussed in more detail in Chapters 3 and 4.

Implemented example agents discussed in Chapter 7 make use of this *proprioception-for* relation. MGLAIR does not attempt to enumerate all the possible relations between different types of modalities, but provides a model for agent designers to establish such relations. Chapter 7 also discusses an implemented agent with the ability to learn about the connection between one of its efferent and one of its afferent modalities. Though MGLAIR does not contain a general model of this kind of learning, the example provides the agent with a way to discover something about its own perceptual capabilities and to make the connection to one of its acting modalities. Chapter 7 presents implemented MGLAIR agents, including an agent embodied in a hardware robot with several afferent and efferent modalities, the concurrent use of which is crucial to its navigating and operating in the world.

Chapter 2

Background

2.1 Cognitive Architectures

2.1.1 Defining *Cognitive Architecture*

There are at least as many definitions of *cognitive architecture* as there are instances of cognitive architectures. There are quite a few of these including 3T, 4D/RCS, ACT-R, AIS, APEX, ART, BECCA, biSoar, CERA-CRANIUM, CIRCA, Chrest, Clarion, CogAff, COGNET, CogPrime, CoJACK, DeStin, Emile, Entropy Reduction Engine, Epic, FORR, GLAIR, GMU-BICA, HOS, HTM, Icarus, Leabra, LIDA, MIDAS, NARS, OMAR, Pogamut, Polyscheme, Prodigy, PRS, Remote Agent Architecture, RCS, SAL, SAMPLE, Soar, Tosca, and Ymir [Langley et al., 2009] [Samsonovich et al., 2010] and [Pew and Mavor, 1998].

Below I discuss some of these, including representative examples of traditional *symbolic* architectures, *connectionist* architectures, and *hybrid* architectures that combine properties of the first two categories.

The common element tying together what we consider to be *cognitive architectures* is a specification of a system in terms of its parts, their functionality, and the integration of and interactions between distinct parts, such that the whole is sufficient instantiate cognition, or *mind*.

John Anderson, creator of ACT-R [Anderson, 1996] defines *cognitive architecture* as:

> ...a specification of the structure of the brain at a level of abstraction that explains how it achieves the function of the mind [Anderson, 2007, p. 7].

This definition reflects Anderson's (and ACT-R's) approach to cognition, which is focused on creating models that resemble the human brain, with modules that correspond to functional units in the brain and

that process information in a similar manner.

John E. Laird, co-creator of Soar (along with Paul Rosenbloom and Alan Newell) defines a *cognitive architecture* as:

> ...the fixed infrastructure that supports the acquisition and use of knowledge. ... A cognitive architecture consists of:
>
> - memories for storing knowledge
> - processing units that extract, select, combine, and store knowledge
> - languages for representing the knowledge that is stored and processed [Laird, 2008, p. 2]

Ron Sun, creator of the hybrid neural network-based architecture, CLARION, defines it as follows:

> A cognitive architecture provides a concrete framework for more detailed modeling of cognitive phenomena, through specifying essential structures, divisions of modules, relations between modules, and a variety of other aspects [Sun, 2004, p. 341].

An early paper on GLAIR as an *embodied agent architecture* [Hexmoor et al., 1993a] characterizes *cognitive architecture* as a type of architecture "for understanding/modeling behaviors of an anthropomorphic agent", noting that cognitive architectures are concerned with "the relationships that exist among the structures of memory, reasoning abilities, intelligent behavior, and mental states and experiences" and often ignore the body to focus narrowly on modeling the mind.

While this last point remains true of some work in the field, it is now more common for mainstream cognitive architectures to be used as platforms to implement embodied cognitive *agents* capable of acting and interacting in the (real or simulated) world [Best and Lebiere, 2006] [Rickel and Johnson, 2000].

I define *cognitive architecture* as

> A *cognitive architecture* is a high-level specification of a system in terms of its parts, their functionality, and the interactions between those parts such that the system may be used as a platform on which models of cognition may be implemented and tested.

In particular, I am concerned with the type of cognitive architecture in which models take the form of cognitive *agents* capable not only of intelligent "thought", but also of embodied intelligent action in a world. The focus of this dissertation is on providing such an architecture that specifically addresses the nature of *modality* and *multimodal perception and action*.

This view of *cognitive architecture* is consistent with the goals of the SNePS project:

> The long term goal of the SNePS Research Group is to understand the nature of intelligent cognitive processes by developing and experimenting with computational cognitive agents that are able to use and understand natural language, reason, act, and solve problems in a wide variety of domains [The SNePS Research Group, 2010].

2.1.2 Overview

Cognitive architectures can be divided into *symbolic* architectures that roughly stick to the Physical Symbol System Hypothesis, modeling cognition using a symbolic language of thought, and *connectionist* architectures that use neural networks and related models and techniques from machine learning. This distinction is becoming increasingly difficult to make as traditional symbolic architectures add components that are inspired by the connectionist approach. The usual result is a *hybrid* cognitive architecture. Some architectures are designed from the ground up to be hybrids. This section discusses these distinctions and evaluates several architectures that fall within these categories.

Symbolic cognitive architectures model cognition (memory, planning, etc) primarily in terms of the manipulation of symbols in formal systems. These include semantic networks, logics, production systems, and frames [Langley et al., 2009]. Early cognitive architectures were all symbolic, in line with Allen Newell and Herbert Simon's *Physical Symbol System Hypothesis*:

> A physical symbol system has the necessary and sufficient means for general intelligent action [Newell and Simon, 1976].

Artificial intelligence work that takes symbol manipulation as sufficient to realize human-level intelligence have been characterized as *GOFAI* (Good Old-Fashioned Artificial Intelligence) [Haugeland, 1989].

SNePS, which forms the knowledge layer of GLAIR and MGLAIR is a part of this tradition, combining logical, frame-based and network-based representations and inference methods [Shapiro and The SNePS Implementation Group, 2007].

In *Unified Theories of Cognition*, [Newell, 1994] Alan Newell argues for the development of theories that account for all of cognitive behavior with a "single set of mechanisms." Central to this project are problem solving and decision making, which had already been treated (c.f. [Newell et al., 1959]), but also perception and motor behavior, which were then absent from many accounts of cognition. Newell presents Soar as an exemplar unified theory for what he terms the *human cognitive architecture*. The Soar architecture, which is still under active development and discussed more below, organized its account of cognition into *central cognition* handling memory, planning, reasoning, etc, and perceptual and motor mechanisms [Newell, 1994].

Because the world usually doesn't present itself as symbols, purely symbolic systems are not well-suited to the task of implementing embodied cognitive agents that sense and act in the world.

Connectionist architectures, inspired by the low-level structure of the brain [Rumelhart and McClelland, 1986], combine many simple elements performing very simple operations, into e.g., *artificial neural networks*, which are organized into interconnected layers of simple simulated neurons. The connectionist approach is

well-suited to low-level control, pattern recognition and learning tasks, but not suited for logical inference, problem solving and planning, conceptual reasoning, and other areas that symbolic systems dominate [Nilsson, 2007].

Many *hybrid* architectures now use a combination of these approaches, combining a symbolic model of central higher cognition with neural networks and other connectionist techniques like reinforcement learning [Barto et al., 1981] to handle some learning tasks, or to more closely model the implementation in the brain of perceptual and motor processing [Taatgen and Anderson, 2010].

Alan Turing, in addressing the possibility of thinking machines, proposed the Imitation Game [Turing, 1950], versions of which have come to be popularly known as the Turing Test. Strong AI skeptics like John Searle have argued against the possibility of a mere symbol-manipulating device instantiating thought [Searle et al., 1980]. Turing seems to have anticipated this, writing that the best way to get a thinking machine is to give it sense organs and teach it to use natural language, i.e. to embody it in the world [Turing, 1950].

George Lakoff in [Lakoff, 1987] argues against what he calls the "traditional" view in cognitive science, which includes the notion that thought is disembodied and abstract symbol manipulation. He argues that human thought and conceptual systems are rooted in bodily experience. The central role of *embodiment* in the functioning of the mind, and as a way of grounding symbols at the core of cognitive systems, is now widely accepted [Harnad, 1990], [Wilson, 2002], [Barsalou, 2010]. This is handled in GLAIR and MGLAIR by aligning symbols in the knowledge layer with modality-specific sub-symbolic structures in the perceptuo motor layer [Shapiro and Ismail, 2001].

Ron Sun's *Desiderata for Cognitive Architectures* [Sun, 2004] include *modularity* as a key feature. MGLAIR exhibits functional modularity in its separation of the architecture into layers, and domain-specific modularity [Fodor, 1983] by establishing separate modalities for different kinds of perceptual and motor processing. There is debate over the extent to which perception (e.g. visual perception) is cognitively penetrable. One view is that cognition influences perception only in allocating attentional resources and in recognition after low-level early visual processing [Pylyshyn, 1999]. Aside from allowing the agent to consciously act to regulate its modality focus, MGLAIR's afferent modalities currently do not support top-down influence of the agent's cognition on its perceptual processing.

2.1.3 Architectures

Soar [Laird et al., 1987] has its roots in traditional, symbolic production systems but has been extended to include modules for semantic and episodic learning and memory, a reinforcement learning module, and more.

Recent extensions to Soar also include short-term and long-term visual memory to store and manipulate visual percepts [Laird, 2008]. Soar does not seem to address the sort of generalized multimodal perception and action included as part of MGLAIR, and recent work to extend Soar has focused on unrelated topics. However, work to incorporate other modalities is listed as a potential area for future research in [Derbinsky and Laird, 2010].

The *ACT-R* (Adaptive Control of Thought—Rational) architecture [Anderson, 1996] is based on a traditional production system, with condition-action pairs at the center of its reasoning capabilities. Initially, work on ACT-R concentrated on symbolic central cognition to the exclusion of perception and action [Anderson et al., 2004]. More recently, it was extended, with ACT-R/PM (ACT-R perceptual-motor) to include perceptual and motor modules for vision, audition, speech and motor control inspired by those in EPIC [Byrne, 2001], [Anderson, 2007]. Modularity is a key feature of ACT-R. It divides processing among distinct modules (perceptual, motor, declarative memory, etc) operating in parallel, each of which is connected to the central, procedural, production system via buffers. Each of these modules is supposed to correspond to a distinct region of the human brain. ACT-R supports sub-symbolic learning, as well as the adoption of new productions as the result of a production compilation learning mechanism [Anderson et al., 2004]. ACT-R can be considered to handle multimodal processing, in the sense that many of the built-in information processing modules connected to the central production system correspond to commonly-used perceptual and efferent modalities for embodied agents (e.g. *manual*, *visual*, *aural*, *vocal*) [Anderson et al., 2007]. However, unlike in MGLAIR, these processing modules are fixed in number and in nature, and altering them would require modifications to the architecture itself.

The *bi-Soar* architecture [Kurup and Chandrasekaran, 2007] is based on Soar and extends its (symbolic, production-based) representation capabilities by adding diagrammatic representations of visual data using the DRS (diagrammatic representation system) [Chandrasekaran et al., 2005]. This allows for bi-modal cognitive states and problem states, and perception and action that make use of both symbolic and diagrammatic representations. bi-Soar does not include a model for general multimodal perception and action. Its notion of modalities does not include support for multiple modalities that extends beyond the bi-modal integration of symbolic and diagrammatic representation and reasoning.

CERA (Conscious and Emotional Reasoning Architecture) [Arrabales et al., 2009] is a layered architecture organized into the *CORE*, *instantiation*, and *physical* layers, which roughly correspond to the *KL*, *PML*, and *SAL* in MGLAIR. Its sensory processing model is based on preprocessing to generate atomic "single" percepts based on raw sensor data from a single modality at the lowest level, and on the composition of single

percepts into more complex percepts. Percepts from a single modality can be combined to form multimodal percepts. For instance, sonar and visual percepts may be combined to form a single percept corresponding to an object of interest in an agent's field of vision. As with other architectures in this list, the selection of available modalities within CERA is seems to be fixed. Descriptions of the project focus on the perceptual integration of three sensory modalities in particular for the purposes of spatial reasoning: sonar, contact, and visual.

EPIC (Executive-Process/Interactive Control) [Kieras and Meyer, 1997] is an architecture designed specifically for modeling multimodal embodied tasks. It consists of a central, production-system-based "cognitive processor" surrounded by modules for short- and long-term memory and specialized processing, including a fixed limited set of sensory and motor module processors (*auditory*, *visual*, *tactile*, *ocular motor*, *vocal motor*, and *manual motor*) for handling perceptuo-motor tasks. Each of these processes data appropriate to its modality using its own dedicated working memory and interfaces with the working-memory module of the cognitive processor. The cognitive processor can use production rules simultaneously in parallel threads. EPIC ties its multimodal capabilities to dedicated sensory and motor processor modules with very specific features: the visual processor, for instance, models a particular kind of foveal vision with built-in assumptions about the size and other properties of the retina. Visual pattern recognition to convert physical visual data (i.e. arrays of pixels) is not performed within EPIC. That is, the architecture's eye processor takes as input structures representing the properties of objects it is to perceive, but does not dictate how those are produced.

CLARION (Connectionist Learning with Adaptive Rule Induction ON-line) is a *hybrid* architecture, mixing explicit (symbolic) and implicit (connectionist) representations within its subsystems. CLARION divides its reasoning capabilities among four main subsystems/modules: *action-centered*, *non-action-centered*, *motivational*, and *metacognitive* [Sun, 2007].

DUAL is a uniquely hybrid multi-agent system made up of many simple components ("agents") operating in parallel as part of a network [1] This design is inspired by, and bears a resemblance to Marvin Minsky's proposed *Society of Mind* [Minsky, 1988]. Each DUAL agent has both symbolic (frame-based) and connectionist parts handling declarative and procedural knowledge, and contextual relevance respectively. Excitatory links among the agents are used to spread activations; these links also have symbolic representations associated with them [Kokinov, 1994].

BECCA (Brain-Emulating Cognition and Control Architecture) [Rohrer, 2011] is an example of a purely

[1] DUAL agents are not considered to be *cognitive* agents; they are individually much too simple.

connectionist architecture. It combines unsupervised feature creation and reinforcement learning-based feature learning. BECCA has been applied to tasks like visual tracking, robotic arm control, and hand-eye coordination. [Rohrer, 2012]. BECCA aims to become a general reinforcement learning agent. It would be interesting to adopt elements of the BECCA approach as lower-level parts of perceptual and motor modalities in an architecture like MGLAIR. There is precedent for this kind of hybridism in the GLAIR family, for instance in the use of reinforcement learning to learn routine actions [Hexmoor, 1995].

2.1.4 Cognitive Architecture Properties and Evaluation Criteria

[Langley et al., 2009] presents properties, capabilities, and criteria for evaluation of cognitive architectures. This section discusses MGLAIR in terms of some of the elements in this framework.

- *Representation of Knowledge*: The knowledge layer is implemented in the SNePS knowledge representation, reasoning, and acting system. Implementations of the other layers (especially the SAL, which is specific to an agent's embodiment) vary from one agent to the next.

- *Mixture of Representational Formalisms*: SNePS itself supports multiple formalisms because of its triune nature as simultaneously logic-based, frame-based, and network-based.

- *Support for Meta-knowledge* - MGLAIR directly supports meta-knowledge and meta-cognition, including straightforward representation of propositions about propositions due to SNePS' use of term-based logic. This obviates the use of work-arounds such as a separate *holds* predicate, which is commonly used to allow propositions about propositions in first-order logics. By including in the knowledge layer a term that refers to the agent itself, MGLAIR agents are able to represent and reason about themselves and perceive their own actions as sensory information reaching the KL through the PML.

  By allowing an agent to act while simultaneously sensing the effects of its actions on the state of its body, for instance, by the use of proprioceptive modalities, and by allowing agents to have conscious access to (some) modalities, MGLAIR provides consciously-accessible feedback that can be used by agents to reason effectively about their own actions.

- *Declarative and Procedural Representations*: MGLAIR's acting system contains procedural representations in the form of policies, actions and acts, as well as declarative propositions about acts. Whether it is considered as logic-based, frame-based, or network-based, the representation used for other types of knowledge in the KL is unarguably declarative.

- *Structured/Hierarchical Organization of Knowledge*: SNePS uses an inherently structured organization of knowledge. Its term-based predicate logic representation allows for "nested" proposition-valued terms that refer to other terms, including other terms that denote propositions. SNePS is often used to represent hierarchical information, including subclass/superclass hierarchies, parthood and other mereological relations, and similar information used in ontological reasoning.

- *Generality*: MGLAIR is a general-purpose architecture. Historically, GLAIR has been used to implement agents that operate in a wide variety of environments and domains and perform diverse tasks. For example, agents (some with rudimentary multimodal processing capabilities) have been created to: act in dramatic performances (these include robotic agents acting in the real world, and simulated agents controlling avatars in virtual environments) [Shapiro et al., 2005]; model early human mathematical cognition [Goldfain, 2008]; reason about threats in the cyber security domain [Kandefer et al., 2007]; and many more. Chapter 7 presents example MGLAIR agents that use very different embodiments, and operate in different (real and simulated environments) environments.

- *Versatility*: An advantage of MGLAIR's multi-layered architecture is that it allows agents' minds to be moved between similar but distinct embodiments, environments, and contexts without major alterations. It is possible to develop and test a MGLAIR agent in a simulated environment before moving the agent to a hardware robot in the real world (or in some cases to another simulation). MGLAIR's treatment of modality places no restriction on the number or type of afferent or efferent modalities an agent can use, and modalities can be defined and added, or removed from the agent with relative ease when the environment or the agent's embodiment require it. An MGLAIR modality's properties are specified in its definition, which allows for a range of different configurations based on modalities' buffer properties, focus-handling, conflict-handling, etc.

- *Integration of Percepts*: Perception is crucial for MGLAIR as an architecture for *embodied* cognitive agents. One significant aspect of perception is the integration of percepts from different modalities into a coherent experience. MGLAIR's general model of modality facilitates the addition and integration of new types of perceptual modalities, and provides built-in mechanisms for managing the transformation and transmission of raw sense as it crosses the layers of the architecture.

- *Allocation of Perceptual Resources*: Part of the task of integrating perception involves deciding how to allocate perceptual resources, a behavior that [Langley et al., 2009] characterizes as *attention*. MGLAIR's model of focus deals specifically with the regulation of an agent's computational load and

its allocation to individual sensory modalities. I view this model of focus as a necessary building block to a full model of attention.

2.2 Previous Work on GLAIR and GLAIR Agents

MGLAIR is based on, and extends, the GLAIR (Grounded Layered Architecture with Integrated Reasoning) cognitive agent architecture [Hexmoor et al., 1993b].

GLAIR has as a major part of its knowledge layer (the subsystems of which combine to produce conscious reasoning) the SNePS knowledge representation, reasoning, and acting system [Shapiro and The SNePS Implementation Group, 2007]. Work on GLAIR began when Kumar and Shapiro extended SNePS' pure reasoning abilities with a subsystem capable of integrating reasoning with acting and planning [Kumar et al., 1988] [Kumar and Shapiro, 1991]. This acting system is a key part of MGLAIR.

GLAIR has been used to instantiate many embodied cognitive agents in real and simulated environments, including a Foveal Extra-Vehicular Activity Helper-Retriever (FEVAHR) [Shapiro, 1998] embodied both in a simulation and a hardware robot with a number of sensors and effectors; and agents realizing models of psychological phenomena such as the ability to identify perceptually indistinguishable objects [Santore, 2005], theories of early mathematical cognition [Goldfain, 2008]; numerous agent-actors in virtual dramatic performances [Anstey et al., 2009]; and many others.

Though many of these have the ability to sense using multiple sensors and act with multiple effectors, and some have been considered MGLAIR agents, they lack a shared framework that specifies the nature, behavior, and general use of modalities. To the extent that these agents do divide their capabilities into separate modalities, those are implemented ad-hoc with no standard approach to their integration with reasoning and acting, no way of abstracting away modality properties and handling, and no set of mechanisms for handling them. Prior to the development of the MGLAIR architecture, designing and implementing GLAIR agents with some multimodal capabilities required the agent architect to reimplement functionality that may already exist in other agents at all layers of the architecture. The theory and implementation of MGLAIR presented here establishes modalities as instantiable objects that are parts of agents and whose properties determine their behavior in the context of built-in mechanisms that allow for their simultaneous use by the agent. The implementation provides a library of functionality for use by agent architects to define and use modalities without concerning themselves with low-level implementation details like handling percept queues, managing communication between separate layers, etc.

Chapter 3

Modality in MGLAIR

3.1 Introduction

Each MGLAIR modality corresponds to a single afferent or efferent capability of an agent. Each modality is a limited resource capable of implementing only a limited number of related activities, or of transmitting and transforming into percepts a specific type of sensation.

The set of modalities available to an agent necessarily depends on the agent's embodiment: an agent embodied in a robot with motorized wheels and a camera-based vision system should be capable of moving from one spot to another by consciously performing actions that use its locomotion modality, and of perceiving visual facts about the world through the use of its visual modality.

The architecture's design permits the concurrent use of multiple modalities. Since each modality corresponds to a distinct capability, the use of one does not interfere internally with the use of others, though using a modality will often change the state of the world and thereby affect the use of others.

That is, both afferent and efferent MGLAIR modalities exhibit a high degree of modularity [Fodor, 1983]: each is specific to a particular type of sensor or effector, and the internal details of each modality is inaccessible to the agent's mind. There is debate over the extent to which perceptual subsystems can be isolated from the rest of cognitive systems of which they are a part [Pylyshyn, 1999]. The model of modality presented here may possess more modular isolation than actually exists in humans. MGLAIR's model of modal *focus*, discussed more in section 3.8, does permit an agent to consciously exert some top-down influence over processing in the lower layers of a modality.

While MGLAIR does provide default modality definitions for commonly-used modalities, my focus is on

providing a general model of modality rather than on implementing a particular set of specific modalities. The model makes claims about what, in general, a modality *is*: what properties it has, how these properties determine its behavior, how modalities are integrated with the rest of MGLAIR as a cognitive system, and so on. This general model and corresponding implementation allow agent designers to extend, exclude, or modify the basic set of modalities, and to define as many additional modalities as are needed for different agents and embodiments. It provides sensible, customizable, default procedures for the integration of afferent modality percepts within the knowledge layer.

Recall that in MGLAIR all of an agent's conscious thought takes place in the agent's symbolic language of thought at the KL. MGLAIR's modality-specific PML structures, with which symbolic knowledge is aligned, constitute unconscious multi-modal representations.

Any action an agent consciously performs is available as an abstract symbolic representation in the knowledge layer, corresponding to low-level motor control commands in the SAL, which is connected to it through alignment with the intermediate PML structures. The flow of distinct types of sensory and motor impulses between the agent's mind and body occur independently, each in its own modality.

An MGLAIR modality possesses a directional data channel that connects the mind to the body (see Section 3.2.4). The modality itself handles the transmission and transformation of data in the channel: raw sense data within an afferent modality at the SAL is passed up to the PML and converted to perceptual structures. These structures in the PML are converted to conscious symbolic representations of percepts as they are passed upwards to the KL. The architecture itself does not specify how raw sense data is converted into PML structures except in the case of provided example modalities. The exact nature and implementation of that transformation depends on the type of modality in question and on the agent's embodiment. Similarly for efferent modalities: when the agent consciously acts, the symbolic representation of that action is transformed as it is passed down through the layers until it reaches the SAL, where the act produces an effect on the state of the agents body in the world.

The SNePS KL and its connections to the PML are implemented in Common Lisp. MGLAIR does not require the PML itself or the SAL to be implemented using any particular technology: the software implementations of an agent's KL, PML, and SAL may even run on different physical machines. Communication between layers of the architecture is by default achieved with TCP/IP sockets or other interprocess communication to pass sensory and motor impulses. The architecture does not require the use of sockets for this: layer interface specifications deal with the nature of the data that will be passed (format, capacity, etc). Any compatible interlayer connection type may be used.

3.2 Modality Properties

An MGLAIR modality is a 10-tuple of modality properties: ⟨ `name`, `type`, `predicates`, `channel`, `buffer`, `access`, `focus`, `conflict`, `description`, `relations` ⟩, all of which are described below.

`name` - A unique name identifying the modality (See §3.2.1)

`type` - The modality's type (See §3.2.2)

`predicates` -Knowledge layer predicates associated with the modality. (See §3.2.3)

`channel` - Specification for the modality's data channel (See §3.2.4)

`buffer` - Specification for the modality's buffer (See §3.2.5)

`access` - Conscious access permissions granting or denying the agent knowledge of the modality (See §3.2.6)

`focus` - Permissions governing focus for the modality (See §3.2.7)

`conflict` - Specification governing conflicting access attempts (See §3.2.8)

`description` - A human-readable description of the modality (See §3.2.9)

`rels` - Relations to other modalities (See §3.2.10)

3.2.1 Modality Name

The modality name is a symbol that identifies the modality (e.g. `locomotion`). If the agent has conscious access to the modality, the name is the symbol by which the agent refers to it. The name is also used within the acting system and other Lisp modules for managing modalities.

3.2.2 Modality Type

Each modality has a type that is a subtype of either `afferent` or `efferent`. The modality type determines the direction of the modality's data channel, as well as which default mechanisms are used to integrate it with the knowledge layer.

For instance, one subtype of `efferent` modality might be `wheeled-locomotion`, an instance of which an agent implementation would employ to control its wheels.

3.2.3 KL Predicates

Knowledge layer predicates associated with an afferent modality are used to construct symbolic representations of percepts. For instance, the predicate `Saw(x,t)` could be associated with an agents `vision` modality and used to make assertions that the agent visually perceived some object at some time.

For efferent modalities, these KL terms identify primitive acts associated with the modality. For instance, `move(d, u)` might be associated with an agent's `locomotive` modality, and the agent might perform acts with `move` in order to move its body a certain number of units in a particular direction.

3.2.4 Data Channel

An MGLAIR modality's data channel facilitates the transmission of sense data and act impulses across layers of the architecture. As discussed in Section 3.11, this involves some method of interprocess communication.

A modality's data channel specification must provide the `connection type` and `required parameters` in order to establish this connection.

For instance, a `TCP/IP Socket` connection has parameters `hostname` and `socket`, which are required to establish a data channel over TCP/IP sockets. The data channel specification may also include a `data format` restriction, which determines the type of data and impulses the modality data channel can convey.

3.2.5 Modality Buffer

The modality buffer specification determines the properties of the modality's buffer. An afferent modality's buffer is used to store data arriving via its data channel while the data awaits processing. For instance, an agent with a `visual` modality may perform non-trivial processing on the sense data passing through that modality before it is converted into percepts available at the knowledge layer. In efferent modalities, the buffer is used to store act impulses that have yet to be processed by the PMLc.

Three components determine the handling and flow of data within a modality. First, the modality buffer, discussed above and described in more detail in 3.7 has a capacity and/or time-based handlers that manage how much data can be temporarily stored within the modality's data channel and when it is removed. Second, the conflict handling specification, discussed more below in 3.2.8 can be used to further refine how attempted simultaneous multiple uses of the modality data channel are handled. Third, modality focus, discussed in 3.8 interacts with these first two mechanisms: raising the focus on a particular modality dedicates more computational resources to its post-data channel processing and thereby makes its buffer less likely to fill up or experience conflicts.

3.2.6 Conscious Access Permissions

MGLAIR allows agent designers to hide a modality from an agent, making it consciously inaccessible for the agent, or to grant the agent access to some information about a modality – including that it is a modality. A conscious access permission has the value `true` or `false`. If `true`, then the agent has access to basic information about the modality, including the ability to refer to it by its `name`. If `false`, the agent does not have this ability. Such an agent will still have the ability to perceive (or act) using the modality, but will not have a way of consciously referring to it, or believing anything about it. The default value for conscious access permission for `modality`, which is inherited by all subclasses of both `efferent` and `afferent`, is `true`. Agents' conscious access to modalities is discussed in more detail in section 3.9.

3.2.7 Focus Permissions

The main purpose of modality focus is to allow agents to regulate the amount of "attention" they pay to individual modalities. The modality focus specification includes include both an *initial focus level* for the modality, and *a modification permitted flag* indicating whether the agent may alter its focus on this modality. Unless otherwise specified by the agent designer, all modalities start with a default focus level that is agent-modifiable. Note that an agent must have conscious access to a modality in order to adjust its focus on that modality. A more detailed description of modality focus is presented in Section 3.8.

3.2.8 Conflict Handling Specification

A conflict handling specification determines how attempted multiple concurrent accesses of the modality will be handled, for instance if the agent attempts to `move` using its `locomotion` modality when that modality is already in use.

Available options are `blocking`, `interrupting`, `suspending`, and `queuing`. These are described and discussed further in Section 3.6.

3.2.9 Description

The modality description is a human-readable string, the contents of which have no influence on the workings of the modality. It is used for display purposes and as a form of internal documentation for modalities. For instance, an agent's `locomotion` modality might have the description "Allows the agent to translate and rotate using its wheeled locomotion system" – the contents of this description are inaccessible to the agent,

even when the agent has conscious access to the modality itself.

3.2.10 Relations to Other Modalities

An MGLAIR modality can have a defined relation to another modality that reflects a causal, logical, or other type of relation that holds between them in the world – e.g. the *proprioception-for* relation, which can hold between an afferent modality and an efferent modality, and establishes that the afferent modality conveys sense data related to the use of the efferent modality, as discussed in Section 3.10. We leave open the possibility of adding other relations between modalities in the future, for instance to support a model of cross-modal interference as discussed in Section 3.10.

3.3 Modality Definitions

A modality definition is a structured description of a modality that specifies each of the properties discussed above for that modality, including the nature of interlayer communications between components of the modality at each level in the architecture.

Modality types and modality instances are represented in the upper PML, where they are integrated with the KL, by CLOS classes and objects that instantiate them. All modality classes are subclasses of the general class `modality`, which has slots for properties common to every modality, including the properties outlined and described above in section 3.2: `name, type, predicates, channel, access, conflict, description, rels`. Subclasses like `efferent`, `afferent`, `locomotion`, and `vision` override the general values for these slots when appropriate, for instance with modality-specific predicate symbols or descriptions.

This arrangement is shown in Figure 3.1, which shows the general classes `modality`, `afferent`, and `efferent` extended by more specific modality classes for `vision` and `locomotion`, which are themselves extended for platform-specific modalities such as `Rovio vision`. The `Rovio vision` class is instantiated as part of an actual robotic agent implementation. `modality`, `afferent`, and `efferent` are abstract enough that they should generally not be directly instantiated. General modalities such as `vision` and `locomotion` are included as part of MGLAIR with sensible default behaviors, and may be directly instantiated as part of agent implementations, though it is usually better for the agent designer to subclass them with embodiment-specific modalities.

Modality types may be defined initially either through the use of Lisp functions that generate modality objects, or by creating a description in RDF that is read in by the system.

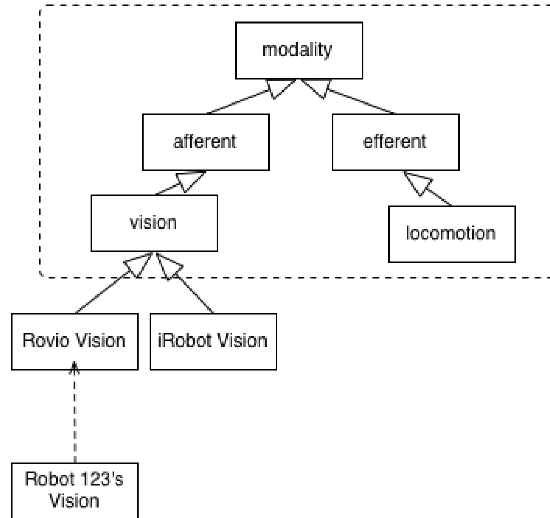

Figure 3.1: Modality CLOS classes and instances

Once generated, a modality definition can be persistently stored as an RDF file to be read in and used by programs that implement the separate layers in the architecture.

3.4 Sensing

Rules in the upper layer of each afferent modality transform perceptual data from the PML into conscious knowledge within the KL. This involves converting perceptual structures into propositional terms that stand for what is being perceived. These rules and structures are implemented in PMLs: the PML, sub-layer *s*. Each such rule is specific to its modality and uses predicates and terms that may be unique to the modality to generate the propositional terms that correspond to percepts in that modality.

Though it is up to the agent/modality designer to specify how complex sense data are are translated into conscious percepts, the architecture includes default translations for each afferent modality. These defaults are activated simply by defining and using an afferent modality that does not override them.

Separate processes (at least one per modality) make the connection between the PML and KL, converting arriving sense data into knowledge concurrently as it arrives. This allows the agent to perceive via all of its afferent modalities simultaneously.

This is achieved by associating with each modality instance in the upper PML a function called a `percept-processor` that is applied to data as it comes out of the modality buffer for that modality. As a simple example, consider a `distance` modality for a sonar-based range detector that produces numeric values

in the range of 0 to 255 indicating the distance to the closest thing in front of it. One way of implementing an agent equipped with this sensor might have a function attached to the sensor's modality that, when a new item comes out of the modality buffer (93, for instance), simply asserts to the knowledge layer the percept `DistIs(93)`. The NXTBot, one of the agents described in Chapter 7, is equipped with this sensor. However, its percept-processor performs the slightly more complicated task of classifying the distance readings as `near`, `far`, etc.

The mechanisms for regulating an agent's *focus* on sensory modalities, described in Section 3.8, operate by lowering or raising the frequency with which a modality's percept processor is run.

3.5 Acting

When a primitive act is defined, it is associated with an efferent modality. A single modality may have associated with it multiple acts that utilize the same basic bodily capability or resource. For instance, an agent with a grasping effector may have a modality dedicated to the use of that grasper. Distinct acts that use the modality, such as *positioning the grasper*, *grasping an object*, and *releasing an object*, have distinct conscious representations. When the agent performs acts, the PML and SAL handle decomposing them from abstract symbols (e.g. `grasp`) into low-level commands (e.g. `apply voltage to motors a, b, c for n seconds`), and executing those commands.

3.6 Single-use

Each efferent modality can carry out only a single activity at a time. Any desired activity that seems to violate this principle should be decomposed into component acts, which may use multiple modalities concurrently. MGLAIR's acting system includes primitive and complex acts. Plans for the latter are formed by combining other acts, either primitive or complex, via control acts

If an agent attempts to use a single efferent modality while it is already in use, the system may handle the situation in one of several ways:

`blocking` - The new impulse is blocked and discarded.

`interrupting` - The new impulse interrupts the current activity and then proceeds as normal. The interrupted activity is discarded,

`suspending` - The new impulse interrupts the current activity and then proceeds as normal. The interrupted activity is suspended and resumes once the interrupter finishes.

`queuing` - The new impulse is placed in an act queue for that modality and is carried out when the modality is next available.

The manner in which such conflicts are handled is a property of each modality, and part of its definition.

An agent that is capable of sensing or knowing that it is using an efferent modality (see Section 3.10) should be designed to consciously handle this. For instance, an agent that is capable of sensing that it is currently carrying out an action using its locomotion modality is capable of consciously waiting until that action has finished before using the modality for another action, or of deciding to terminate the ongoing action.

Afferent modalities are subject to similar restrictions and design choices. If a flood of sensory information overwhelms an afferent modality, it must be dealt with by blocking, interrupting, suspending, or queuing, or some combination thereof. Given the nature of sensation, it is not at all clear that it means anything to suspend and resume a sensation, so this option is not available for afferent modalities.

3.7 Modality Buffers

Each afferent modality data channel includes a buffer of sense data arriving from the SAL via the lower levels of the PML. Sense data are added to the buffer as they are generated, and removed in the same order for processing by the upper levels of the PML, where modality-specific perceptual processing mechanisms convert PML structures into symbolic representation of percepts. Sense data in the modality buffers are marked with timestamps. Neither the raw sense data nor their timestamps are consciously accessible to the agent, though both can be used within the PML to construct percepts.

An afferent modality that generates sense data faster than the agent can process and perceive them would be problematic without mechanisms to handle such cases. To prevent agents from becoming overwhelmed by sense data generated faster than they can be perceived, modality buffers have an optional fixed capacity, which is specified when the modality is defined. It would usually be ridiculous for an agent operating in real-time to suddenly perceive an event that affected its sensors a significant time in the past – and this is an inevitable consequence of failing to limit the modality buffer's capacity.

A modality with a fixed capacity may handle incoming sense data that would otherwise be placed in its (full) buffer either by refusing and discarding arriving sense data when the buffer is full, or by pushing out

old data to make room for the new. Modality buffers allow for either of these possibilities.

Modality buffers without a fixed capacity must specify an expiration interval for sense data. Old data in a modality buffer that reaches expiration are discarded without being processed and perceived. A buffer may have both a capacity and an expiration interval. Modality buffers have a fixed capacity by default. It is recommended to use a capacity limit, even in cases where the agent designer expects the expiration interval, to prevent the buffer from growing too large.

A more detailed discussion of the structures and procedures underlying MGLAIR modality buffers appears in Chapter 4. Implementation details appear in Chapter 6.

3.8 Modality Focus

Though MGLAIR does not currently contain a full model of *attention*, it does include related capabilities that will facilitate the development of such a model, and that allow agents to consciously choose to dedicate more of their resources to some modalities. An MGLAIR agent may elect to focus more on a particular afferent modality than on others by raising or lowering the focus level assigned to each, or to ignore a particular modality altogether.

Each distinct modality has its own processes in the upper level of the PML that operate continually and concurrently to manage the flow and transformation of data and impulses in the modality. Altering a modality's focus increases or decreases the frequency with which its internal processes run.

The ability to govern modality focus is useful when an agent is carrying out a task that relies heavily on perceiving with a subset of its sensory modalities, or when percepts in some modalities may be more urgent than percepts in others. An agent embodied in a robot with cliff detection and speech recognition might focus more on perceiving with its cliff detection modality than on its speech recognition. In this case, and in many others, the less urgent modality may have a much higher volume of sense data than the more urgent one. Without a mechanism to selectively allocate its "attention", such an agent would by default dedicate its attention to the modality producing the most percepts, and end up delaying processing or, in the worst case, completely ignoring, the more urgent and relevant modality.

An agent may alter its focus on a modality by performing one of the following acts with the modality name as its argument:

`focus(<modality-name>)` increases the agent's focus on the modality.

`unfocus(<modality-name>)` decreases the agent's focus on the modality.

`ignore(<modality-name>)` causes the agent to ignore the modality completely.

`attend(<modality-name>)` causes the agent to stop ignoring the modality, restoring it to the focus level it held before being ignored.

`restore-focus(<modality-name>)` sets the modality's focus back to its starting level.

By default, all of an agent's modalities start with equal focus levels. Each modality definition may include an initial level of focus for the modality. A modality definition may also prohibit the agent from changing a modality's level of focus.

A more detailed discussion of the structures and mechanisms underlying MGLAIR modality focus appears in Chapter 4.

3.9 Conscious Access to Modalities

In order for an agent to consciously act to modify its focus on a modality, the agent must be aware that it possesses the modality, and must have a way of referring to it. Not all modalities must be accessible to their agents in this way. Whether a modality is consciously accessible to the agent is a property of that modality specified in that modality's definition.

Percepts arriving at the KL, and the resulting symbolic beliefs, are linked to the modality that produced them. This allows an agent to reflect on how it came to believe that something is the case ("I saw it"), and to perform meta-reasoning about its own perceptual capabilities.

An agent's awareness of its own modality may be attained simply by including a term in the KL that refers to the modality, along with some other basic assertions.

Another possibility that the architecture does not preclude is that an agent might be capable of *becoming aware of* modalities that it was not aware of initially. Such an agent could be designed to notice capabilities that it is not consciously aware of, for instance by reflecting on new knowledge that it acquires through perception. It could invent its own symbols to refer to the modality and learn to associate percepts with the modality.

An agent that knows about some of its own modalities may also use this knowledge to reason about its acts and their effects in combination with what it perceives. For instance, an agent that has a suitably rich model of the world will know that any successful use of its locomotive modality will result in that agent's body inhabiting a different spatial region, and may be capable of verifying this using one or more of its

sensory modalities – and of avoiding being surprised by apparently frequent changes in the world it visually perceives.

Agents also benefit from knowledge about limitations of their sensory modalities: an agent with a visual system that can only look in one direction at a time will be better at perceiving and planning if it has some conscious knowledge of this restriction.

3.10 Relations between Modalities

We have implemented a simple model of proprioception for MGLAIR in which an efferent modality is associated with a related afferent modality, allowing the agent to perceive its own actions based on sense data generated in the SAL. For example, an agent that is embodied in a wheeled robot with motors that include a tachometer can use this feature to sense its use of its motors.

The modalities for *locomotion* and *locomotive proprioception* are connected at the PML, where a locomotive proprioception modality register stores perceptual data pertaining to the current state of locomotion. This may include a simple sense corresponding to its just being in use, or may be a more advanced sense based on the position of motors, the duration of the current use, etc. Depending on the application and design of the agent, the agent may be given conscious access to this register. In this case, the agent may simply know whenever its motors are in use because it is able to directly perceive that fact. Alternatively, proprioceptive modality registers may remain hidden in the PML with their contents inaccessible to conscious reasoning. In this case, the sense information they represent can still be used to manage act impulses for that modality within the PML, for instance by blocking new acts in the associated efferent modality while it is still in use at the SAL.

We have used the same pattern to create an agent whose embodiment includes a tri-axial accelerometer that acts as a "vestibular" afferent modality tied to the agent's locomotive afferent modality. This agent learns what it feels like to carry out certain locomotive acts, and uses that information to monitor its body, reflect on the effectiveness of its actions, etc.

This example of an efferent modality tied to a corresponding afferent modality may serve as a model for the development of more complex inter-modal interactions. A similar approach might join senses from modalities before they are perceived. E.g., an agent with vision and sonar-based object detection might merge sensory data from those two modalities into joint percepts in the PML. This approach might be used to model many of the forms of cross-modal interactions observed in nature [McGurk and MacDonald, 1976] [Stein and Meredith, 1993] [Calvert et al., 2004].

3.11 Platform Independence

For some MGLAIR agents, the software parts of the KL, PML, and SAL are implemented using different programming languages, and executed on different physical machines.

The architecture facilitates this sort of heterogeneity by specifying interfaces between the layers that establish how each layer communicates with those adjacent to it. Communication between layers of the architecture is often achieved with TCP/IP sockets or other interprocess communication to pass sensory and motor impulses. The architecture does not require the use of sockets for this: layer interface specifications deal with the nature of the data that will be passed (format, capacity, etc). Any compatible interlayer connection type may be used.

3.12 Modality Granularity

Example modalities mentioned so far, and the modalities used by MGLAIR agents discussed in detail later in this dissertation have been at the level of granularity of the general senses we humans possess: `vision`, `hearing`, etc. While this level of granularity seems like a sensible default, it is not required by MGLAIR, and the level of granularity used for a particular agent and agent embodiment may depend almost entirely on the hardware and the purpose for the agent. For instance, an agent with a roughly cylindrical robotic body that has tens of sonar-based distance centers around its circumference could be designed perceive with these sensors in aggregate as a single "proximity" sense in a single modality, OR could be designed with a separate modality for each of the sensors. The only strong restriction MGLAIR places on the flexibility of modality granularity is the prohibition on the use of multiple modalities for a single bodily resource or capability. To have two modalities for a single effector, for instance, would break this rule and undermine the effectiveness of modality mechanisms that rely on it.

Chapter 4

Modality Mechanisms

Chapter 3 introduced and gave an overview of the MGLAIR model of modality, and outlined the basic properties of modalities, and the mechanisms they use. This chapter describes in more detail how modalities are used and how they are integrated with the agent's conscious mind at the knowledge layer. It also presents several key structures and mechanisms that accomplish this, including modality-specific perceptual functions, perceptual buffers, buffer management mechanisms, and the model of modality focus.

4.1 Becoming Aware of Percepts: Integration with the KL

4.1.1 Modality and Architecture Overview

Recall that the MGLAIR architecture is divided into three layers, the Knowledge Layer, the Sensori-Actuator Layer, and the Perceptual-Motor layer. The PML is further stratified into sub-layers. The highest PML sub-layer, comprised of PMLa and PMLs, grounds KL symbols for actions and percepts in subconscious actions and perceptual structures respectively. The lowest PML sub-layer, the PMLc, directly abstracts the sensors and effectors at the SAL into the basic behavioral repertoire of the robot body. The middle PML layer, the PMLb, handles translation and communication between the PMLa and the PMLc.[1]

An MGLAIR agent becomes consciously aware of percepts once they have been added to its knowledge layer. A gradation of abstractions across the layers of the architecture terminates in symbolic knowledge at

[1]In GLAIR, the original names of the PML sub-layers *PMLa*, *PMLb*, and *PMLc* come from the first letters of the alphabet, in order, from the top down. PMLs was later added parallel to PMLa. These now stand for *PMLsensing* and *PMLacting*, reflecting the nature of the split in the upper sub-layer. The reader may find it mnemonic to think of the PMLb as *PMLbridge*, as it serves to connect and communicate between the PMLa/s and PMLc. PMLc may be thought of as *PMLconcrete*, as it is the PML sub-layer responsible for dealing with the specifics of the SAL.

the KL.

Each afferent modality contains within its PML a perceptual buffer that stores perceptual structures, which have moved up through the layers of the architecture but have yet to be perceived by the agent. Sense data originating at the SAL pass up through the PMLc and PMLb along the modality's data channel and are converted into perceptual structures that are added to the modality's perceptual buffer. Buffer maintenance processes operating at the upper level of the PML remove these structures from the perceptual buffer, and apply the modality's perceptual function to them, which converts them to symbolic percepts and asserts them to the KL.

Perceptual processes in each afferent modality apply modality-specific sense-handling functions and perceptual functions to each sensory structure that has made its way through the modality's buffer. The implementation details of each perceptual function are specific to the modality and agent that it is part of, though all have the same basic task of transforming perceptual structures into symbolic percepts.

I take the contents of the PML - its structures and processes - to constitute *unconscious modal representations* of the things in the world that they correspond to. These are not accessible to the agent (hence "unconscious") and cannot be reflected upon by the agent. Though these perceptual structures and the processes that manipulate them are unknowable to the agent, they place a crucial role in its cognition. Alignment with PML structures that correspond to sense data is key to the agent's conscious awareness of the world.

4.1.2 Single Modality Example

As a simple example, consider an agent embodied in a small robot with - among other sensors - a unidirectional ultrasonic range finder that the agent uses to avoid bumping into obstacles as it goes about performing some task. In order for the agent to use information from this sensor as it reasons and acts, it must have a way of representing that information and of becoming aware of it. The agent becomes aware of percepts via a modality that connects the sensor to the agent's mind, with mechanisms to convert raw sense data into the agent's perceptual representation. The range finder operates by sending out sound waves and then detecting how long it takes for the echo to bounce off the nearest object and return to the sensor. At the lowest level - the SAL - the data produced by this sensor is in the form of voltage levels generated by a piezoelectric receiver. These values are converted into an integer value between 0 and 255, which corresponds to the distance to the nearest object in front of the sensor. If the agent's only use for this sensor is to avoid banging into walls and other obstacles, then there's not much point to having the agent capable

of consciously perceiving and representing all of these possible values. Instead, it may suffice for the agent to perceive that the next thing in front of it is very close, very far, or in-between. This can be achieved by having the PML convert values like `34, 148`, etc, into structures at the granularity we want the agent to perceive: `close`, `medium`, and so on. These are removed from the modality's sensory buffer and processed by the modality's perceptual function, which produces symbolic representations that are then asserted at the KL, e.g. the term `DistanceIs(far)`, which represents the proposition that [currently][2] the distance to the nearest object in front of the agent is far. These percepts (beliefs) affect the agents behavior if it holds policies like "when the nearest thing in front of me is very close, stop any forward motion and turn before resuming it." Figure 4.1 shows the layers and examples of information present at each.

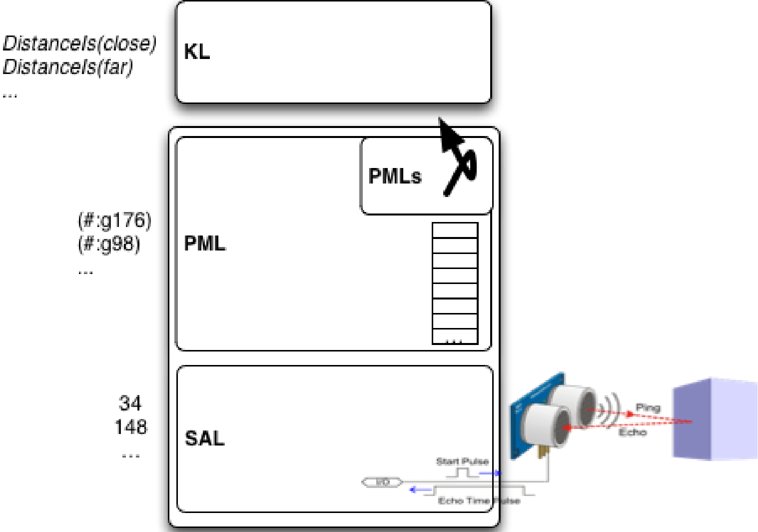

Figure 4.1: View of a single afferent modality for an ultrasonic sensor [3]

Note that though, in this example, the simple PML structures mentioned above helpfully take the form of (Lisp) symbols that look like English words, there is no reason that they must be like this, and there's no dependency between the English words they look like and either the state of world that they correspond to, or the agent's representation of the percepts. They could just as well be generated symbols (as in Figure 4.1), or even be arranged so that both the percept `DistanceIs(far)` and the situation in the world in which the closest obstacle is far from the agent are aligned with a symbol in the PML that looks like `close`. In general, a modality's perceptual function makes conscious perceptual knowledge out of structures within the

[2]This example ignores time for simplicity's sake.

[3]Sensor image from `http://learn.parallax.com/KickStart/28015`, available under the Creative Commons Attribution-Share Alike 3.0 license.

PML for that modality. These structures have meaning only through being aligned with a certain state of affairs in the world.

In this example the transformation of sense data within the PML and the role of the perceptual process are simple, in part because of the simplicity of the sense data itself. This could all be achieved with a few conditional rules that operate on the sensor values. However, even with a simple sensor with such a small range of values, it is possible to perform more complex perceptual processing in the PML. For instance, one might use a statistical classifier and train the agent (interactively or otherwise) to distinguish `close` from `far`, etc. I have implemented an agent that uses an approach similar to this in processing sense data for its `vestibular` modality, in which the agent learns what it feels like when it is moving in different ways by learning to make connections between its own actions and the PML structure features in an accelerometry modality. This is discussed in more detail in Chapter 5.

4.2 Perceptual Functions

Each afferent modality has a function specific to it that is applied to perceptual structures in its PMLs to convert them into symbolic knowledge and assert them to the KL. The nature of each perceptual function depends on the type of perceptual structures it handles, which depends on the nature of the sensor and modality. However, all perceptual functions have some commonalities because of their common purpose: each perceptual function takes as its input a PML structure that has passed through the modality's perceptual buffer, and produces as output terms in the language of the Knowledge Layer that use the frame(s) associated with the modality to represent the percept. The perceptual function is also responsible for linking the asserted percept to the modality in which it originated, and may also serve to convert timing information included with the percept and integrate it with the agent's model of time.

For instance, the sample distance-sensing modality in 4.1.2 has a perceptual function responsible for taking simple PML structures like `(#:g176)` from the modality's percept buffer and converting them into KL terms like `DistanceIs(far)`, then asserting them. A more complicated perceptual function might take a PML structure that represents an arbitrarily large sequence of features, in e.g., a full visual modality, and perform more intensive operations in order to convert that structure into a representation suitable for assertion at the KL.

This variability in the complexity of the processing performed by perceptual functions, and hence in the amount of computational resources each may consume, is one motivating factor behind MGLAIR's model of modality focus. Briefly, a modality's focus determines how much processing time that modality's perceptual

function will get relative to the other modalities. This is discussed more below in Section 4.5.

4.2.1 Linking Knowledge to Modalities

Besides simply building and asserting terms for the agent's conscious representation of a percept, a modality's perceptual function also establishes a link between the resulting terms and the modality in which they originated. This involves embedding a reference to the modality within the SNePS data structure that represents each term. The modality in which each term originated is considered its *supporting modality*. The SNePS Belief Revision System (SNeBR) includes a *support set* for every proposition, which indicates whether the proposition is a hypothesis (in case it was asserted directly) or is derived (in which case the propositions from which it was derived constitute its *origin set*) [Shapiro and The SNePS Implementation Group, 2007]. A proposition that has been asserted by a modality's perceptual function (rather than being asserted outside of any modality, or derived from other asserted propositions) is currently represented by a hypothesis origin tag in its support set. This, in combination with its supporting modality, indicates that the proposition originated in the agent's perceptual system rather than as an amodal assertion. SNeBR's current model of belief revision does not treat perceived propositions any differently from those that are asserted, e.g., at the SNePSLOG prompt. However, this link between perceptual assertions and the modalities that produced them could serve as the basis for interesting future work on modality-specific belief revision in MGLAIR. For instance, we might want an agent to adjust its own beliefs about one of its sensors or modalities - about its reliability or characteristics - and use this information to adjust not only the asserted percepts that originated within that modality, but also beliefs that were derived from it.

An implicit link between each term and its originating modality is also established through the use of the frame(s) associated with that modality in the terms. Neither of these links is necessarily accessible to the agent, which may or may not know explicitly that it possesses such a modality. As discussed earlier in Section 3.9., whether the agent can know about (believe propositions about) a modality is a feature of that modality determined at the time that the modality is defined.

As discussed below in Section 4.5, it can be useful for agents to have knowledge of their own modalities, for instance in order to consciously act to adjust a modality's focus, or to represent and reason about relationships between modalities: one might want an agent to be capable of learning and representing a causal link between the use of an efferent modality and its perception in a separate afferent modality (a grabber or other effector, and a pressure sensor on the end of that effector, for instance). It can also be useful for agents to know how certain knowledge was attained by consciously representing a connection

between a piece of knowledge and the modality through which it was attained. Sometimes the answer to the question "how do you know that?" is "I saw it." Under one of the the simplest possible implementations of an agent with a visual modality, the agent might be set up to represent having seen a thing `x` at some time `t` with a term like `Saw(I, x, t)` without having any explicit knowledge of its visual modality, or a way of referring to it, or any other information connecting different appearances of "Saw(...)" to each other (except, of course, for their common use of the same frame). In such a case, the agent consciously treats this perceptual knowledge no differently from any other hypothesis. In order to make an explicit connection between an asserted proposition `ap` and a modality `m` that will be accessible to the agent, the modality's perceptual function may, in addition to asserting `ap`, also assert *that* `ap` *was perceived via modality* `m` – provided that the agent has been granted conscious access to `m` and has a symbol that allows it to refer to it.

4.3 Perceptual Buffers

Consider the case where a modality's perceptual process takes some (even small) amount of time, t_p to process each input and produce symbolic percepts from it. What should happen if, during one t_p span, the modality's sensor and SAL sometimes generate multiple sense data which propagate up through the modality's data channel? One possibility is that the newest sense data are simply discarded if the modality's perceptual process is still busy with the last structure it received. In this case, the agent might often fail to perceive something about the world that it was capable of perceiving just because of unfortunate minor timing issue. Another way of handling this would be to interrupt the perceptual process' work and set it to work on new PML structures as they arrive. This could also cause the agent to miss something it could have perceived, or it might even leave the percepts in some partly-constructed state. While the latter could be interesting as a way of modeling some illusions or other natural phenomena, the current model of MGLAIR tries to avoid effects like this through the use of perceptual buffers within each afferent modality.

Modality buffers for each modality in the PML queue up perceptual structures before they are processed by perceptual functions to be consciously perceived by the agent. The behavior of these buffers depends in part on their properties, specified as part of each modality definition. These include the buffer *capacity*, a limit on the number of elements the buffer can hold. A buffer may have an unlimited capacity, in which case it must have an *expiration interval*, an amount of time after which data in the buffer expires and is discarded without being perceived by the agent.

Note that the problem of a full buffer raises some of the same issues discussed above about what to do with newly-produced sense data. Ultimately there is no perfect solution to the problem of an afferent modality

generating sense data faster than it can be processed and perceived by the agent. The main contribution of modality buffers is to smooth out perception and minimize the loss of sensory data that the agent would otherwise perceive because of temporary large disparities between the speed with which the sensor generates data and the time it takes to process and perceive that data. By adjusting a buffer's size and/or its expiration interval, and by specifying how full buffers are handled, an agent designer may achieve a range of possible effects, some of which may be more appropriate to particular kinds of agents and modalities than to others.

Adding PML structures to the perceptual buffer

Timestamped PML structures assembled at the PMLb sub-layer are added to the modality buffer as they are created. If the buffer is full when a structure is ready to be added, for instance because the buffer has a fixed capacity and the expiration interval is too long, then an attempt to add a new structure to the buffer will be handled in accordance with the modality's conflict-handling property.

If the modality buffer is set up to block in this situation, then the new PML structure will be discarded rather than added to the buffer. This means it will never be perceived by the agent. Otherwise, the default behavior is for the buffer to make space for the new structure by deleting, unperceived, the oldest unprocessed structure in the buffer – even though its expiration has not yet been reached.

A third option, not currently implemented, would be to have the new structure replace the previous newest structure (i.e. the newest structure already in the buffer). Though it may be possible to think of scenarios where this would best produces a desired behavior, it is in practice quite similar to simply rejecting the newest structure.

Yet another possibility, and an interesting candidate for future work would be to remove perceptual structures at random after some time, or to have them decay over time so that they are not lost in strict order, but rather disappear with increasing probability over time.

A related possibility is to have perceptual structures in the buffer decay over time by corrupting or losing features until, after some amount of time sitting in the buffer unprocessed and unperceived, the entire structure has been rendered useless.

Processing PML structures from the perceptual buffer

Each modality buffer has a buffer management process that repeatedly removes the oldest non-expired structure from the buffer and applies the modality's main perceptual function to it.

As discussed more below in Section 4.5, altering a modality's focus will influence the processing of its

buffer by adjusting the priority of the modality's buffer management and perceptual function processes. Consider an agent with just two modalities with nearly all of the same properties connected to two identical sensors in the same environment, but with different levels of focus. In this situation, the modality with the lower focus will remove and process PML structures from its buffer less frequently, and will therefore be more likely to reach its capacity or to have elements reach their expiration times before they are processed and perceived. Or, to put it another way: an agent that is not focused on some modality may fail to perceive some things that the modality is capable of sensing.

4.4 Act impulses in efferent modalities

Examples and discussion in the above sections are focused on afferent modalities and their use across the layers of the architecture to connect sensors with conscious perception. Much of the detail is the same for efferent modalities, which connect conscious acting at the knowledge layer to effectors in the agent's SAL.

At the knowledge layer, SNeRE, the SNePS Rational Engine, connects the agent's reasoning and acting capabilities through the management of policies and plans. An example policy for a game-playing agent (stated in English from the agent's perspective) is, "Whenever it is my turn I should select a move and then make that move." An example plan is, "To move a piece, I first pick up the piece, then move the piece to its new position." A plan for picking up a piece is: "To pick up a piece, I first open my grasping effector, then position it over the piece, then I lower it, then I grasp with it, then I raise it." Each act plan consists of a *complex act*, which is being defined by the act plan, and a sequence of other acts that comprise the complex act. These may be themselves complex acts or primitive acts. All such plans must eventually bottom out in *primitive acts* – acts that the agent cannot introspect on or further divide into their composite parts, but which it may simply `perform`. Primitive acts are attached to functions in the PMLa.

Each efferent modality contains within its PML an action buffer that stores act impulses resulting from conscious actions the agent has performed, but that have not yet been executed at the lower levels (i.e. they have not yet caused the relevant effectors to have their effects on the world).

Plans for complex acts may be comprised of acts that use different modalities. For instance, depending on the agent's embodiment, moving a grasper and using it may be independent of each other, and therefore use separate modalities. Since these separate modalities operate independently, actions may occur in them simultaneously. For instance, *moving the grasper arm to a position* and *opening the grasper*, if they use separate motors, could very well be performed at the same time without effecting each other's operation. In fact, it would save time to do these steps in parallel if possible. SNeRE does not currently include a

construct that allows act plans to contain instructions like *start performing a1 and a2 at exactly the same time* (e.g., *begin moving the grasper arm to a certain position and, at the same time, begin opening the grasper*). As discussed more below, consciously performing an act that uses some modality results in an act impulse being added to the modality's action buffer. Because modalities - including their buffers and effectors - operate independently, two acts `perform`ed in sequence may actually be carried out in parallel if they use separate modalities. Interesting future work on efferent modalities includes providing constructs within SNeRE to specify that two actions in separate modalities should be performed in strict sequence, or should be performed in parallel. Another way of achieving strict sequencing is to set preconditions for acts that will only hold when the previous act has been successfully performed. One of the themes of Chapter 5 is using afferent modalities in combination with efferent in order to verify the effectiveness of acts.

4.4.1 Efferent Modality Buffers

Like percept buffers, action buffers are located in the PMLb. Acts impulses are added to the buffer at the PMLb as a result of primitive acts that are performed at the PMLa, and are removed and processed by the PMLc, where they are further decomposed into low-level commands suitable for use by the SAL. For instance, a primitive action for a grasping effector might be to move the effector some number of units in a particular direction. When the PML function attached to this action is called, it places a structure representing this impulse and its parameters into the modality's action buffer. As soon as the modality is available (immediately, if it was not already carrying out some action when the `move` action was performed), the structure is removed from the buffer and processed at the PMLc, where it is converted into commands the SAL can execute (e.g. apply a certain amount of voltage to a particular motor for a duration). See Figure 4.2.

The action buffers in efferent modalities have similar properties to the percept buffers in afferent modalities: their capacities are configurable as are their expiration intervals. One difference is that MGLAIR's model of modality focus currently applies only to perception and does not account for focusing on actions in efferent modalities.

When an efferent modality's act buffer is empty (i.e. when the modality is available for immediate use), an act impulse added to the buffer will be removed and executed immediately.

What happens when an action is performed using the modality when its buffer is non-empty? The default behavior is to add the act impulse to the buffer from which it will be removed in turn when the modality is

[4]Gripper image from `http://commons.wikimedia.org/wiki/File:Shadow_Hand_Bulb_large_Alpha.png`, available under the Creative Commons Attribution-Share Alike 3.0 Unported license.

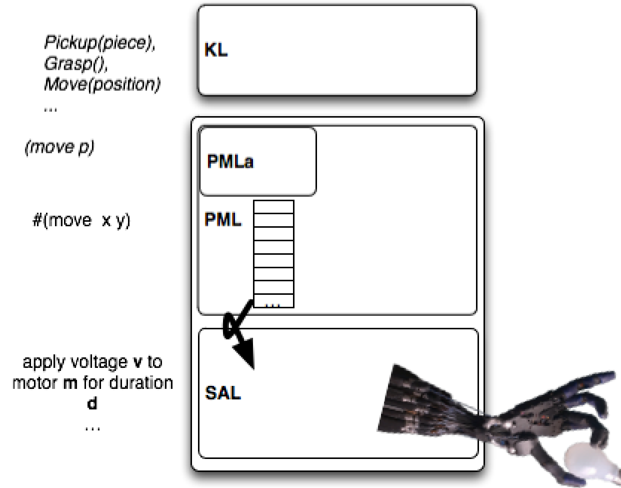

Figure 4.2: View of a single efferent modality for a gripper-effector [4]

available. Another option (determined at the modality level) is for the new impulse to clear the buffer of any existing act impulses. Note that in this case there is no point to having a buffer capacity larger than one. In this case, the new act impulse may either interrupt the current action (if the SAL is carrying out an action), or wait for it to finish. Whether an action within a modality may be interrupted and terminated depends on the embodiment and the nature of the modality. For modalities with durative acts (e.g., a locomotive modality used by an agent to move from one location to another), it makes sense to allow the modality's acts to be interrupted and terminated: an agent that is engaged in moving from one spot to another should be capable of stopping that action if its goals or beliefs change as the result of reasoning or perception. Actions such as opening a gripper might be non-interruptible, either because of the nature of that modality's implementation, or because the agent designer decides to disallow interruption, for instance because it might leave the agent in an unknown or undesirable state.

MGLAIR also allows agents to consciously perform an act to halt and discard all activity in a modality, provided that the agent is consciously aware of that modality (i.e., has symbol for the modality).

4.5 Regulating Modality Focus

As mentioned in 3.8, MGLAIR includes a model of modality focus that allows agents to dedicate more or less of their computational resources to perceiving within a particular afferent modality. An agent designer

may specify a default focus level for a modality and either permit or forbid the agent to adjust its focus levels on its own modalities. This allows agents and agent designers to prioritize or de-prioritize processing of percepts from a particular modality relative to the others, which an agent may use to strategically adjust its perceptual processing depending on its current task and priorities.

This section discusses the focusing mechanism for modalities in detail, with examples and discussion of the effects of their use on an agent's behavior.

4.5.1 Motivation

MGLAIR is specifically designed to support concurrent use of multiple separate modalities in agent implementations. A major advantage of modeling modalities each as its own object with its own processes operating on it is to prevent modalities from interfering with each other. If two different pieces of sensory data in two different modalities - e.g. hearing and vision - are available at roughly the same time, an agent shouldn't have to finish processing and perceiving one of them (the sight, say) before it starts work on perceiving the other (the sound). The structures (e.g. modality buffers) and processes (buffer management processes, perceptual processes) described above operate in parallel to achieve this modality independence.

However, because each agent has only a limited amount of computational resources, it is possible for a modality to interfere with the operation of others by monopolizing those resources (e.g. processors). This can happen when a modality has to work harder than others to convert its sensory data into perceptual knowledge, and it can happen when a modality is simply producing much more, or much richer, sensory data than others.

This type of disparity is common enough: consider a webcam-equipped robotic agent that constantly performs complicated visual tasks with its visual modality (let's say it's on a scavenger hunt) while navigating in its environment, and also possesses a course-grained distance sensor that only produces output when the agent is about to bump into something. In such a case the visual modality is continually transmitting and converting into percepts large amounts of sensory data, while the distance sensor only occasionally sends very simple and easily perceived sensory data. This arrangement may be detrimental to the agent if it needs to avoid bumping into walls.

Without a mechanism to selectively allocate its focus' and emphasize the modality that is more relevant (perhaps urgently so) to its situation, the agent would by default expend most of its processing power on the less relevant modality.

Adjusting properties of the modalities' perceptual buffers (capacity, expiration interval) will help to

achieve reasonable behavior within each modality (e.g. to keep the agent's perceptual processing from getting backed up, and thereby keep it perceiving relatively current occurrences). However, so long as the visual modality in the above example always has something in its perceptual buffer that needs to be perceived, it will tend to consume a large amount of the available processing power. Note that, though this is not necessarily undesirable for all agents and modalities, it will certainly be a problem in the case discussed above if the agent needs to perceive impending bumps via its distance sensor as quickly as possible, for instance, to take some evasive action.

Though this is most obviously a problem when there is a significant disparity between modalities in terms of their perceptual processing time and frequency and richness of sensory data transmitted, the same issue comes up even among more or less equivalent modalities if one of them is producing percepts that are more urgent or more critical to the agent's behavior given its current task and situation.

MGLAIR deals with these issues by providing a model of modality "focus" that allows for a fine-grained prioritization for perceptual processing in afferent modalities by agents and their designers. Modality focus and supporting mechanisms are discussed in more detail below.

4.5.2 Focus

As discussed above, each modality's buffer management and perceptual processing tasks run in separate processes. To prioritize processing perception within a particular modality, MGLAIR elevates the priority of these processes. Internally, each raising (or lowering of focus) for a modality process amounts to halving (or doubling) the amount of time that its processes are inactive between operations.

The system also provides a way for agents to completely ignore a modality, and to resume attending to it. Completely ignoring a modality will be useful more rarely than focusing and unfocusing, but it is at least possible that some agents will occasionally find one sensor or another to be irrelevant to the current task. For instance, an agent that is tasked with navigating some environment (using, in part, a distance sensor), locating some item within the environment, and then manipulating that item from a stationary position, will have little use for the distance sensor while it is stationary. In such a case, it might make sense for the agent to completely ignore that modality for some period of time, freeing up the computational resources that modality would otherwise use for use by its remaining modalities.

Unless otherwise specified at the time the agent is created and its modalities defined, all modalities have the same default focus level. The agent designer may specify a different starting focus level in the modality definition, and may also permit or forbid an agent to change its own focus levels. Additionally, in the current

model, an agent must have been granted conscious access to a modality, and have a symbol by which to refer to it, in order to act to modify the focus on that modality.

This design choice - requiring agents to have a symbol by which to refer to a modality, and to act explicitly act in order to raise or lower focus on the modality - is just one way of handling focus modification. One possible alternative, discussed at the end of this section, would allow agents to adjust focus on a modality without the explicit knowledge that it is a modality.

An agent cannot in a single act set its focus on a modality to any of the available levels. Rather, the agent must either raise or lower its focus by one step at a time. Multiple raisings or lowerings in rapid succession are allowed. There are seven levels of modality focus: three below `normal`, which is the default for every modality that does not specify a starting focus, and three above `normal`. The levels have symbolic labels that agent designers may use when specifying modality focus as part of the modality definition: `lowest`, `low`, `reduced`, `normal`, `elevated`, `high`, `highest`

4.5.3 Effects of Changing Focus

Every modality's perceptual buffer has attached to it a process running the following loop:

```
while true:
    where PS is the oldest, non-expired element from the modality's percept buffer
    remove PS from the percept buffer
    apply the modality's perceptual function to PS
    sleep for a small interval
```

Raising focus on a modality causes this loop to run more frequently for that modality; lowering it has the opposite effect. Modality focus is adjusted by changing the small interval for which the process sleeps in each loop iteration. The interval is computed from a base interval and the modality focus level. Modality focus operates on a log scale: each step up (e.g., from `normal` to `elevated`) corresponds to a halving of the sleep interval in the buffer process.

In my implementation, this is achieved by letting modality focus levels from `lowest`, ..., `highest` correspond internally to the numbers `-3`, ..., `3` respectively. Given some base sleep interval s seconds and focus level l, the interval for which the modality buffer process sleeps is $s \cdot 2^{-l}$ seconds. The internal base interval is currently set to .1 seconds. At the `highest` focus level, the process sleeps for 12.5 milliseconds, at the lowest: 800 milliseconds. Normal use of the architecture should not involve changing the base sleep interval

value.

For some modality `m`, the agent's performing the act `focus(m)` results in the modality's focus moving up to the next level, e.g. from `normal` to `elevated`, (or staying the same if `m`'s focus was already `highest`. When the agent performs `unfocus(m)`, `m`'s focus will move down one level, e.g. from `highest` to `high` – unless `m`'s focus was already `lowest`. When the agent performs `ignore(m)`, its perceptual buffer process is halted completely. `attend(m)` restores `m`'s perceptual buffer process in case it was previously `ignored`. Performing `restore-focus(m)` causes `m`'s focus level to return to its initial value - either the default starting value or the starting value specified by the agent designer.

The result is that, all other things being equal, an agent with modalities `m1` and `m2`, where `m1` has a higher focus level than `m2` will process and perceive sensory data originating in `m1` more often and more quickly than that originating in `m2`. This is useful when perceiving via `m1` is more relevant to the agent's current task and situation than perceiving via `m2`, and it is especially useful to prevent `mod2`'s perceptual from delaying the perception of `m1` in case `m2` is more active or takes longer to process its percepts than `m1` does.

Clearly, there is some interaction between the effects of adjusted focus on a modality and the behavior of its perceptual buffer: all other things being equal, a modality with a higher focus will remove structures from its perceptual buffer with greater frequency, and will convert them into percepts faster than a modality with lower focus. In the case that a perceptual buffer would fill to capacity or reach its expiration interval and begin dropping unperceived structures, raising the modality's focus is another way of counteracting the loss of unperceived sensory data.

4.5.4 Measuring Effects of Changing Focus

The intended effect of raising focus on one modality is to reduce the amount of time it takes for sensory data in that modality to be consciously perceived by the agent compared to the amount of time it would otherwise take in the same situation. This section describes and presents tests confirming that this effect is achieved.

In order to compare the effects of differing focus levels on modalities' transmission of sensory data and processing it into conscious percepts, I built time logging capabilities into modalities. These are not intended for use by agent designers or other users of MGLAIR, nor are they accessible to the agent in any way.

With these logging capabilities enabled, the system records, for each PML structure, the *buffer-in time* - the time at which it is added to the modality's perceptual buffer; the *buffer-out time* - the time at which is is removed from the perceptual buffer; and the *assertion time* - the time at which the corresponding percept

has been asserted at the knowledge layer (perceived by the agent). In this section, I abbreviate these as t_{bin}, t_{bout}, and $t_{asserted}$ respectively.

Logging these allows us to determine how long each individual PML structure spends in its perceptual buffer (t_{bout} -t_{bin}), as well as how long it takes for the modality's perceptual processing function to complete the task of converting it into a percept. The latter, $t_{bout} - t_{asserted}$, includes the time it takes to assert the percept at the knowledge layer.

To test the effects of the focus mechanisms when there is a disparity between the amount of processing two modalities use, I created a simple agent with two modalities, attached to simulated sensors that send a stream of data to each. I will refer to these as the agent's *visual modality* and *distance modality* just to extend the example used earlier. Note that neither is hooked up to a real sensor and the simple agent connected to them does not use their percepts as part of any interesting task. Despite this, the test serves to illustrate the effect of adjusting focus.

The simulated sensor connected to the visual modality sends a burst of sensory data every 100 milliseconds. The visual modality perceptual buffer quickly fills with the corresponding PML structures, ensuring that its perceptual process will never be left waiting for something to do.

Meanwhile, after a short pause to allow the visual modality's buffer to fill, the distance modality's simulated sensor sends ten impulses separated in time by between 2000 and 4000 milliseconds.

Both modalities' perceptual functions perform very simple translations of their perceptual structures into terms that are directly asserted at the knowledge layer.

In one set of tests, both modalities retain the same default focus level (`normal`). In another set, the visual modality stays at `normal` focus, but the distance modality is set to `elevated` focus. In a third set, the visual modality stays at `normal` focus and the distance modality's focus is set to `high`.

Optionally - that is, in some test, but not in others - the visual modality's perceptual function performs a `noop` for a random amount of time between 100 and 300 milliseconds. This is intended to simulate the visual modality's taking a non-trivial amount of time to process and perceive.

Tests for each of the six combinations (equal focus without a visual processing delay, equal focus with a visual processing delay, elevated distance focus without a visual processing delay, etc) were run twice each with the time logging capabilities enabled, and the mean and median time to perceive (i.e. $t_{asserted} - t_{bin}$) for each were recorded.

The result, shown in Figure 4.3 below, confirms the usefulness of the focusing mechanisms.

The vertical axis is milliseconds. Each of the six bars shows the mean time to perceive in the `distance`

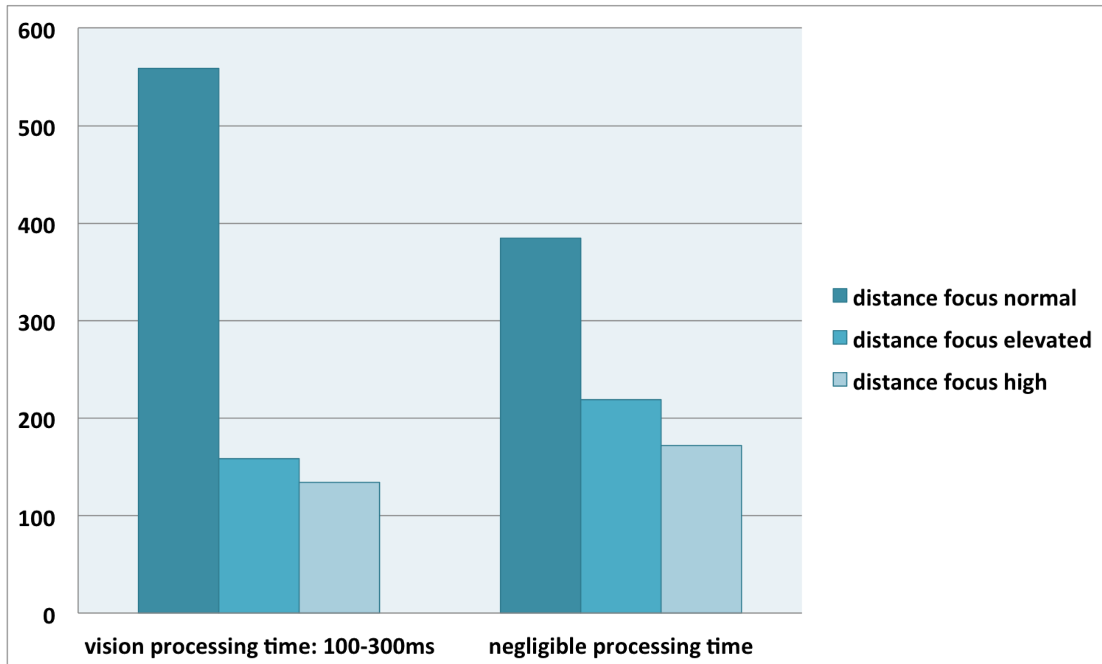

Figure 4.3: Effects of changing focus on one of two modalities

modality under different circumstances. The three bars in the right half of the chart show the mean time to perceive in the `distance` modality when the processing task being performed to perceive in the `vision` modality takes a negligible amount of time (as does the processing task in `distance`). The three bars in the left half of the chart show the mean time to perceive in `distance` when `vision`'s processing task takes a random amount of time between 100 and 300 milliseconds each time it is run. The darkest bars on both sides of the chart show the mean time to perceive in `distance` when `distance` and `vision` have the same focus level of `normal`. The medium and lightest bars show the result when `distance` has focus levels of `elevated` and `high` respectively.

As this shows, especially when the visual modality's perceptual process takes a non-trivial amount of time (with the introduced delay), the mean time to process and assert in the distance modality is quite high when the two have equal focus (around 560 milliseconds). When the distance modality's focus level is raised to `elevated`, its mean time drops to about 160 milliseconds. It drops further to around 130 milliseconds when its focus is raised to `high`.

When the visual modality is not taking extra time for perceptual processing, the results are less dramatic but still striking: the mean time to perceive in the distance modality drops from 385 milliseconds (equal focus) to 219 milliseconds (distance focus `elevated`), to 172 (distance focus `high`).

These results demonstrate that, even in this relatively simple case, adjusting a modality's focus has

the intended effect, and will significantly improve an agent's ability to perceive using the modality with heightened focus when perception would otherwise be delayed. Chapter 5 discusses some example agents that make use of this ability.

4.5.5 Focus Future Work and Conclusions

Focus without Explicit Knowledge of Modalities

The model of modality focus presented here requires the agent to have conscious access to some information about its modality, including a symbol by which to refer to it. Another interesting possibility, not currently implemented, would allow an agent to modify its focus on *percepts* in a modality by performing acts like `focus` on a particular percept or set of percepts, and thereby increase the level of focus on the modality that produced them. Since perceptual beliefs are already internally linked to the modality in which they originated, an implementation of this alternative strategy would be fairly straightforward. An advantage of handling focus in this way is that it would not require agents to know explicitly what their modalities are, or to have names for them. Rather, agents would increase the perceptual processing dedicated to certain types of percepts simply by performing a mental act involving some instances of the types of percepts that are of interest.

Focus and Fatiguing

Section 4.5.1 discusses an example agent with a modality for a distance sensor that only produces output when the agent is very close to an object in front of it and therefore about to bump into something. Consider an agent with very similar sensor with one key difference: this distance sensor, at the SAL, constantly sends sensory data up through its modality – even though its value rarely changes significantly.

In this case, depending on the sensor details and on the agent's current task and situation, the agent may waste a lot of time processing and perceiving sensory data that corresponds to no real change in the state of the world with respect to that particular sensor. Humans and other animals often deal with this problem through sensory adaptation, e.g. olfactory fatiguing, [Steinmetz et al., 1970] [Dalton, 2000] in which the sensory system stops responding to a repeated stimulus.

One MGLAIR approach to modeling this, not currently implemented, would be to introduce an optional fatiguing mechanism within the PML to automatically scale down a modality's focus in case it repeatedly produces sensory data that is too similar (determined by some modality-specific similarity function). Such a fatiguing mechanism might also adjust a modality's buffer properties (e.g. expiration interval) in order to

save resources in this situation.

Conclusions

MGLAIR's model of modality includes a focusing mechanism that, in combination with its flexible perceptual buffer implementation, allows agent designers and, optionally, agents themselves to prioritize perceptual processing in some modalities relative to others. The main advantage of these features is to allow agents to focus on what's important to them given their task and situation in the environment, and to respond promptly to relevant information.

Chapter 5 discusses issues pertaining to the details of sensing and acting in individual modalities, including some case studies that highlight the features discussed in this chapter.

Chapter 5

Sensing and Acting in Modalities

Chapter 4 presents key structures and mechanisms underlying the MGLAIR architecture's model of modality, including perceptual buffers and the processes that operate on them, and focusing mechanisms that allow agents to adjust the amount of resources they use for particular modalities.

This chapter investigates in more detail how sensing and acting work in individual modalities that are parts of agents, and in pairings of efferent and afferent modalities, and presents some examples of modalities from implemented, embodied MGLAIR agents.

5.1 Effectiveness of Actions

The MGLAIR model of modality and its support for concurrent use of multiple modalities allows an agent to have real-time knowledge of what it is doing: whether an action it's started carrying out is still in progress or has finished, how the act is progressing, whether an action had its intended effect, and so on. This is extremely useful under exceptional conditions: an effector or sensor fails, the world fails to meet the agent's expectations, or something else goes wrong within the modality. It is also useful under more normal circumstances: it makes sense for an agent that knows about the intended effect of its actions to use that information, along with one or more of its afferent modalities, to verify that the effect has been achieved. In many cases there is a causal relation between efferent and afferent modalities: moving, for example, will change the agent's location in the world, which will, in turn, change what the agent sees or otherwise senses via afferent modalities that are dependent on its location.

Both the intended effects of actions and the agent's ability to perceive those effects are often specific to

modalities, either to the efferent modality the agent uses to act or to the afferent modality or modalities it can use to perceive the expected changes in the world.

Consider, for example, an agent with the ability to move forward, backward, and to rotate by turning its wheels (e.g. the agent described in Chapter 7 that is embodied in an NXT robot). If the agent is to interact effectively with the world, it should be capable of knowing whether its act impulses — the outputs of its acting system, which travel down through the layers of the architecture and are eventually decomposed into low-level motor commands — are having the intended effect. In case they do not, for one of several reasons, it would be best for the agent to detect this discrepancy between its intended acts (i.e. those acts "perform"ed at the knowledge layer) and the actual movement of its body.

When might this happen? Our simple wheeled agent might be held in the air so that its wheels are spinning uselessly, rather than propelling it forward. More practically, the agent might get caught on something — the leg of a chair, for instance — in such a way that its wheels are moving but its body is not actually moving forward. We can play other tricks on the agent as well: operating it on ice (or another surface with less traction than it expects or is accustomed to), putting it on a treadmill, etc. What will be the result in all of these cases? The agent, when it intends to move, will perform the action by sending the appropriate impulse down through the layers of the architecture, and the wheels will turn in a way that, under more normal conditions, would propel the agent forward along the surface on which it was resting. In order for the agent to know that its action is not having the intended effect, it would be useful for the agent to possess a sense that allows it to distinguish between being in motion and being still, or to have a sense that lets the agent know when its wheels are actually moving. The former would be something like a vestibular sense, the latter a proprioceptive sense for the agent's locomotive effectors.

5.1.1 Effects of Acts

As discussed above, it is desirable for agents to be able to know about the effects of acting in efferent modalities, and to be capable of knowing something about a modality's status (e.g. that an effector is currently moving, or is in a particular position).

If we don't want agents that believe, and behave as if, they are omnipotent (and we don't because they're usually not), a distinction must be made between the intended effects of actions and their actual effects. The effect of an agent's having performed the act "move forward `n` units" using its locomotive modality *should* be that the agent's body inhabits a different spatial region than it did before the act was performed, preferably a spatial region n units in front of the region it inhabited just before before performing the act. This is the

intended effect of the act. There are a multitude of ways this act can fail to have its intended effect. Even if everything goes as planned and the act has exactly the intended effect, it may take some non-trivial amount of time to complete.

SNeRE includes the `Effect` construct, which can be used to connect acts and propositions about their effects:

> **Effect(a, p)**: The effect of performing the act a is that the proposition p will hold. The SNeRE executive ... will automatically cause p to be asserted after a is performed. [Shapiro and The SNePS Implementation Group, 2007]

Though this is useful in planning and executing plans, agents in general should not believe that their acts have successfully and immediately accomplished their intended effect without verifying that whenever possible.

Sometimes it can make sense for an agent to expect its actions to always have the intended effects, even for an embodied agent operating with a simple simulated body in a simple simulated world, like the world used by the SNePS Wumpus World Agent [Shapiro and Kandefer, 2005]. If an agent's performing a grabbing action in the presence of gold will *always* result in the agent's having the gold in its grabber, then it makes sense to design such an agent to believe that it has the gold after grabbing.

However, for any more complex world, embodiment, or task, the agent should generally take a more skeptical view of its own abilities. The act of grasping an object can fail in any number of ways: the agent may be in slightly the wrong position relative to the object or it may be mistaken about the location of the object or about the orientation of its grasper; the grasper might be less precise than is desirable, or be prone to an occasional glitch.

Particularly when the agent's embodiment allows it to verify the efficacy of an act, it should avoid automatically concluding that the intended effect of performing an act has been achieved just because the act has been performed.

5.1.2 Act Completion

A related issue is knowledge of act completion. When used to act, efferent modalities send act impulses down through the layers of the architecture using their asynchronous data channels. Agent designers (and agents) should not assume that the agent's `move` act, for instance, will have completed just because the PMLa function that implements it has returned. This is especially relevant when dealing with complex acts, such as sequences of acts, in which the completion — successful or otherwise — of one act is a prerequisite

for starting another, or in other situations where the intent is for the agent to first complete one act before going on to the next.

For instance, an agent that is to go from point A to point B in sequence should verify that it has successfully arrived at point A before it begins moving to point B. If, instead, the agent considers the act "go to A" to have succeeded while it is still being carried out, it may start to go to A and then interrupt that action to go to B before ever arriving at A. Both the ability to sense that the efferent modality is still in use, and the ability to verify the effects of an action using one or more perceptual capabilities, depend on the agent's having some information about the connection between two or more modalities. MGLAIR's support for ways of expressing and implementing this knowledge is discussed in detail below.

5.2 Connecting Modalities

While it is in the process of moving around, an agent with basic proprioceptive and vestibular senses has several ways of knowing what is actually happening to its body and the world – and therefore, of drawing conclusions about the effectiveness of its actions.

First, the agent may consciously know, without sensing, what it is doing or trying to do because it has consciously begun carrying out a particular act and has not become aware that that act has finished, or has not consciously terminated it. Second, an agent using a proprioceptive modality might feel, and thereby know, whether its locomotive actuators are actually moving. Third, the agent can feel through the use of its vestibular modality whether its body is actually in motion. An agent with a visual system and an implementation of simple visual ego-motion estimation capabilities might see its field of vision changing in a way that is consistent with what it expects when it is moving, and take this as evidence that it is actually in motion. There are numerous other possibilities depending on the agent's afferent and efferent capabilities.

This ability to know, via different perceptual modalities, what its body is currently doing allows an agent to reason and act more robustly, and with a better picture of the world and its relations to the world, than it otherwise could. An agent equipped with these sensory capabilties and with some knowledge of the connection between sensing and acting in the relevant modalities is capable of realizing when something is wrong that its actions are not having the intended effects. The connecting knowledge of a causal relation between modalities may be explicitly specified by the agent designer, as in the case of the proprioceptive modality discussed below, but we also allow for the possibility of making a less explicit connection and allowing the agent to learn how a particular afferent modality is influenced by the use of an efferent one, as in the example vestibular modality, which is discussed more below.

5.2.1 Proprioceptive Sense

The NXT robot discussed in full in Chapter 7 has a wheeled locomotive modality that uses motors with rotation sensors whose values change each time the motor (wheel) moves. The agent is equipped with a proprioceptive afferent modality that allows it to sense when its wheels are in motion. This is of immediate use, for instance, in giving the agent a straightforward way to tell when a locomotive act has completed. During the act, the agent feels that the motors are operating (by believing, at the KL, things like `Moving(I)`). When the motors have stopped moving, the agent experiences that as well (believing `~Moving(I)`).

5.2.2 Vestibular Sense

Another MGLAIR agent is embodied in an iRobot Create with a tri-axial accelerometer. The accelerometer is used to give the agent a vestibular sense that allows it to feel when it is in motion.

The agent learns what different locomotive actions feel like by acting – driving around with the use of its locomotive modality. During training a classifier in the agent's PMLc learns classes that represent the agent's sensations of moving forward, turning, stopping, etc. The classifier is trained by giving it access to symbolic representations of the agent's act impulses when they are carried out at the PMLc. These are used as labels for the incoming sensor data for the classes of motion it learns. The remaining input to the classifier is a set of features derived from the raw sensor data over a short window of time, including the mean and peak acceleration along each axis.

During normal (post-training) operation, the classifier in the PMLc for the vestibular modality takes as input feature vectors derived from raw sense data at the SAL, and produces as output activity descriptions that mirror the PMLb's representation of locomotive acts. These are propagated up to the PMLb and PMLs, where they are converted to symbolic propositional knowledge in the agent's language of thought that expresses, e.g., "I am turning."

Section 7.2.1 presents more details on the MGLAIR Create agent and on the the implementation of its vestibular modality.

5.2.3 Knowledge of Modalities' Connections

Acts performed by embodied agents change the environment that the agent is operating in. This change in the state of the world will often be perceptible to the agent via one or more of its afferent modalities. These are, of course, also perceptible to other observant agents in the same world. Control acts, on the other hand,

take place at the knowledge layer and are therefore private, directly affecting the contents of the agent's mind.

Some acts, however, seem to lie in the middle between these two possibilities: acts that change the state of the agent's body in ways that only the agent can directly experience. Examples include: the agent's turning its camera, moving its wheels, manipulating a grasping effector, and so on. While these actions may also change something about the world other than the spatial region the agent's body inhabits, it is worth noting that they may also have effects that can only be directly observed or experienced by the agent itself. That is, while any observer might be able to tell by looking at it that the agent's wheels are turning, an agent equipped with the appropriate sensor connected as an MGLAIR modality can directly observe that its wheel *feels* like it is moving. One might claim that the agent has a private qualitative experience of the motion of its wheels that might very well constitute something like qualia — but I make no such claim here.

In any case, this connection between the agent's ability to act in one efferent modality and to directly sense the action via a related afferent modality seems to be a more specific special case of the general situation where one modality's use may change what is perceived via others.

MGLAIR provides several ways of making the connection between modalities, which are outlined in the following sections.

Ignore the Connection at the Knowledge Layer

One way of handling this connection in an agent implementation is to simply ignore it within the implementation of the agent's Knowledge Layer: use the two related modalities (*locomotion* and *locomotive-proprioception*, for instance) and arrange things so that when the agent performs `move(...)`, its wheels turn; and when its wheels are turning, the agent perceives `Moving(I,NOW)` or similar. With this simple approach, the agent will know when it's moving and when it's not — which can be very useful information — but it will not necessarily know that there's a connection between acts in its `locomotive` modality and percepts in its `locomotive-proprioception` modality.

This simple arrangement could be extended for use with an agent that is initially given no information about connections between its modalities, but is designed to *discover* connections between them, for instance by carrying out experiments wherein the agent systematically acts and observes its sensations. Humans and other animals learn motor control this way (i.e., we are not "booted up" with mature routines already in place), and such an agent would be an interesting use of MGLAIR and its model of modality. I leave that as a possibility for future work.

Making Connections Explicit at Knowledge Layer

Another possibility is to handle making the connection between two modalities completely at the knowledge layer by giving the agent explicit knowledge that, e.g., use of the `locomotive` modality will change what is perceived via the `locomotive-proprioception`. A full treatment of this would involve temporal aspects, e.g. an assertion to the effect that *while* the agent is performing `move(...)`, it should feel `Moving(I,...)`. An agent thus equipped with explicit knowledge of the connection might use that information to determine that its act has completed:[1] if you begin to perform `move`, then immediately feel that you're `Moving`, then stop feeling that you're `Moving`, then it may be reasonable to conclude that you've completed that act.

Inferring Connections from Background Knowledge

An alternative to giving the agent no information whatsoever about connections between its modalities is to give it explicit general knowledge about the modalities and how they interact with the world. For instance, we might create an agent whose model of the world includes a rich enough representation of space to represent (1) basic information about its locomotive capability and its effect on the world: that its use changes the spatial region that the agent's body inhabits; and (2) basic information about its visual modality and how the world impinges on it: that the location and orientation of the agent's camera (part of its visual modality, which the agent uses to see the world) will determine in part what the agent sees. Either of these pieces of information will allow the agent to make better sense of the use of the modality it concerns. Combined, they entail facts about a causal connection between the two modalities: since moving changes the agent's location, and the agent's location determines what it can see, moving will change what the agent sees. This sort of knowledge not only allows the agent to plan and predict more effectively, it also provides the agent with a way of checking whether locomotive acts are having the intended effects.

The agent described above with a vestibular modality has a built-in knowledge that there's a connection between that modality and its locomotive modality, but the agent does not know initially in what way they are related. During training, the agent learns what it feels like to carry out each of the available actions in its locomotive modality by performing them repeatedly while observing and forming an associating with the vestibular modality.

One possibility for future work on this is to build into modalities information about how they interact with the world, for instance by embedding in modality descriptions background knowledge through, e.g. references to external ontologies such as DOLCE, where useful.

[1] The architecture does not allow one to assume that a primitive act, for instance, has terminated just because the function that implements it has returned

5.2.4 Interrupting Actions

As discussed above, an agent that knows what it is currently doing and about the connections between its actions, their intended effects, and that has a way of checking these may be expected to adjust what it is doing in response to changing circumstances. One such adjustment is the discontinuation of an action the agent has begun.

We want agents to be capable of discontinuing action because an action might fail to have its intended effect due to some exceptional condition (e.g. the agent is trying to move forward but is caught on an obstacle that is preventing this), or because the state of the world has changed in a way that was not anticipated (e.g. the agent is reaching out with its effector to grab an object that was stationary, but the object has now moved out of its reach). In such cases, it will often make sense for the agent to stop what it is doing and perform some new, more appropriate action after reassessing the situation (e.g., go around the obstacle then continue on to its destination; move to the new location of the object before trying to grab it, or pick a different object, or give up and do something else).

Because MGLAIR agents sense and act in multiple modalities simultaneously, they can change their goals, beliefs, or intentions as a result of perception and inference while carrying out an action. For example, an agent that intended to pick up object A across the room, and so performed the high-level composite action `retrieve(A)`, but has just realized as a result of inference that it doesn't really need A after all, might very well want to stop the action `retrieve(A)` (so long as doing so doesn't leave the agent in an unrecoverable or dangerous state or have other undesirable effects).

Alternatively, perhaps object A is still suitable, but the agent has noticed an object B that is just as good but happens to be much closer to the agent's current location. If the agent is in a hurry or is designed to conserve its energy when possible, halting the complex action `retrieve(A)` – which most likely has a bunch of constituent acts like `goto(A)`, `pickup(A)`, etc – and performing `retrieve(B)` instead might be the best approach.

As discussed in Chapters 3 and 4, efferent modalities' data channels provide mechanisms at a low level for canceling primitive act impulses resulting from actions that the agent has consciously performed, but that have not yet been realized at the SAL (and hence not yet actually performed by its body). These mechanisms are used to deal with conflicts between impulses that arise from, e.g., an agent's consciously performing actions more rapidly than its body can implement them.

This functionality, provided by these conflict-handling mechanisms, is also used internally within the architecture to implement conscious termination of actions that are under way.

5.3 Passive and Active Sensing

Some pre-MGLAIR agents use a form of active sensing, in which performing a primitive act simply activates a procedure at the PMLa or PMLb that accesses the world to get the information it needs and asserts the results back to the KL. For example, one agent embodied in a software robot that navigates a simulated environment containing halls of numbered rooms has a primitive action to *read the room number* of the room currently in front of it. In one implementation of this agent, that primitive action reaches all the way from the agent's PMLa down into the world, and retrieves the value of the room number, then asserts that it is the room number the agent is facing. This type of active sensing - where the agent consciously acts in order to perceive something, and the perception is a direct result of the act implemented in the same function that realizes the act - does not fit with MGLAIR's model of modality, which maintains a strict separation between sensing and acting for embodied agents by dividing the agent's basic capabilities into separate modalities based on the physical resources they use.

MGLAIR's afferent modalities use passive sensing: the world impinges on the agent's sensors, which convey data upward through the attached modalities until it is consciously perceived at the KL. By default, this happens without the agent explicitly trying to do anything. Though an agent's acts will often change what it perceives, the agent usually does not have to explicitly perform an act (e.g. `look`) in order to sense (`see`), because its visual apparatus is continually operating (it must, of course, explicitly perform some act in order to, reposition its sensors, or turn them on and off, etc).

Active sensing is achieved in MGLAIR by combining afferent and efferent modalities. If an agent needs to perceive using a particular (afferent) modality only when it has performed some action using an efferent modality, then the agent's PMLc or SAL must make this connection. Figure 5.1 shows the Greenfoot-based Delivery agent presented in §7.3 as the agent is standing facing a room with room number over its entrance. This agent perceives some visual phenomena passively, e.g. what type of thing is currently in front of the agent (a wall, a room, etc), but does not read room numbers unless it acts to do so. If the agent is to read the number on a room in front of it only when it decides to actively perform a certain act (`readRoomNumber()`), then that act should be defined and associated with a modality (perhaps called `visual-efferent`. The implementation of the corresponding activity at the agent's SAL should activate the visual mechanism for reading room numbers, resulting in the appropriate sensory data traveling up the agent's visual modality. In the Delivery agent operating in a 2D simulated world, this involves having the PMLc pass the impulse for reading room numbers to the appropriate method at the SAL for the agent's visual modality. In a hardware robotic implementation, the PMLc would similarly pass an impulse to the SAL that turns on a sensor or

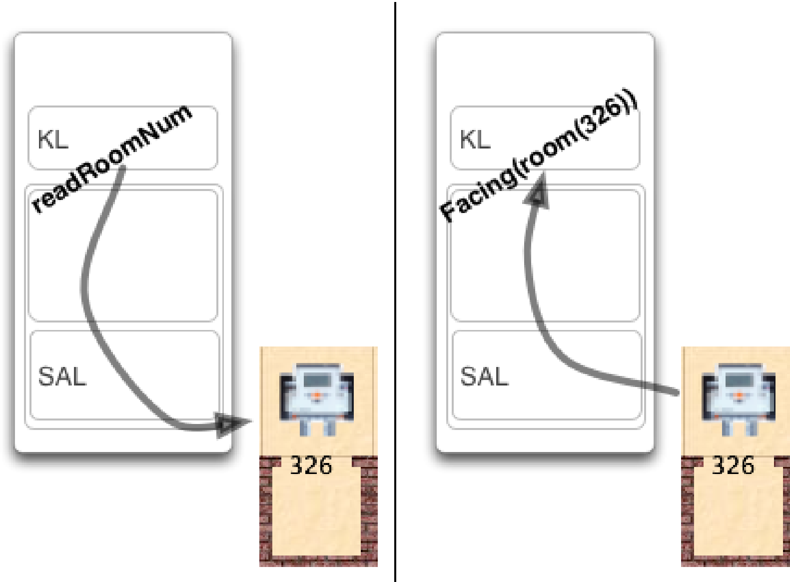

Figure 5.1: Active sensing in MGLAIR. Left: act impulse in efferent modality. Right: resulting percept from afferent modality

modifies its operation.

An additional possibility, not currently implemented, for active sensing is to have the act that triggers the sensing task modify the corresponding functions that implement the agent's perceptual processing in the afferent modality. It would be inefficient, impractical, and unrealistic to have an agent that tries to read *everything* as if it were text — blank walls, empty corridors, etc — especially if the perceptual processing that goes into recognizing and perceiving letters and words is intensive. That is, if the agent's visual SAL sends sensory data up through the PMLc that consists of a set of visual features (lines, etc), we might want the agent to go to the effort of figuring out what words it's seeing only when it's engaged in reading. Such a mechanism for acting to alter perception would also be useful in the development of a full MGLAIR model of attention.

5.4 Modality Registers

In existing GLAIR agents, which lack MGLAIR's model of modality, "modality registers" are used to keep track of what the agent is currently doing (per effector) or perceiving (per sensor) [Shapiro and Ismail, 2003]. In the case of some efferent modalities, the register's value is set by the procedure that implements the action [Shapiro and Ismail, 2001]. That is, agents can reflect and report on what they are doing because the procedures they perform to act directly modify the registers to record the action.

MGLAIR includes, as part of modalities, modality registers that store information about the agent's current use of the modality. Setting the value of this register based on what the agent is currently trying to do with the modality (or on what the agent has started doing with the modality and does not yet know that it's stopped doing) is useful because it provides yet another way for the agent to know about what it's currently doing: by storing the most recently performed act for each efferent modality, we expand the agent's ability to reason about its actions, and to compare what it senses e.g., through a related proprioceptive afferent modality, with what it knows, expects, etc.

Afferent modality registers store the KL term for the most recently perceived item. Efferent modality registers store the KL term for the most recently executed action.

The use of an efferent modality buffer for a particular modality is activated by explicitly attaching primitive acts to that modality at the PMLa. When an act is performed, the KL term denoting that act is automatically stored in the modality register for the corresponding modality. For example, consider an agent with a `locomotive` modality that includes an act `move`, which takes a direction as a parameter. When the agent performs `move(forward)`, the procedure for the primitive action at its PMLa is executed and the term `move(forward)` is added to the modality's register.

5.4.1 Managing Modality Registers

By default, the term stored in a modality buffer remains in place until a new term replaces it. This happens for afferent modalities when a new percept is produced within the same modality, and for efferent when a different action is performed using the modality.

For an agent that's not making very active use of some modalities, the information stored in the registers for those modalities may become stale. It is not ideal for an agent that performed `move(forward)` five minutes ago - and hasn't used that modality since - to report that it is moving forward. However, in some cases, it will make sense to keep - and use - "old" terms in the modality register. For instance, a visual modality register that stores a term for the thing the agent is currently looking at will remain relevant unless the agent's field of vision has changed – in which case, the old term will be replaced.

One alternative proposal, not currently implemented, for the removal of terms from modality registers is the use of a timer attached to each term when it arrives in the register, and which causes the term to be removed when the timer expires. Another alternative for efferent modalities with corresponding (proprioceptive) afferent modalities is to attach a function that watches the afferent modality for changes indicative of an end to the action being performed in the efferent. For instance, consider an efferent `locomotive` modality

and an afferent `locomotive-proprioception` modality. The latter allows the agent to perceive via sensors in its wheel when actions like `move(forward)` start and end (and when they are under way). If the term `move(forward)` is removed from the `locomotive` modality register as a result of perceiving that the motion of its wheels has come to an end, then such an agent would have very accurate and up-to-date information stored in its efferent `locomotive` modality register. Note that with this arrangement, the fact that its wheels are no longer moving is already consciously known to the agent as soon as it is the case because the agent is processing and perceiving sensations from the `locomotive-proprioception` modality. That is, the agent will know `Moving()` if it has been set up to perceive that. However, without a term for its action stored in the modality register the agent might be able to report that it is moving its wheels, but it won't be able to explain that in terms of the act it is performing to realize the motion.

5.5 Modality Examples

This section further illustrates the use of modalities in MGLAIR by presenting example modalities from implemented agents, and highlighting the modality design issues and patterns discussed above. It includes more detailed descriptions of modality implementations than appear in earlier chapters and sections. These include simple afferent and efferent modalities, as well as the use of afferent and efferent modalities in combination as parts of agents.

5.5.1 NXT Agent

The example modalities discussed in this section are implemented as part of an agent embodied in the NXT robot discussed in more detail in Chapter 7, which gives full implementation details of this agent and all its modalities.

The NXT platform is very flexible and allows the creation of simple hardware robots with a number of different sensors and effectors in many different configurations. Its servo motor-based actuators can be used to construct a number of different types of effectors, including the locomotive and grasping effectors discussed here. It also includes sensors, such as a touch sensor, and an ultrasonic range sensor, among others.

The NXT sensors and effectors attach to a small programmable computer that implements the agent's SAL, accessible by a Python API.[2] The agent's PMLc is implemented in python and uses nxt-python to communicate with the hardware over a USB port.

[2] `https://code.google.com/p/nxt-python/`

For instance, the following line of Python gets the current value from the ultrasonic range sensor on the NXT's Port 4:

```
nxt.sensor.Ultrasonic(self.b, PORT_4).get_sample()
```

The following causes the motor plugged into Port A to turn 25 degrees (at power 10 out of 128):

```
nxt.motor.Motor(self.b, PORT_A).turn(10, 25)
```

The agent embodied in this robot includes the following efferent modalities:

- Locomotion - `move forward/backward`, `turn left/right`
- Grabber - `grab`, `release`

Its afferent modalities include:

- Locomotion-proprioception - `Moving(forward)`
- Vision - `DistIs(near/far/etc)`

These are described in more detail in the following sections.

5.5.2 Afferent Range Sensor Modality

The robot's range sensor broadcasts and receives ultrasound waves, producing as output scalar values in the range of 0 to 255 that indicate how far away in front of the sensor is the nearest solid object, where 0 indicates that there is a solid object directly in front of the sensor, and 255 indicates that the nearest object in front of the sensor is as far away as the sensor can detect (about 2.5 meters). The agent is configured to use this modality passively. That is, the agent senses the distance to objects in front of it automatically and without performing any particular act to get that information.

In order for the agent to use this sensor to navigate, it must be capable of becoming aware of (i.e. perceiving) the information about the world that this sensor provides. The agent has assigned to this sensor an afferent modality. It is more crucial to the agent's navigation that it have updated knowledge of the current state of the world with respect to objects in front of it, than that the agent perceives every possible change. A sensible configuration for this modality given this constraint is to give it a buffer with a very small capacity and to have arriving data from the sensor crowd out the oldest unprocessed structure in the buffer, by setting it's conflict handling to `squeeze`. The buffer expiration interval for this modality is not specified below, and so will default to `unlimited`. Perceiving with this modality is relatively simple. Because we want the agent to always avoid banging into solid objects, the information that the modality conveys is of

high priority, so it's focus - which the agent is permitted to modify - is set to `high`. The following shows the definition for the range-sensor's modality.

```
(define-modality 'range
  :type 'afferent-modality
  :predicates '(DistIs)
  :access t
  :conflict 'squeeze
  :buffer '((capacity . 2))
  :focus '(t . high)
  :channel '((port . 9574) (host . localhost))
  :description "range-sensing modality used to detect what's in front of the agent")
```

My implementation of the `range` modality includes a Python class (`UltraSonicSensor`) that implements a wrapper around the sensor at the SAL, and is configured to send a message up to the PMLc when the range sensor's value changes. The API for accessing the sensor only produces a value when asked – i.e. when `get_sample()` is called on the sensor. `UltraSonicSensor` runs in its own thread and polls the sensor for its value every once in a while. When a new value is available, that value is passed up to the Python PMLc, which prepares it for consumption by the PMLb and sends it into the modality's data channel. As discussed more in the next two chapters, this pattern is used repeatedly across implemented MGLAIR agents: a method to get a sensor value is wrapped in a thread that monitors its value and passes the result along the PMLc.

Once the message crosses the modality data channel, a process dedicated to the `range` modality removes it from the data channel, adds a timestamp, and puts the result into the modality's buffer. The data is added to the buffer in accordance with the buffer specification. Here, that means it is placed immediately into the buffer if the buffer has room. Otherwise, if the buffer does not have room, the oldest item still in the buffer is removed and the new element is added. Though this modality has a small buffer, it has to perform only simple perceptual processing on sensory data emerging from the buffer, and the modality is operating with high focus. If, during the course of operation, the agent designer notices that this buffer is often full, or that many sensory impulses are being discarded before they can be perceived by the agent, the buffer's properties or the modality's focus can be adjusted to avoid this.

If the timestamped sensory structure makes it to the front of the buffer without being discarded, it is

removed by the `range` modality's `sense-handler` running in the modality's dedicated perceptual process. This process will carry the structure forward through the rest of its transformation until it is consciously perceived at the KL. The sense handler function removes the timestamp and passes the structure to the modality's PMLs function, which performs the final steps to convert it into a term in the KL language (e.g. `DistIs(far)`, and then asserts it. The agent believes that the distance to the next thing in front of it is only ever at most one of `close`, `medium`, `far`, and `distant`. If it already believes that one of these is the case, then perceiving a new value will result in a conflict that is resolved by the removal of the previously-existing belief.

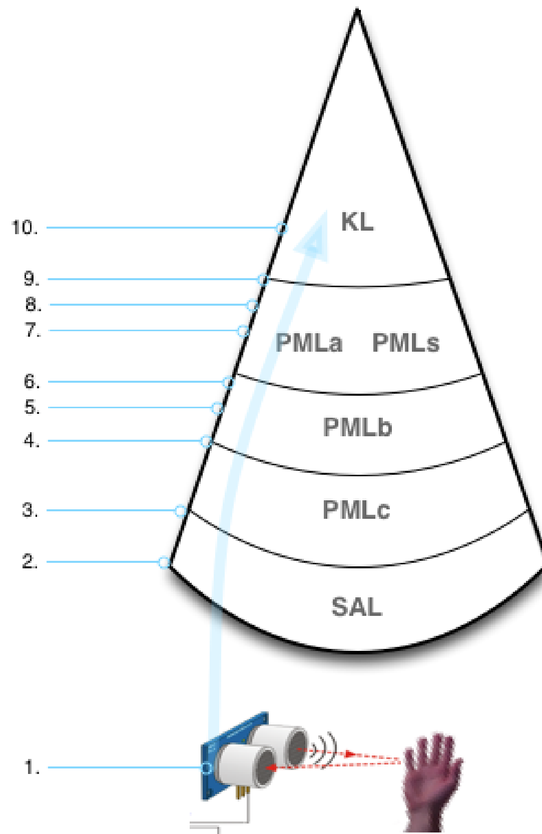

Figure 5.2: Perceiving distance to a solid object with the range modality

Figure 5.2 has labels for the major steps involved in the agent's coming to perceive the distance. These are described below.

1. There is an object in front of the sensor, and the sensor is sending ultrasound waves, and measuring how long it takes them to bounce off the object, in order to determine the distance to the object. Let's say the object is about 0.2 m away.

2. The `UltraSonicSensor` thread polls the sensor and gets its value: `24`.

3. The output value `24` from the sensor is passed to the PMLc.

4. The PMLc puts the value in the modality's data channel by calling the `setSense` method on the `range` modality object. Before doing that, it makes sure the message is in the format the PMLb is expecting: "(d . 25)"

5. The data channel reader process reads the value from the modality's data channel, timestamps it, and adds the result to the modality's buffer: `(PMLb:add-to-buffer (61378511662 . ''(d . 88)''))`.

6. Assuming it isn't pushed out of the buffer by incoming data before it can be perceived, the perceptual process removes the structure from the buffer when its turn comes, and passes it to the sense handler in the PMLb: `(PMLb:sense-distance (61378511662 . ''(d . 88)''))`.

7. The PMLb sense handler passes the structure to the PMLs perception function: `(PMLs:perceive-distance 88)`. Here the PMLb function has removed the timestamp and the `d`, which indicates that it's a distance – `perceive-distance` is only expecting distances.

8. The PMLs perception function finishes decomposing the structure and translates it into a KL term: `DistIs(near)` (the function groups the scalar distance values into one of a handful of rough distance terms the agent knows about).

9. The PMLs perception function asserts the term to the KL: `perform believe(DistIs(near))`.

10. The agent now believes `DistIs(near)` and has disbelieved `DistIs(far)`, etc if it previously held any of them.

5.5.3 Efferent Grabber Modality

The NXT agent's grasping effector is implemented using a single servo motor that opens and closes a claw-like grabber attached to the front of the robot. To use this effector, the agent must have a way of acting to open and close the grabber.

The agent operates this effector by consciously `perform`ing acts at the KL that use the `grabber` efferent modality, e.g. `perform(grab()`. The representation of this action at the KL has attached to it a PMLa primitive action, which is a function that the SNePS acting system calls to execute the action whenever `grab()` is performed. `PMLa:grab-act` calls the appropriate PMLb function, passing it the name of the

modality (`grabber`) in which the act is taking place. At the PMLb, the function `PMLb:execute` is called, which sends the impulse into the modality's buffer. If it survives the modality's conflict handling and buffer handling constraints, it is passed on into the modality's data channel. A separate thread for each modality at the PMLc reads the modality's data channel and converts incoming impulses into SAL commands.

The definition for the `grabber` modality follows. It is set to resolve conflicting access attempts by blocking them. That is, if the agent performs `grab()` and then `release()` in such rapid succession that the grabber hasn't yet finished moving to carry out the first action when the impulse for the second action arrives at the PMLb, the second is simply discarded. Especially when this mode of modality conflict handling is selected, agent architects should avoid creating agents that believe without verifying that the effects of their actions have occurred.

```
(define-modality 'grabber
  :type 'efferent-modality
  :predicates '(grab release)
  :conflict 'block
  :access nil
  :channel '((port . 9577) (host . localhost))
  :description "Used to manipulate the agent's grasping effector")
```

Figure 5.3 highlights these steps for a specific example.

1. The agent decides to grab – perhaps it has discovered that there's an object within reach such that grasping that object will fulfill one of the agent's goals. It acts to achieve this: `perform grab()`.

2. SNeRE's `execute-primaction` function causes the PMLs primitive action function `grab-act` to be evaluated. Unlike other actions, e.g. locomotive acts discussed in §5.5.4, the `grabber` modality's actions are simple enough that they require no arguments.

3. The PMLa function calls a PMLb function that implements the grabbing action: `PMLb::grab('grabber t)`.

4. `PMLb:grab` prepares an impulse that the PMLc will understand and passes it to `PMLb:execute`: `(PMLb:execute 'grabber ''(g . True)''`, which adds the impulse to the modality buffer (if the buffer's configuration and current state allow it).

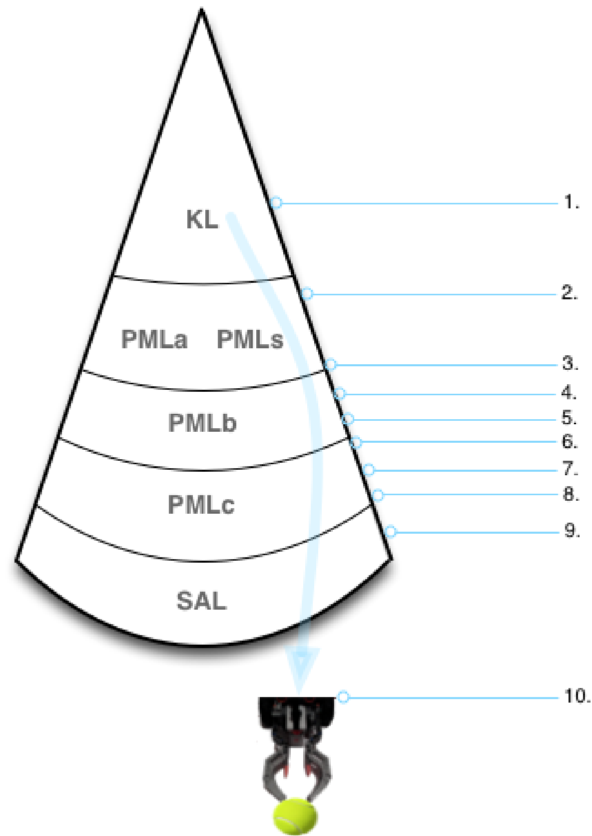

Figure 5.3: Operating the grasping effector

5. A timestamp is added to the impulse and it (e.g. `(61378511662 . ''(g . True)'')`) moves through the buffer.

6. Another process removes the timestamped impulse from the buffer and adds it to the modality data channel.

7. The impulse is removed from the modality data channel at the PMLc by the thread that watches the buffer, and passed to an impulse handling method in the PMLc:

   `pmlc.handleImpulse(''grabber'', ''(61378511662 . (g . True)'')`

8. The PMLc impulse handling method examines the impulse and converts it into SAL commands: `bot.closeGrabber()`

9. The SAL method sends the corresponding command to the relevant motor:

   `Motor(self.b, PORT_A).turn(10, 25)`

10. Voltage is applied the motor that operates the effector and the grasper closes.

5.5.4 Locomotion & Proprioception

Full implementation details of the NXT agent's locomotive, locomotive-proprioception, and other modalities are found in Chapter 7 and Appendix B. The `locomotive` and `locomotive-proprioception` modalities operate much like any other efferent and efferent modalities.

The bot's locomotive capability is implemented using two servo motors identical to the one used to operate its grasping effector. Each of these is connected to a driving wheel. Though these can be operated independently, they are combined into a single modality for this agent. That is, the agent can perform acts like `go(forward)` or `turn(left)`, which each use both of the agent's wheels, but the agent is not designed with the ability to move just one of the wheels at a time.

These servo motors also function as sensors in that it is possible to get from them a reading for a motor's rotational position. The agent's afferent `locomotive-proprioception` modality is tied to the servo sensors, thereby providing the agent with feedback on the use of its wheels. The agent itself does not constantly know the rotational position of each of its wheels – though that would be one possible way of implementing it. I have chosen instead to have the agent perceive simply *that its wheels are moving*. This is achieved within the Python code at the SAL as follows. When the agent acts in its `locomotive` modality, for instance to `go(forward)`, the impulse to move travels down through the layers of the architecture much like the impulse to close the grasping effector described in detail in §. When it gets to the PMLc/SAL interface, the Python method that implements moving forward in terms of sending instructions to the NXT micro controller also spawns a separate thread to monitor the tachometric data from the wheel motors. This thread monitors the sensors, sending a sensory impulse indicating that the wheels are in motion up through the afferent `locomotive-proprioception` modality. When the wheels stop moving, the thread sends a final impulse indicating that they have stopped, and then exits. The agent, of course, is privy to none of these details: it perceives using its `locomotive-proprioception` modality propositions like `Moving()` or `Moving()`, which mean that the agent's wheels are currently moving, or that it's wheel are not currently moving, respectively. If needed, this sense could be extended to allow the agent to perceive how fast it is moving, or in which direction. See Chapter 7 for implementation and other specific details of this capability.

5.5.5 Active Vision in Delivery Agent

The MGLAIR Greenfoot delivery agent operates in a 2D grid-based world that simulates the floor of a building. The simulation consists of corridors, and rooms that have entrances on the corridors. The rooms are numbered, and some of them contain packages. The agent uses its grasping effector to pick up packages

and put them down in the course of delivering packages from one room to another. Using its visual modality, the agent can read the room numbers and see what is ahead in the cell it is facing (options include a wall, an entrance to a room, or part of a corridor. This agent passively senses what is in front of it (e.g. `Facing(room)` when it is facing the entrance to a room), but must actively sense in order to read the room number (which only works when it is facing a room, of course).

As with proprioception, active sensing involves a connection between an efferent modality and afferent modality: the Delivery agent uses an efferent modality called `visual-efferent` in order to actively perform sensory actions such as reading the room number when it is facing a room's entrance. In active sensing, the agent acts using one efferent modality in order to change what it perceives in some afferent modality, rather than to effect some change in the world. In proprioception, the agent also perceives something in an afferent modality as a consequence of acting in the related efferent modality, but that action is usually taken for some reason other than to have that perceptual experience. The perceptual experience may be used to verify that an action is having an intended effect, but the reason for the action from the agent's perspective is to bring about a state of affairs where the agent's body is in a different location, or where its grabber is holding something it was not holding before, and so on.

When the agent performs `readRoomNumber()` the act impulse in the `visual-efferent` modality travels down through the layers of the architecture to the (Java-based) SAL implementing the simulated robot's body. The Java method that implements visual-efferent acts causes a sensory impulse representing the agent's view of the room number to be sent up through the afferent `visual` modality. Eventually, this results in the evaluation of the PMLs function for visual perception (`PML:perceive-vision`), which converts the structure into a conscious percept like (`Facing(room(5))`, which it then asserts. The result is that when the agent decides to perceive a particular kind of thing and performs the appropriate action, it will shortly after have the matching visual perceptual experience – so long as the world and its sensors are properly aligned for the agent to, e.g. actually read a room number.

5.6 Conclusions

MGLAIR agents use concurrently-operating afferent and efferent modalities and combinations thereof in the service of sensing, reasoning, and acting. In addition to allowing for simultaneous action and perception using multiple of the agent's capabilities, these modalities provide the agent with a more complete view of the world and of its situation within the world than it would otherwise have. Chapter 6 builds on the examples given in this chapter, describing the implementation of MGLAIR, the functionality available to

agent-architects, and how it is used. Chapter 7 further illustrates the use of MGLAIR through examples and implementation details of complete MGLAIR agents and the modalities they use.

Chapter 6

MGLAIR Implementation

6.1 Introduction

This chapter presents the details of my implementation of the MGLAIR architecture. The implementation of MGLAIR described in this chapter deals primarily with the use of modalities and with structures and mechanisms that support their use. All of the internal work involved in handling modalities - connecting their data channels, managing their buffers and focus levels, and so on - is implemented and available as software that agent architects use in combination with the SNePS Knowledge Representation and Reasoning System and its subsystems to create MGLAIR agents. In this chapter, I describe MGLAIR's provided functionality, how it is implemented, and how it is to be used in implemented agents.

Section 5.5 has introduced some example modalities and provides details on how a modality is defined and used in the architecture. This chapter expands on that information and presents implementation details for key facilities. Chapter 7 presents full examples of implemented agents and the details of their implementations.

6.1.1 Across Layers

The MGLAIR Knowledge Layer is made up of SNePS and its subsystems, the implementations of which I do not describe in detail here, except for those parts that are specific to MGLAIR and its support for modalities. The definitions of the PMLa and PMLs for a given agent are mostly agent-specific, dealing with primitive actions and perceptual capabilities that interface directly with the agent's KL and are specific to each agent and modality. These functions are implemented in Lisp by the agent architect, and operate in

the same running instance of Common Lisp as the KL.

Many of the structures and mechanisms used by MGLAIR modalities as parts of implemented agents are located within the PMLb, which interacts with the PMLa and PMLs running in the same Lisp image, and with the PMLc through the modality's data channel. This implementation includes definitions of modality classes, all of the functions and processes within the PML for managing modalities' data channels, buffers, focus, etc (described in Chapter 4), as well as the connections upward to the PMLa and PMLs — and hence to the sensing and acting systems in the KL — and to the PMLc. The implementation of an agent's Sensory-Actuator Layer depends on its embodiment. The PMLc connects to the PMLb through the modality data channels, and is implemented separately from the PMLa/s and PMLb.

Below I present details of implemented PMLc layers in Java and in Python. Both of these include classes and interfaces for handling modalities at the PMLc that are extended by the agent architect to implement the PMLc layers of individual agents. These general PMLc facilities are provided as part of the implementation of MGLAIR. An agent designer creating a new agent with an SAL that is accessible from either Python or Java should use one of these two APIs to develop the PMLc.

Below, examples of specific uses (e.g. uses of `define-modality`, or definitions of PMLs perceptual functions) are taken from agents described in full in Chapter 7. A full listing of the MGLAIR implementation appears in Appendix A.

6.1.2 Typical Use

In typical use of the MGLAIR software implementation, the agent architect loads and uses the facilities discussed in this chapter without modification. Nothing prevents the agent-architect from modifying the implementation of MGLAIR, e.g. by redefining built-in functions, but changes that alter the architecture itself are not part of typical agent development. Except where otherwise specified in this chapter, the functions, structures, and code presented are all parts of MGLAIR. Currently, this means that they are contained in a single directory called `mglair` that contains no agent-specific code. Using MGLAIR to create an agent involves placing this directory where the agent is being defined, and loading its PML files from the agent-specific PMLa and PMLb.

Modalities are defined using `define-modality` (§6.2.2), or by loading in modality definitions from a persistent file (§6.8). In each of the example agents discussed in detail in Chapter 7, the agent's modalities are defined in Lisp file called `modality-defns.cl`, which is loaded by the agent's implementation in `pmlb.cl`.

Using Efferent Modalities

SNeRE provides the function `attach-primaction`, which associates an action node form with a PMLa function that implements that action. If an agent is to turn, for instance, (using its `locomotive` modality) by performing acts that the agent represents using a term like `turn(left)`, then the agent architect uses `define-primaction` to define a PMLa primitive action function (called, e.g. `turn-act`) that implements turning with the use of that modality. The association between KL and PMLa actions is established by calling `attach-primaction`.

The (PMLa) primitive action must call a PMLb function that implements the action, passing as arguments the modality in which the action takes place, as well as a representation of the act impulse that the PMLb can handle. This PMLb function must be defined by the agent architect. It is the highest layer that should know about the PMLc and SAL. The PMLb function produces a representation of the act impulse that will be accessible to the PMLc (once it has passed through the modality's buffer and data channel). The PMLb function passes the impulse along the modality using the built-in function `PMLb:execute` (see §6.6.3) The behavior of the modality once `PMLb:execute` is called depends in part on the modality's properties: its data channel, buffer, and conflict handling specifications. If, for instance, the modality is currently being used but has a buffer with space in it, and is configured to queue up conflicting act impulses, the impulse is added to its buffer. As a result of calling `PMLb:execute` the impulse is either added to the buffer and then removed in turn and added to the modality's data channel, or it is sent directly to the data channel. Unless is it blocked or otherwise discarded, it eventually makes its way into the data channel, where it is sent to the PMLc. The PMLc runs separate threads that watch each modality's data channel, removing and processing impulses in order as they arrive. The agent architect's job at the PMLc is to make sure that the `handleImpulse` method for each modality is implemented, can understand the impulse emerging from the data channel, and activates the appropriate functionality at the SAL. As discussed more in §6.7.1, both the Java and Python PMLc implementations handle reading in modality configurations from file, as well as initiating, maintaining, and monitoring connections to the modality data channels. In order use these capabilities, the agent architect needs only to ensure that the PMLc has access to a file that describes the modalities (see §6.8).

Using Afferent Modalities

At the PMLc, each afferent modality is responsible for translating the output of one of the agent's sensors into messages that can be digested by the PMLb, and for sending them into the modality's data channel.

The agent architect must write code for this translation that uses knowledge of the SAL and of what the modality's PMLb function accepts as input. [1]

Similarly to the case with efferent modalities the PMLc API provides methods for connecting to, and sending messages with, modality data channels (see §6.7.1).

The Lisp PMLb operates a process running a data channel reader for each afferent modality. This process is responsible for retrieving arriving sense impulses from the data channel and adding them to the modality's buffer (see §6.6.4). The buffer's operation is completely automated from the agent architect's point of view: after specifying at the time the modality is defined its buffer's properties and the conflict-handling method, the architect does not worry about how these are implemented.

A separate process running in each afferent modality impulses out of the buffer and passes them to the modality's PMLb `sense-handler`. This is a function, one per modality, that the agent architect writes and associates with the modality (see §6.6.7). When applied to a sensory structure as it emerges from the afferent modality buffer, the sense-handler's job is to process that data (e.g., stripping from it the timestamp that was added when it went into the buffer), performing any required transformations before passing it along to the appropriate PMLs perceptual function in a format that function can understand.

Each PMLs perceptual function converts its input into a KL term and adds that term to the knowledge base. These are written by the agent architect, one per modality, and must handle input from the PMLb sense handler.

PMLs perceptual functions take the final step in the agent's becoming aware of (perceiving) the percept. PMLs perceptual functions add terms to be perceived using the included PMLs function `believe`, which performs the mental act of believing the term (and using automated belief revision to resolve any conflict), and then returns the KL term that resulted from this.

Each PMLs perceptual function should evaluate to the KL term that results from a successful perception, as this is used to add the resulting term to the modality register (6.6.8).

6.1.3 Typical Use Summary

When creating an MGLAIR agent, the agent architect is responsible for:

- Designing and implementing the KL.
- Defining modalities using `define-modality`.

[1] One possibility for future improvements in the implementation here is to allow the PMLb and PMLc to share information with each other about the formatting and permissible values of the act impulses and sensory data they exchange.

- Defining primitive actions in the PMLa for acts in afferent modalities using `define-primaction` and attaching them to the action nodes at the KL using `attach-primaction`.

- Defining PMLb functions to be called by those PMLa primitive actions, and that themselves call `PMLb:execute`.

- Defining PMLc methods to handle incoming act impulses and translate them into commands at the SAL.

- Defining PMLc methods for efferent modalities to take sensor output from the SAL and format it for use by the PMLb.

- Defining PMLb data handler functions that are applied to the contents of the afferent modality buffer as they are removed, and which call PMLs perceptual functions.

- Defining those PMLs functions, ensuring that they build correct SNePSLOG terms and assert them (using `PMLs:believe`).

The convention is to use names for PMLs perceptual functions that start with `perceive-`, and to define PMLb sense handler functions that have names ending with `-sense-handler`. PMLa primitive actions typically end with `-act`. The proper functioning of the architecture does not depend on these conventions, but following them makes it easier to keep track of what's happening across the layers.

6.2 Modality Specifications

Recall, as discussed in Chapter 3 that an MGLAIR modality specification is a tuple of modality properties, which determine its behavior: ⟨ `name, type, predicates, channel, buffer, access, focus, conflict, description, relations` ⟩:

6.2.1 Modality Classes

Each type of modality is represented by a modality class. Modality classes are implemented as CLOS (Common Lisp Object System) classes in MGLAIR. The implementation includes classes for `modality`, as well as subclasses `afferent-modality` and `efferent-modality` Every modality that is part of an instantiated agent is an instance of a modality class. Below is the definition for the `modality` class, which all modality classes should be subtypes of. `modality` should not be directly instantiated or directly subclassed.

Listing 1 `modality` class definition

```
(defclass modality ()
  (
   ;; the modality's name -- used to uniquely identify it
   (name :initform 'Modality :initarg :name :accessor name)
   ;; predicate/frame associated with the modality
   (frame  :accessor frame :initarg :frame :initform 'Sensed)
   ;; can the agent consciously access the modality? (t or nil)
   (access :accessor access :initarg :access :initform nil)
   ;; stores this modality's data channel process
   (dc-process :accessor dc-process)
   ;; modality percept/act buffer
   (buffer :initform
   (make-instance 'modality-buffer) :initarg :buffer :accessor buffer)
   ;; focus level - initially normal (0)
   (focus :initform 0 :accessor focus)
   ;; is focus modifiable? (t or nil)
   (focus-modifiable :initform nil :accessor focus-modifiable)
   ;; sleep-inc is used internally to manipulate focus
   ;; Not intended to be modified by the user.
   (sleep-inc :initform .1 :accessor sleep-inc)
   ;; If nil delays the running of the  modality
   (runmod :initarg t :accessor runmod)
   ;; Modality register stores KL term for last act/percept
   ;; Modified by execute-primaction, and by perceptual functions
   (register :initform nil :accessor register)
   ;; Human-readable description of the modality
   (description :initarg :description :initform "A modality" :accessor description)
   ;; Modality data channel
   (channel :initarg :channel :initform
    (make-instance 'modality-data-channel ) :accessor channel)))
```

In addition to slots for each of the modality properties discussed earlier in Chapter 3 and others, `modality` class has some slots that are used within the system by various modality mechanisms. These include:

- `sleep-inc`, which is used in the implementation of modality focus, discussed more below in §6.6.9.
- `register`, which implements the modality register, storing KL terms for the most recent act or percept for which the modality has been used, as discussed more in §6.6.8
- `dc-process`, which stores the process that runs the modality's data channel management functions discussed in §6.6.4.

Direct Subclasses of `Modality`

MGLAIR includes definitions for two direct subclasses of `modality`: `afferent-modality` and `efferent-modality`.

The definition for the `afferent-modality` class adds a slot for the `sense-handler` function, applied to the contents of the modality's perceptual buffer. The initial value for this is a very simple function that attempts to assert its argument at the KL. The agent architect is expected to redefine this function. The default is provided as a place-holder, but is unlikely to be useful, even for quite simple instances of afferent modalities. The `afferent-modality` class also includes a slot called `pproc` that stores a reference to the modality's perceptual process, which runs the function that removes items from the modality's buffer and applies its sense handler to them, which eventually results in the PMLs perceptual function being called and in the agent's perceiving. It also overrides the `buffer` slot to specify an `afferent-buffer`.

Listing 2 Definition of `afferent-modality`

```
(defclass afferent-modality (modality)
  ((sense-handler :accessor sense-handler :initform (lambda (x) (snepslog::tell x)))
   (pproc :accessor pproc)
   (buffer :initform
           (make-instance 'afferent-buffer) :initarg :buffer :accessor buffer)))
```

The `efferent-modality` class definition is also a relatively simple extension of `modality`, changing the default frame to roughly match the direction of the modality. The `bproc` slot stores a reference to the process that reads impulses out of the modality's action buffer and adds them to its data channel. The initial form for `buffer` is an instance of `efferent-buffer`.

Efferent modalities do not have the equivalent of the afferent modality's data handler and perceptual function in the PML. Act impulses in an efferent modality are added to the modality's buffer by the PMLb

`execute` function. A separate buffer management process for the modality handles moving impulses through its buffer and adding unexpired impulses to its data channel in turn.

Listing 3 Definition of `efferent-modality`

```
(defclass efferent-modality (modality)
  ((frame  :accessor frame :initarg :frame :initform 'Move)
   (bproc :accessor bproc)
   (buffer :initform
   (make-instance 'efferent-buffer :initarg :buffer :accessor buffer)))
```

Modality Subclasses vs Instances

Though every modality is an instance of a modality class, and every modality class is a subclass of either `afferent-modality` or `efferent-modality`, it is not necessarily the case that every modality is an instance of a *proper* subclass of `afferent-modality` or `efferent-modality`: sometimes it will make sense to create a modality that is a direct instance of one of these two, and sometimes it will make more sense to subclass and then instantiate. The choice about when to do this is up to the agent designer. As a rule of thumb, it is most useful to create subclasses for a new type of modality when there may be multiple instances of that modality within the same agent. For instance, an agent whose embodiment includes vision realized through two independently-operated camera systems should attach a separate afferent modality to each. If the processing performed by both is similar or identical (i.e. if they use the same perceptual functions), then it makes sense to create a `visual-modality` subclass and then create instances of it for `vision1` and `vision2`.

For instance, below is an example definition for an `afferent-modality` subclass called `visual-modality`, in which a very simple default data handler is provided.

Listing 4 Example of defining a class for `visual-modality`

```
(defclass visual-modality (afferent-modality)
  ((data-handler :accessor data-handler
 :initform (lambda (s) (PMLs:see s)))))
```

If the agent uses many identical sensors (e.g. an array of ultrasonic range sensors surrounding an agent's robotic enclosure), then there is an even stronger case for defining a class that includes elements these sensors have in common rather than redefining the same elements repeatedly. For instance, it might be desirable

for all of an agent's many ultrasonic range sensors to have a high focus and a buffer with a small size that handles conflicts by squeezing out the oldest unperceived sensory impulses. In this case it would make sense to set defaults for these values at the level of the class that the modalities instantiates.l

Generally when an agent is designed to have only one instance of a particular modality, there is no reason to create a new subclass for that modality instead of creating and modifying an instance of the existing modality classes. The following section presents the implementation of modality definitions.

6.2.2 Defining Modalities

Modalities are defined using the `define-modality` function, which takes arguments for each modality property, and creates an instance of the appropriate modality class. Modality definitions may also be written to — or read in from — files that persistently store the definitions, as discussed more in §6.8.

`define-modality` adds the newly-defined modality to the `*modalitiesHash*`, where all modalities objects are stored in the PML.

The full definition for `define-modality` is too long to fit on a single page here, but appears in Appendix A. `define-modality` creates an instance of the specified modality type and fills in the modality's slots that correspond to its properties based on their values specified as arguments to `define-modality`.

Listing 5 `define-modality`'s lambda list

```
(defun define-modality  (name type &key predicates access focus conflict description
                           channel buffer  sense-handler delay-running print) ....)
```

Most of the arguments to `define-modality` are optional because they have default values. Exceptions are `type` and `name`, which are required. `name` uniquely identifies the modality and is used internally in functions that look up modalities, etc. `type` must be a subtype of `afferent-modality` or `efferent-modality`. `define-modality` checks all of its arguments to make sure they are non-`nil` in order to avoid overwriting modality class default values.

The translation from `define-modality` arguments to corresponding `modality` class slots is straightforward (i.e. `name` becomes `name`, and so on).

`type` should be a symbol that names a subclass of `afferent-modality` or `efferent-modality`.

`frames` should be a list of symbols each of whose symbol-names corresponds to a frame used at the knowledge layer. A list of the corresponding symbols in the `snepslog` package are stored in the modality object's `frame` slot.

`access` denies the agent conscious access to this modality unless it is non-`nil`.

The `focus` specification is a dotted pair (`focus-modifiable . initial-focus`). If these are not specified in any modality definition, they default to `nil` and `0` (`normal`), respectively.

`description` should be a string containing a human-readable description of the modality.

The data channel specification, `channel`, is an association list - a list of pairs - in which the keywords `type`, `port`, and `host` each appear as the first element of a pair, and their values appear as the second element of the matching pair. For instance, `((type . socket) (port . 8080) (host . localhost))` specifies a data channel that allows the PMLc to connect to the PMLb via TCP/IP sockets on port 8080 on the local machine.

Note that `socket` is the only type of data channel connected currently supported in the MGLAIR implementation, and hence is the default type. The implementation leaves open the possibility of adding other data channel types in the future. Recall that the modality data channel is used to connect the PMLb to the PMLc, and that these two layers may be, and often are, realized on separate physical computers.

The `buffer` specification is an association list using keywords `capacity` and `exp-int`, which specify the buffer's capacity and expiration interval, after which elements in the buffer will be discarded without being perceived at the KL (for afferent) or executed at the SAL (efferent). The defaults for both subtypes of `modality` are discussed below in 6.6.5.

`sense-hander` should be a symbol that has the same `symbol-name` as the sense handler function defined for the modality in the agent's PMLb.

If `delay-running` is non-nil, the modality will not actually be run (i.e. the processes for the modality's internal mechanisms will not be started) until the `run-modality` function is evaluated with that modality as its argument. Otherwise, it will be run by `define-modality` as soon as its properties have been set.

`print` does not affect the modality's functioning. If non-nil, it will cause the `print-modality` function to be called on the modality at the end of `define-modality`, which will cause information about the modality's properties to be displayed.

6.2.3 Storing and Retrieving Modalities

Modalities Hash

All defined modalities are stored internally in the hash table `*modalitiesHash*`. Each is added to the table at the time it is defined by `define-modality`. `*modalitiesHash*` keys are the modalities' names (symbols) and the values are the modality objects themselves. Normally the agent-architect will not need to

interact directly with the `*modalitiesHash*`, but uses predefined functions in the PMLb package to access the modality objects.

Modality Functions

MGLAIR includes a number of functions that operate on modalities and simplify using and manipulating them. Some of the most useful of these include the following.

The `run-modality` function takes a modality and starts the processes that manage that modality. For an afferent modality these are the data channel reader, and the perceptual process that contains buffer management and calls the the modality's PMLb function and PMLs perceptual function. Running an efferent modality starts the data channel process and the process that monitors the modality buffer, adding impulses to the data channel.

The `get-modality` function takes a symbol whose `symbol-name` is the name of a modality, and returns that modality object if it exists, and `nil` otherwise.

The function `get-modalities-by-type` takes as its argument a type and returns a list of all modalities that instantiate that type. For example, `(get-modalities-by-type 'afferent-modality)` gets the list of all afferent modalities.

The full listing of functions that operate on modalities is available in Appendix A. Many of these are not intended for use in the normal course of agent development, or as parts of agents, but are useful when dealing with the internal workings of modalities.

6.3 Knowledge Layer

MGLAIR's KL is comprised of SNePS and its subsystems (e.g. the SNeRE acting system). The KL connects to the PMLa by attaching to KL primitive actions PMLa functions that implement them.

For instance, an action common to many implemented agents that include locomotive capabilties is `turn`. At the KL, this action may be represented by a frame definition (e.g. `define-frame turn(action d)`) that is associated with a PMLa function (e.g., `turn-act`) by use of `attach-primaction`, as follows: `(attach-primaction turn turn-act)`.

The result is that whenever the agent consciously performs the action `turn` with some argument indicating the direction (e.g. `perform turn(left)`, the PMLa function `turn-act` is called with a SNePS term for `left` as its argument, and a term describing the action (`turn(left)`) is added to the modality register for the relevant modality. The PMLa is discussed more below in §6.4.

The PMLs connects upward to the KL through perceptual functions that take as their input sensory structures from the PMLb, process those structures, build KL terms representing the corresponding percepts, and assert them using a mental act such as:

```
perform believe(Facing(wall))
```

See §6.5 below for more on the PMLs.

6.4 PMLa

Unlike the PMLb, where most of the built-in modality mechanisms are implemented, and the KL, where all of the reasoning, sensing, and acting capabilities that are included with SNePS lie, the PMLa contains relatively little that is agent-independent. That is, most of the contents of an MGLAIR agent's PMLa are agent-specific, consisting mainly of primitive action definitions that reflect the agent's capabilties. The included agent-independent capabilities implemented at the PMLa include primitive actions that operate on modalities to adjust their focus.

PMLa Primitive Actions

For example, the following is a definition for a PMLa primitive action for an agent with a grasping effector. It calls a PMLb function `grab` that, as discussed below, takes an argument for the modality in which the action takes place, and one for the direction of the motion. Here, that second argument is `t.` which corresponds to closing the grasping effector.

Listing 6 Definition of PMLa `grab-act` for an agent with a grasping-effector

```
(define-primaction grab-act ()
 (PMLb:grab 'grabber t))
```

This primitive action is attached at the KL to the term `grab`, which allows the agent to consciously perform actions that move its effector.

Listing 7 Attaching the primitive action `grab-act` to the KL term `grab`

```
(attach-primaction grab grab-act)
```

Focus Actions

The MGLAIR PMLa includes agent-independent primitive actions for adjusting modality focus, to be used by agents that are granted the ability to do so. Such agents must be granted conscious access to the relevant modalities. In in order to adjust modality focus in the course of reasoning and acting, such agents must be designed with policies that will trigger focus-adjusting actions. The primitive action for `decrease-modality-focus`, which operates on a modality using the PMLb function `adjust-focus`, appears below.

Listing 8 Definition of `decrease-modality-focus` function

```
(define-primaction decrease-modality-focus ((dmfmod))
  (PMLb::adjust-focus (sneps:node-na dmfmod) :decrease? t))
```

These actions and the frames for them are automatically defined and attached when MGLAIR is loaded. For example, the `unfocus` definition is: `define-frame unfocus(action dmfmod).`

When an `unfocus` action is performed, e.g. `unfocus(locomotion`, the `dmfmod` slot is filled by a term that names the modality. Performing the action results in the attached primitive action (`decrease-modality-focus`) being called with the modality's name as its argument. MGLAIR's PMLa includes attached primitive actions for the other focus acts described in §3.8: `focus`, `ignore`, `attend`, and `restore-focus`.

6.5 PMLs

Like the PMLa, the PMLs contains very little that is agent-independent. The bulk of its contents for any agent consists of agent-specific and modality-specific perceptual functions that are applied to the sensory structures emerging from the PMLb, resulting in perception at the KL.

PMLs:believe

PMLs perceptual functions (modality-specific and written by the agent architect) should evaluate to the node that resulted from their believing it. This is used to make sure that each modality's modality register stores the term for the thing last perceived in that modality. A PMLs perceptual function can do this by building a string that looks like a KL term and passing that string to the general function `PMLs:believe`, which is included as part of MGLAIR. `PMLs::believe` uses the tell/ask interface to SNePS to perform the mental act of believing the term, which does not itself return a result, and then immediately asks whether

it that term is believed:

Listing 9 Definition of PMLs:believe

```
(defmacro believe (termstr &rest args)
  `(let ((term (funcall #'format nil ,termstr ,@args)))
     (snepslog:tell "perform believe(~A)." term)
     (snepslog:ask term)))
```

6.6 PMLb

At the PMLb, the agent architect defines two kinds of functions. Sense-handling functions are automatically applied to sensory data as they emerge from the modality's buffer. Action functions are called by the PMLa and implement the modality's actions.

6.6.1 PMLb Sense Handler Functions

The agent architect writes PMLb sense-handling functions that accept as input time-stamped structures and that are responsible for passing input to the PMLs perceptual function for that modality (in the format it is expecting). The exact format of the input to this PMLb function depends on what the PMLc is adding to the modality's data channel. For the example range-sensor sense-handler defined below, this simple task consists of stripping away the timestamp, which the PMLs function has no use for, `read`ing the result, and passing it to the PMLs perceptual function. An example sense datum that this function might handle is:

`(61378511662 . ''(d . 88)''))`. Note that `range-sense-handler` will evaluate to the result of evaluating the call to `perceive-distance`. This is fact is used in the maintenance of the modality's register.

Listing 10 Example definition of an NXT range sensor modality's sense-handler

```
(defun range-sense-handler (v)
  (PMLs:perceive-distance (read-from-string (rest v))))
```

The structures coming out of the modality buffer are always dotted pairs, the first element of which is a timestamp representing the time (in milliseconds) that the data entered the buffer. The second element of this structure is a string read from the modality's data channel. Its contents are determined by the implementation of the PMLc for the modality.

A useful strategy when dealing with PMLbs and PMLcs that are implemented on different systems (e.g. Common Lisp and Python), is to have the PMLb send into the data channel information that will be easily digested by the PMLc (e.g. a string that can be evaluated as some structure at the PMLc), and to have the PMLc return the favor (sending strings that can be evaluated by the PMLb). One example of this is the NXT agent discussed in detail in Chapter 7.

6.6.2 PMLb Action Functions

The agent architect writes PMLb action functions that are called by the PMLa primitive actions, each of which passes down a symbol naming the modality that the action is in.

As an example consider the function definition below from the NXT agent, which builds a string that the (Python-based) PMLc can easily process (the function `make-tuple-str` takes a list and builds a string that will evaluate to a Python tuple), and passes it along to be `PMLb:execute`d in the modality for the grab action. The impulse produced here is the string "(g, False)" when the `close?` parameter is nil, and "(g, True)" otherwise.

Listing 11 Definition of PMLb function `grab`, which builds a representation of the action accessible to the Python PMLb, and `PMLb::execute`s it

```
(defun grab (mod &optional close?)
  (execute mod (make-tuple-str `(g ,(if close? "True" "False")))))
```

6.6.3 PMLb:execute

Every efferent PMLb function uses `PMLb:execute` to add to the modality's buffer and data channel impulses for the actions it implements. When the modality has an action buffer with available space, `execute` adds the impulse to the buffer. If the buffer has no available space and is set to block incoming impulses in this case, or if the modality has been configured to block new impulses while it is carrying out an action, then the impulse is discarded.

Blocking Act Impulses

How does a modality know when it is currently *carrying out an action*? Knowledge of this is necessarily dependent on the agent's embodiment. For agents that require the use of this information, the PMLc is obligated to monitor its own use of the relevant effector and send a control message back up through the data

channel when the effector is no longer in use. When this option is used, the impulse for the action currently being carried out by the PMLc/SAL is stored within the PMLb and a process is spawned to watch for the confirmation that the action has completed. Each such act impulse is uniquely identified by the timestamp (in milliseconds) that was added to it at the time it was put into the modality's buffer. The PMLc process responsible for confirming that the action has completed is expected to do this by sending the timestamp for the act up through the data channel's socket.

The ability to block new acts in a modality is key, particularly when the agent also lacks a way of consciously sensing its own actions. Consider an agent that has a policy for approaching an object of interest, e.g., *whenever the object is in front of me and I am not near the object, go forward.* When the agent consciously performs *going forward*, the primitive action function that implements it calls the corresponding PMLb function, which calls `execute`, and the impulse goes into the modality's buffer and data channel, and is eventually realized in the motors actually turning the wheels and moving the robot forward. The PMLa primitive action finishes its evaluation after the impulse has been added to the modality's buffer. This will usually occur a long time before the wheels have finished rotating in order to move the agent's body forward. Consequently, while it is still moving forward, the agent will perceive that it is not yet near the object, and the action for this policy (*go forward*) will be selected and executed again. This may happen several times while the original action is still being carried out. The result, if the modality is simply queuing act impulses and executing them in order, will be a disaster: the agent will have in its buffer many *go forward* impulses, and these will continue to be executed long after it's reached (and passed) the object of interest (or smashed into a wall, fallen off a cliff, etc.) This undesirable behavior can be mitigated somewhat by giving the `locomotive` modality in question a small buffer and setting it to block impulses when the buffer is full, but this has other drawbacks (including possibly frequent lost act impulses).

An agent like the example discussed above but with the ability to sense via proprioception whether its wheels are in motion, as is the case with the NXT agent introduced in the previous chapter and discussed in detail in Chapter 7, could avoid this issue by using policies that take into account the state of its body, e.g. *whenever the object is in front of me and I am not near the object* **and I am not moving**, *go forward.*

In some cases it may not be feasible to have the PMLs monitor the effector if, e.g., there is no sensor available to provide feedback. This makes it difficult to block new act impulses *while the old one is being carried out*, as the system has no way of knowing when this is no longer the case. If, in such a case, we still want the modality to block act impulses when it is in use, there are two options: the PMLc may be designed to wait an appropriate amount of time and then send confirmation that the action has completed,

or the PMLb may be set to clear the flag indicating that the modality is in use after some period of time. In either case, the agent architect should select a time appropriate to actions for the modality in question. If going forward by a single unit in the locomotive modality usually takes around 300 milliseconds, then that would be an appropriate value for this timeout. Even in case the effector supports monitoring its progress with an appropriate sensor, and the PMLc is configured to report act completion, it is a good idea to have a default timeout in the PMLb process that watches for this confirmation so that the modality is not locked permanently if there is a miscommunication with the PMLc. Appropriate values for this default timeout vary, but should be at least as long as it takes for actions in the modality to finish.

6.6.4 Modality Data Channels

The PMLb and PMLc each run separate processes for each modality, often implemented in different programming languages, and sometimes even executed on different physical machines. Modality data channels are used to pass act impulses and sensory data between the PMLb and the PMLc. The implementation under discussion here uses TCP/IP sockets to make the data channel connections. These channels could be implemented in a number of ways, depending on the system(s) used. These options include simple streams within the same Lisp image, Unix Interprocess Communication between processes on the same physical machine, or the use of other networking protocols, e.g. UDP.

It is the PMLc's responsibility to initiate the connection to the PMLb, which listens on a designated port for each modality. The port and hostname for each modality is specified as part of the modality definition. This information is stored in the XML file that is produced when a modality definition is exported (see §6.8).

The PMLc takes these files as input and uses them to set up the connection. The PMLc side of the modality data channel is discussed more in §6.7 below. Briefly: the MGLAIR implementation provides facilities in Java and Python that read in modality definition files and build objects that represent the modality's data channels, which are then passed to methods that connect to the channels.

The PMLb end of modality data channels is implemented using Allegro Common Lisp's socket library. The CLOS class for modality data channels is defined as:

Listing 12 `modality-data-channel` class definition

```
(defclass modality-data-channel ()
  ((socket :initform nil :initarg :socket :accessor socket)
  (channel-stream :initform nil :initarg :channel-stream :accessor channel-stream)))
```

Each modality has its `data-channel` slot filled by an instance of `modality-data-channel`. The `attach-channel` method takes as its arguments a modality and a modality data channel and attaches the channel to the modality – unless the modality already has a connected channel.

In afferent modalities the data flows up to the PMLb from the PMLc, the PMLb handles taking items out of the data channel and adding them to the perceptual buffer. A separate process handles this for each afferent modality, running that modality's data channel reader. Each data channel reader process is named ⟨*modality-name*⟩*-channel-reader*.

The default data channel reader for each modality is established using the `run-dc-reader` function, defined below. When given an afferent modality as an argument, `run-dc-reader` runs a function in a new process that first listens for and accepts a connection on the modality's data channel (`socket::accept-connection` blocks until a connection is initiated by the PMLc). The function then enters a loop that reads one line (string) at a time from the socket, trims whitespace, and adds it to the modality's buffer.

Listing 13 Definition for `run-dc-reader` function, run on every afferent modality's data channel

```
(defmethod run-dc-reader ((amod modality))
  (setf (dc-process amod)
(mp:process-run-function
 (concatenate 'string (symbol-name (name amod) ) "-channel-reader")
 (lambda ()
   (let ((sock (socket::accept-connection (socket (channel amod)))))
     (setf (socket (channel amod)) sock)
     (cl-user::while t
     (funcall #'add-to-buffer (buffer amod)
      (string-trim '(#\newline #\return #\null)
```

6.6.5 Modality Buffers

Each modality, whether afferent or efferent, has a modality buffer as part of it. In both types of modalities, the buffer is stored in the `buffer` slot in the CLOS modality object.

In afferent modalities, the buffer queues up sensory structures that have arrived at the PMLb via the modality's data channel. These are added to the buffer by the modality's data channel reader process (§6.6.4) using the `add-to-buffer` function, which takes as arguments a modality buffer and an item to be added.

When an unexpired structure reaches the front of an afferent modality's buffer, the PMLb sense-handler function for that modality is applied to it, as discussed below in §6.6.7.

In efferent modalities, the modality buffer stores act impulses that have not yet been sent to the data

channel. These are added by the `PMLb:execute` function using `add-to-buffer` in accordance with the modality's conflict handling and buffer properties.

Normally, the agent architect does not use `add-to-buffer` directly, though the choice of buffer properties in each modality's definition will influence its behavior when it is called (either by `execute` or modality's data channel reader).

6.6.6 Buffer CLOS Classes

The `modality-buffer` class is defined as follows:

Listing 14 Definition of `modality-buffer` class

```
(defclass modality-buffer ()
  ((queue :initform (make-instance 'mp:queue) :accessor queue)
   (exp-int :initform -1 :initarg :exp-int :accessor exp-int)
   (conflict :initform 'squeeze :initarg :conflict :accessor conflict)
   (capacity :initform -1 :initarg :capacity :accessor capacity)))
```

Modality buffers use the Allegro Common Lisp's multiprocessing `queue` data structure, which provides atomic enqueue and dequeue operations.

The buffer's expiration interval defaults to `-1` which, in this implementation, means that data in the buffer do not expire. The default conflict-handling for full buffers is `squeezing`, which means that the oldest item in the buffer is discarded to make room for a new item. The default buffer capacity is also unlimited.

Subclasses for `afferent-buffer` and `efferent-buffer` override some of these defaults. For instance, the `efferent-buffer` class establishes a small capacity for efferent buffers and sets the conflict handling to `blocking`, so that new impulses are rejected if the buffer is full.

6.6.7 Afferent Sense Handlers

A separate process for each afferent modality runs its data handler. The function `run-sense-handler` is typically called once for each modality, and is responsible for starting that modality's data handling process and storing the process in the modality object's `process` slot.

Each afferent modality's sense handling process runs the `sense-handler-fn` function, which operates an infinite loop that gets the next unexpired item out of the modality's buffer and pass it to the modality's sense handler function, stored in the afferent modality's `sense-handler` slot:

Listing 15 Definition of `sense-handler-fn`, which is run for every afferent modality

```
(defun sense-handler-fn (amod)
  ;;; runs a looping process that checks this modality's  buffer for incoming senses
  ;;; & manages expiration
  (cl-user::while t
   ;; grab the next structure from the modality buffer
   (let ((sensed (buffer-pop (buffer amod))))
     ;; get rid of data older than the deadline
     (unless  (or (null sensed)
  (expired? sensed (exp-int (buffer amod))))
       ;; whether it's expired depends on the exp-interval for this modality
       (setf (register amod) (funcall (sense-handler amod)  sensed)))
     (sleep (sleep-inc amod)))))
```

Listing 16 `run-sense-handler` creates the process that runs the data handler for each modality

```
(defmethod run-sense-handler ((amod afferent-modality))
  "Runs an afferent modality's sense handler"
  (setf (pproc amod)
        (mp:process-run-function
         (symbol-name (name amod)) (lambda () (funcall #'sense-handler-fn amod)))))
```

In a previous version of the implementation, `sense-handler-fn` was an anonymous function defined within `run-sense-handler`. Here it appears as a separate named function just for clarity's sake and for ease of formatting. `sense-handler-fn` is not called by any function other than `run-sense-hander` and is not intended to be called directly by any agent-specific code.

Note that `run-sense-handler` is called for each afferent modality in the course of starting that modality. Usually this is done at the end of `define-modality`, though it can be delayed and later started explicitly with a call to `run-modality` for each modality.

6.6.8 Modality Registers

Afferent

Afferent modality registers store the KL term for the last percept perceived by the agent. This is accomplished by having the process that runs the modality's sense handler loop add the value returned by the sense handler to the modality's register slot each time the sense handler is called. The PMLb sense handler must make sure to evaluate to the KL term, which is handled by having the corresponding PMLs function evaluate to the result of calling `PMLs:believe`, as discussed in §6.5.

Efferent

Each efferent modality's register stores the KL term for the last action the agent has performed. This is done in the `execute-primaction` function. `execute-primaction` is a built-in part of SNePS. It takes as its argument an action node, and uses the internal mapping between action nodes and primitive action functions to execute the action. It has been modified in MGLAIR to also look up the modality in which the action is taking place (based on the association between KL terms used in frames for the modality), and then store the node for that action in the modality's register. For instance, when an agent performs `turn(left)` using its `locomotive` modality, `execute-primaction` looks up the attached primitive action function in the PMLa, `turn-act`, and also uses the symbol `turn` to look up the efferent modality with which it's associated (i.e. `locomotion`). The KL term for the action node `turn(left)` is then stored in the modality register for that modality.

6.6.9 Focus

As discussed in §4.5, modality focus allows agents to prioritize perceptual processing in afferent modalities by adjusting the amount of time that the modality's internal processes are inactive between operations.

The modality focus level, which is a property of each modality ranges from `lowest`, ..., `highest`, which correspond internally to the numbers `-3`, ..., `3` respectively.

The internal base interval `*mod-sleep-base*`, used to determine how long modality processes should sleep, is .1 seconds. That is, for a focus level f, the interval for which the modality buffer process sleeps is $(0.1) \cdot 2^{-f}$ seconds. Note that lowering or raising a modality's focus halves or doubles this duration.

Listing 17 Definition of `focus-sleep`, which calculates the sleep time for the given focus level

```
(defmethod focus-sleep ((f number))
  "Given a focus level, calculate the amount of seconds to sleep"
  (* *mod-sleep-base* (expt 2 (* -1 f))))
```

`adjust-focus`, when applied to a modality, raises or lowers its focus and updates the modality's sleep interval, which changes the frequency with which the modality's data handler process runs.

The implementation of focus and concurrent processing of modalities uses Allegro Common Lisp's `multiprocessing` package. The minimum sleep time for a process is set to 10 milliseconds in MGLAIR. Code in Allegro Common Lisp that attempts to sleep for less than the minimum sleep time is unstable. `adjust-focus` makes sure that a modality's focus level is within the specified boundaries. If `adjust-focus`

Listing 18 Definition of `adjust-focus` function, which increases or decreases the given modality's focus

```
(defmethod adjust-focus ((amod afferent-modality) &key decrease?)
  "Increase or decrease the modality amod's focus unless it's either
  already highest or lowest. Warns the user and returns the focus or
  nil if adjusting was not successful."
  (if (and (not decrease?) (= (focus amod) focus-highest))
      (warn "Can't increase focus any more")
      (if (and decrease? (= (focus amod) focus-lowest))
          (warn "Can't decrease focus any more")
          (let ((newfocus (funcall (if decrease? #'1+ #'1-) (focus amod))))
              (setf (focus amod) newfocus)
              (setf (sleep-inc amod) (focus-sleep newfocus))
              (focus amod)))))
```

is called to lower focus on a modality whose focus is already `lowest`, or raise it on a modality whose focus is already `highest`, the modality's focus remains at the level it previously had.

`ignore-modality` replaces the modality's data sensing process with a new process whose function just reads messages from the data channel and disregards them.

Listing 19 Definition of the method `ignore-modality`, for ignoring sensations in an afferent modality

```
(defmethod ignore-modality ((amod afferent-modality))
   (set-modality-process amod
 (lambda () (cl-user::while t (read-line (socket (channel amod)))))))
```

`unignore-modality` undoes this and restores the modality's data sensing process with its original function.

6.7 PMLc

Each PMLc implementation is necessarily dependent on the implementation of the agent's body. The PMLc is responsible for connecting to the modality data channels above it, and for translating between the PMLb and the SAL.

In an afferent modality, the PMLc monitors the modality's data channel for act impulses, which the agent-specific part of the PMLc is responsible for converting into commands that the SAL accepts.

In an efferent modality, the PMLc is responsible for responding to sensor data (often by running processes or threads that poll the SAL's sensor), constructing representations of it that the PMLb can handle, and sending them into the modality's data channel.

6.7.1 Connecting and Managing Modality Data Connections

The MGLAIR implementation includes classes and methods for connecting Java-based and Python-based PMLc implementations to data channels. The data channel parameters are provided to the PMLc by giving it access to a persistent description of the modality, as discussed more below in 6.8.

This section discusses the Java implementation, the implementation details of which are presented in full in Appendix A. Full examples of its use are presented in Chapter 7.

Also presented in Appendix A is the Python version, which has the same functionality as the Java version, but is not discussed in any detail here.

`ModalityManager`

The `ModalityManager` class is responsible for connecting all the modality data channels for a single agent and passing messages up through afferent data channels from the agent-specific PMLc, and down to the PMLc from efferent data channels. The `ModalityManager` constructor takes an instance of a class that implements the `PMLcI` interface (see §6.7.2), as well as a String that names an agent configuration file containing modality definitions. The `ModalityManager` creates a `ModalityConnector` for each modality defined in the file, and passes to each `ModalityConnector` the `ModalityConfig` object for the modality whose data channel it is to connect. `ModalityConfig` objects are produced by the `AgentConfigReader` class, which is not described in detail here. It is responsible mainly for reading the XML file containing the agent's modality definitions and producing the corresponding `ModalityConfig` objects, which are used by the `ModalityConnector` and others. The `ModalityManager` maintains a `HashMap<String, ModalityConnector>` called `connectors` whose keys are the names of individual modalities and values are the `ModalityConnector` objects.

`ModalityConnector`

The `ModalityConnector` class establishes the connection to a single modality's data channel. Each runs in a separate thread and contains a loop that constantly watches the data channel for incoming messages. The `ModalityConnector` initially receives information about the modality to whose data channel it is to connect in the form of a `ModalityConfig` instance, as discussed above. It also takes as an argument to its constructor an instance of a class that implements the `PMLcI` interface. This is used by the `ModalityConnector` to send messages down to the agent-specific PMLc when they arrive on the modality's data channel.

6.7.2 PMLc Interface

Each Java MGLAIR agent's PMLc must contain a class that implements the `PMLcI` interface, shown below. The `handleImpulse` method of the agent's PMLc takes the name of a modality and a String representation of an act impulse, and implements the corresponding act in that modality by interacting with the SAL. Each efferent modality's `ModalityConnector` uses its reference to the agent's PMLc to pass incoming act impulses on to the PMLc. Similarly, the `handleSense` method takes a String naming the afferent modality in which the sense takes place and a String representation of a piece of sensory data, and sends that data to the corresponding modality's `ModalityConnector`, which places it into the modality's data channel for consumption by the PMLb.

Listing 20 The PMLcI interface, implemented by each Java MGLAIR agent's specific PMLc

```
interface PMLcI{
        public abstract void handleImpulse(String mname, String i);
        public abstract void handleSense(String mname, String s);
}
```

The agent's PMLc is responsible for initializing the `ModalityManager` and its connection to the modalities. For instance, the following is a simplified version (without error handling) of the constructor for the Java/Greenfoot Delivery Agent described in §7.3:

Listing 21 Java/Greenfoot Delivery Agent's PMLc class constructor

```
public PMLc(String configFN, DeliveryAgent b){
    // initialize a ModalityManager for agent
    modMgr = new ModalityManager(configFN,this);
    // connect the names modalities
    modMgr.connectModalities("locomotion","grabber","vision", "visual-efferent", "range");
    this.bot = b;
}
```

This constructor takes as its arguments the filename for the modality definitions file, and a reference to the `DeliveryAgent` object that implements the simulated robot's SAL.

The PMLc's `handleImpulse` method takes the name of a modality and a string representing an act impulse. For instance, an act impulse corresponding to the action `turn(left)` looks like `(983274341 . (tn . -90>))`, which the `handleImpulse` method translates into, e.g. `bot.turn(-90)` where `bot` is a `DeliveryAgent` instance, and the class includes a method `turn` that causes its body to turn appropriately.

Similarly, the PMLc methods that monitor the body's sensors call `handleSense` to pass sensory data up through the layers, e.g. `handleSense(''vision'', ''(room . 5))''` when the agent is standing in front of room number 5 and has just performed the `readRoomNumber` act.

6.8 Persisting Modalities

As mentioned in §3.3, MGLAIR allows modality definitions to be persistently stored in a file. Such modality definition files can then be read back in to recreate the modalities, though there is no real advantage to doing so if the agent already includes `define-modality` forms for each. However, this does allow the agent designer to forgo specifying modalities using `define-modality` in favor of the file-based definition, if that is preferred.

The main advantage to storing modality definitions outside of the Lisp image is that doing so provides way to supply lower layers of the agent implementation with the information they need to connect to the modalities' data channels. Several of the agents mentioned in previous chapters and described in detail in Chapter 7 have PMLc and SALs that are implemented in Java or Python and connect to the (Lisp) PMLb using their socket-based data channels. The Java and Python PMLc implementations discussed in §6.7 both include classes and functions for reading modality definitions from files, connecting to their data channels, and sending and receiving sensory and act data back and forth to the PMLb using the data channels.

MGLAIR supports writing modalities both to XML files and to RDF/OWL files. Functions for writing and reading modality definitions to/from both file types are provided. These include the function `agent-mods-to-xml`, which takes a name for the agent as its only argument and build an XML representation for all of the agent's modalities stored in `*modalitiesHash*`.

Chapter 7

MGLAIR Agents

This chapter presents examples of MGLAIR agents implemented using the software described in 6 and embodied in both hardware and software environments.

7.1 NXT Agent

The MGLAIR NXT agent is embodied in a hardware robot built on the LEGO NXT 2.0 platform. The agent has effectors and sensors that allow it move around the world, sense and grab objects, sense its own movements, and detect objects using vision and ultrasonic range sensing.

7.1.1 Platform Details

The NXT platform provides the building blocks required to create and use simple robots. Its servo motor-based actuators can be used to construct a number of different types of effectors, including locomotive and grasping effectors. These motors also include tachometric sensors that produce readings indicating the rotational position of the motor. The NXT also includes sensors such as a microphone to capture sound levels, light and color sensors, a touch sensor, and an ultrasonic range sensor. Additional sensor types are available both from the manufacturer and from hobbyists and other third parties.

The NXT sensors and effectors attach to a small programmable computer (the "NXT Intelligent Brick") that interfaces with them. This brick can also communicate with other computing devices via either bluetooth or USB connections. In keeping with the spirit of LEGO building blocks, these basic components can be rearranged endlessly in combination with interlocking plastic pieces (blocks, wheels, etc) to create simple

robots with many different shapes and capabilities.

MGLAIR agents like the one described below use the *nxt-python* [1] package in conjunction with the Python PMLc discussed briefly in §6.7 and presented in full in Appendix A.

7.1.2 NXT Agent Embodiment

The MGLAIR NXT bot, parts of which have been used as an example throughout this dissertation, uses three servo motors: two to move its wheels and one to operate its grabber. It uses the touch sensor, the microphone, and the ultrasonic range sensor

The MGLAIR NXT agent's SAL includes the NXT Brick. The SAL is connected through either its USB or Bluetooth capabilities to the PML, which is implemented in the Python programming language and runs on a separate computer. The agent-specific portions of the PMLc for each modality convert act impulses arriving from the PMLb into SAL commands, executed by calling functions in the `nxt-python` package for handling the various effectors.

In practice the software implementing the agent's PMLc has been run the same physical computer that the rest of the agent's implementation is running on, but this is not necessary, and it has sometimes been advantageous (for technical reasons unrelated to the design of the architecture or the agent) to run the agent's PMLc within a virtual machine that just happens to be on the same physical computer as the rest of the agent's implementation. The NXT agent's PMLa and PMLb are implemented in Common Lisp. The PMLb is connected to the PMLc using the modality data channels introduced in earlier chapters.

This agent also uses a visual modality that is connected to a webcam that is not part of the NXT system. This ability to add to an agent's embodiment a separate piece of hardware with an SAL that (as in this case) is run on a separate physical computer is a nice side effect of MGLAIR's modular division into layers, and the division of it's capabilities into separate modalities. In this case both the NXT and the webcam SAL communicate with the upper layers of the PML through Python: the NXT with `nxt-python`, and the OpenCV-based [2] vision system.

7.1.3 Agent Modalities

The agent embodied in this simple robot has the following efferent modalities:

- Wheeled Locomotion - `move forward/backward`, `turn left/right`

[1] `https://code.google.com/p/nxt-python/`

[2] `http://opencv.willowgarage.com/documentation/python/`

The NXT robot for this agent has two drive wheels, each connected to its own servo motor. These can be used together to effect locomotive actions such as moving and turning the agent's body in the world. The agent can monitor the progress of such actions by feeling (i.e. becoming consciously aware) that its wheels are moving in a particular way through the use of the `locomotive-proprioception` modality discussed below.

- Grasping Effector - `grab`, `release`

  The NXT robot's grasping effector uses the third servo motor. It is mounted on the front of the robot and can be opened or closed to allow the agent to grasp objects in front of it that are of the appropriate size. The grasping effector modality is used as an example of an efferent modality, and described in detail in §5.5.4.

  Note that the hardware would allow the servo motor for the grasper to be used as a sensor for a proprioceptive modality that would allow the agent to sense when its grasper is in use. That feature is not currently implemented, but the implementation would be fairly straightforward using the locomotion and locomotive proprioception modalities as models. Mounted on the front of the body but inside of the grasper is a touch sensor that allows the agent to sense when it has something in its grasper (see discussion of the `tactile` modality below).

  Full implementation details for this modality appear in Appendix B

Its afferent modalities include:

- Locomotion-proprioception - `Moving(forward), etc`

  The agent uses its proprioceptive modality to passively sense when its wheels are in motion, and in which way they are moving. This ability to monitor the success and termination of its own actions has many potential uses, as discussed earlier in this dissertation. *when its wheels are already moving.*

- Tactile - `Feel(bump)`

  The agent's tactile/bump sensor is located within its grabber. It functions as a passive sensor that allows the agent to consciously know at all times whether there is an object within its grabber. This is used by the agent both to initially detect the presence within its grabber of an object the agent is trying to retrieve, as well as to monitor the status of a held object as the agent is moving around. It is not uncommon for an object (e.g. a ball) to escape from the grabber while the agent is moving or turning. The agent should notice events like this immediately and react accordingly, rather than, say,

continuing to drive around in service of some goal that requires the agent to have the object in its possession.

- Range Sensor - `DistIs(near),...`

  The NXT agent's ultrasonic range sensor is discussed in some detail as an example of an afferent modality in an earlier section. See §5.5.2 for a detailed description of the modality, its configuration, and its use. Recall that the agent has an afferent modality for its range sensor that allows the agent to sense the distance to the nearest solid object in front of it. Details of this modality's low-level implementation at the PMLc and SAL appear below. Full implementation details appear in Appendix B

- Visual Object Detection - `SizeIs(obj,large), ColorIs(obj,red), etc`

  The NXT agent includes a non-NXT sensory modality: a visual modality that currently allows the agent to sense the presence of simple objects (colored balls), as well as some of their properties (e.g., color, apparent size, rough location in the field of vision). This modality uses a simple webcam as its sensor, which is connected using the Python MGLAIR PMLc and Python libraries for OpenCV.

  Because the focus of this work is not computer vision, and because vision is often temperamental, there is also an alternative interactive implementation of the same modality that uses a Java program hooked up to the Java PMLc, allowing the user to create visual sensations for the agent. This is discussed more below in 7.1.8.

Modality Definitions

Below is the definition for the agent's `locomotion` and `vision` modality.

Listing 22 Definition for efferent locomotive modality

```
(define-modality  'locomotion
   'efferent-modality
  :frames '(turn go)
  :description "Used by the agent to move and turn"
  :channel '((port . 9576) (host . localhost))
  :conflict 'blocking
  :print t)
```

Listing 23 Definition for afferent visual modality

```
(define-modality 'vision2
   'afferent-modality
  :frames '(BallIs SizeOf LocatedAt)
  :focus '(nil . 1)
  :channel '((port . 9573) (host . localhost))
  :buffer '((capacity . 5))
  :conflict 'squeeze
  :sense-handler 'sense-vision
  :description "Visual modality used by the agent to detect objects"
  :print t)
```

7.1.4 PMLa

When an MGLAIR agent consciously performs a primitive action, the PMLa primitive action function associated with that action is called. A few examples from the NXT agent appear below.

The NXT agent operates its grabber by `performing` `grab` or `release`. The PMLa action `grab-act` implements closing the grabber by calling PMLb's `grab`, passing it the name of the modality and the value `t`. `release-act` (not shown) is similar but passes `nil` to PMLb:grab, which causes the grabber to be opened.

Listing 24 The grasping effector's `grab-act` primitive action

```
(define-primaction grab-act ()
 (sneps:clear-infer)
 (PMLb:grab 'grabber t))
```

The locomotive primitive action `go-act` calls `PMLb:forward` or `PMLb:backward` depending on its arguments. `turn-act` (not shown) is similar.

Listing 25 Locomotive modality's `go-act` primitive action

```
(define-primaction go-act ((godir))
  (cond ((eq godir (sneps:node 'forward))
         (PMLb:forward 'locomotion))
        ((eq godir (sneps:node 'backward))
         (PMLb:backward 'locomotion))
        (otherwise (format sneps:outunit "~&***Invalid go direction: ~S~%" godir))))
```

7.1.5 PMLs

The PMLs is the last stop along the modality for sensory data as it is transformed into conscious percepts. A simple example is the NXT agent's range sensing modality. The PMLs `perceive-distance` function is called by the `range` modality's sense handler, and passed a scalar value between 0 and 255 inclusive. In this implementation the function used to translate this value into a KL term is very straightforward, using a set of conditionals to divide the possible values into a few groups. This works fine for such a simple percept, though having such values hard-coded in to the agent's perception is perhaps less than ideal. A more sophisticated implementation might group the values based on a clustering algorithm, or allow for a period of training in which the agent learns what `near`, `far`, etc look like.

Listing 26 `perceive-distance` converts the scalar distance value into a KL term for the agent's conscious perception

```
(defun perceive-distance (d)
  "takes an int between 0 and 255, asserts the distance based on this (close, far, etc)"
  (believe "DistIs(~A)" (interpret-distance d)))

(defun interpret-distance (dist)
  (cond ((< dist 30) 'close)
        ((< dist 60) 'medium)
        ((< dist 120) 'far)
        (t 'distant)))
```

The PMLs `perceive-vision` function in the NXT agent's visual modality receives from the PMLb structures like the following:

```
((loc . (0.322 . 0.025)) (size . 2.59) (color . (120.0 255.0 1.0)))
```

The first element of this list specifies the center of an object that the visual modality is sensing. The second element specifies the object's size (the percentage of the visual field it takes up), and the third element gives information about the object's color (Hue, Saturation, and Value).

The `interpret-color` function, like the `interpret-distance` function above, performs a very simple translation of values into KL terms representing them; the same criticism and suggestion for improvement applied to both.

`interpret-color` and similar functions are used by `perceive-vision` to build KL terms, which are then asserted using the built-in `PMLs` function `believe`.

Listing 27 `perceive-vision` converts a multi-element structure into assertions about the object it represents

```
(defun interpret-color (hsv)
  (if (isabout 120 (first hsv)) 'blue
      (if (isabout 0 (first hsv)) 'red
          'unknown)))
```

Listing 28 `perceive-vision` converts a multi-element structure into assertions about the object it represents

```
(defun perceive-vision (visuals)
  (let ((loc (interpret-location (rest (first visuals))))
        (size (interpret-size (rest (second visuals))))
        (color (interpret-color (rest (third visuals))))
        (obj "ball"))
    (believe "ColorOf(~A,~A)" obj color)
    (believe "SizeOf(~A,~A)" obj size)
    (believe "LocatedAt(~A,~A)" obj loc)))
```

7.1.6 PMLb

Much of the work performed at the PMLb for any agent is handled by the built-in modality mechanisms managing modality communication, buffers and focus, etc. The agent-specific parts of the PMLb tend to be fairly lean, focused on transforming act impulses and passing them along to the PMLc, or on passing to the PMLs sensory data in the appropriate format. This section presents examples from the NXT agent's efferent and afferent modalities.

Efferent

Below are definitions for `turnLeft` and `forward`, which use the agent's locomotive modality to execute the respective actions. They work by calling the built-in function `execute` and passing it the modality and a string that represents the act impulse. The string is produced with the helper function `make-tuple-str`, which takes Lisp lists or atoms and converts them into strings that Python can evaluate as tuples. The agent's PMLc expects messages from the PMLb to be in this format.

Listing 29 `turnLeft` and `forward`, two locomotive actions at the PMLb

```
(defun turnLeft (mod) (execute mod (make-tuple-str `(t left))))
(defun forward (mod &optional (units 1))
    (execute mod (make-tuple-str `(f ,units))))
```

Afferent

The agent's sense handlers in the PMLb are also simple, generally stripping away the timestamp given to the the sensory data when it was added to the modality's buffer, and either passing along the (string) result or reading the result and then passing along the resulting form to the PMLs function that finalizes the conscious perception of the data. Two examples are `sense-motion` and `sense-vision`:

Listing 30 `sense-motion` and `sense-vision` are two sense handlers in the PMLb

```
(defun sense-motion (input)
  (PMLs::perceive-motion (rest input)))
(defun sense-vision(d)
  (PMLs::perceive-vision (read-from-string (rest d))))
```

Recall that a modality's PMLb sense handlers can be associated with the modality at the time it is defined, and are then called automatically by the buffer management processes when a new piece of sensory data emerges from the buffer.

7.1.7 PMLc

The NXT agent's PMLc is implemented in Python and uses the provided classes for connecting to and using modality data channels:

Listing 31 Initializing modality data connections

```
a = NXTBot.NXTBot() # for interacting with the SAL
# initialize modality data channel connections
mm = ModalityManager("nxt-agent-config.xml", a)
mm.connectModalities('locomotion','vision','grabber', 'tactile', 'locomotive-proprioception')
a.setModalityManager(mm)
```

The thread monitoring each efferent modality calls the PMLc's `handleImpulse` method when a new act impulse arrives, passing the name of the modality along with the impulse. `handleImpulse` directs the impulse to the agent's modality-specific handler, e.g. `handleLocomotion`.

`iToTup` takes a string representing the act impulse and converts it to a Python tuple using string processing and the `literal_eval` function. The agent's PMLb is designed to prepare and send messages that look like readable Python objects in order to facilitate processing them at the PMLc.

`handleLocomotion` examines the result and calls the corresponding method to move the robot's wheels.

Listing 32 `handleImpulse` is called by every afferent modality's data channel watcher when a new impulse arrives

```
def handleImpulseM(self, mod, i):
    tup = self.iToTup(i)
    if mod == 'locomotion':
        self.handleLocomotion(tup)
    elif mod == 'grabber':
        if(tup[0] == "g"):
            self.moveGrabber(tup[1])
            return
 ...
```

Listing 33

```
def handleLocomotion(self,tup):
    if(tup[0] == "t"):
        if(tup[1] ==  "left"):
            self.turnLeft()
        elif(tup[1] == "right"):
            self.turnRight()
    elif(tup[0] == "f"):
        self.forward()
    elif(tup[0] == "b"):
        self.backward()
    else:
        print "got a bad locomotion impulse: " + tup
```

7.1.8 SAL

The agent's SAL is implemented on the NXT brick, which is accessed using the `nxt-python` package, which provides classes and methods that wrap the basic sensors and effectors, presenting them as objects that abstract away the details of the micro-controller and communication with it over USB (or Bluetooth).

An exception is the SAL for the agent's visual modality, discussed more below, which is implemented separately from the NXT.

Acting in The SAL

Listing 34 `turnmotor` turns a motor at a specified power level for a specified number of degrees

```
def turnmotor(self, m, power, degrees):
    m.turn(power, degrees)
```

NXTBot.`runinstruction` takes a list as its argument and unpacks that list into `motorid`, which identifies a motor; a `speed` at which to run the motor; and a number of `degrees` to turn it. ¡It gets the motor object associated with that `motorid` and starts a new thread to turn the motor with those parameters. An example use of `runinstruction` by `turnLeft` appears below.

Listing 35 `runinstruction` starts a new thread to run a motor instruction

```
def runinstruction(self, i):
    motorid, speed, degrees = i
    thread.start_new_thread(
            self.turnmotor,
            (self.motors[motorid], speed, degrees))
```

To effect turning left at the SAL, the bot moves both of its wheels' motors, one in one direction and the other in the opposite direction. This is accomplished by starting two threads using `runinstruction`, one for each wheel. Because `runinstruction` spawns a new thread for each motor command, these two are run simultaneously.

Listing 36 `turnLeft` uses two motors to turn the agent's body to the left

```
    def turnLeft(self):
        for i in ([0, 1, 40, 180], [0, 0, -40, 180]):
                    self.runinstruction(i[1:])
```

The SAL implementation of `moveGrabber`, which is used to open or close the grasping effector is simpler because it only needs to use a single motor:

Listing 37 `moveGrabber` uses a single motor to open or close the grasping effector

```
def moveGrabber(self, close):
    Motor(self.b, PORT_A).turn(-10 if close else 10, 25)
```

Sensing in the SAL

The agent's passively-sensing afferent modalities are implemented with a separate thread for each sensor that polls for its value. For instance, the method `getTouch` is used to get a boolean value from the bot's simple touch sensor: `True` if the sensor is depressed and `False` otherwise.

Listing 38 SAL method to get the touch sensor's value

```
def getTouch(self):
    return Touch(self.b,PORT_1).get_sample()
```

`getTouch` is used by the `TouchSensor` thread to constantly check the sensor's value. The thread repeatedly polls the sensor's value, storing the last value seen. When the value changes, it calls the `senseTouch` method, which causes a message to be sent up into `tactile` modality in the PMLc, and from there into the modality's data channel and beyond.

The ultrasonic sensor, discussed in more detail in §5.5.4, has a similar implementation at the SAL.

Sensing Acting: Proprioception in the SAL

When the agent's SAL executes a command to move (including turning) using its locomotive modality, it also spawns a new thread to watch the servo motors' progress, and to send messages up through the agent's `locomotive-proprioception` modality indicating whether the motors are currently turning. This is accomplished by first sending the message corresponding to the motor's being in use, then running a loop that constantly checks whether the motors have reached their target position. When they have, a message is sent indicating that the motor is no longer in use. An alternative implementation would send messages constantly whether the motors are in use or not, and allow the agent to process and perceive that signal all the time. The main advantage to the current implementation is that it eliminates constant processing of

Listing 39 Class for `TouchSensor` thread

```
class TouchSensor(threading.Thread):
    def run(self):
        self.listen()

    def __init__(self, bot):
        self.bot = bot
        threading.Thread.__init__(self)

    def listen(self):
        last = False
        while True:
            touchp = self.bot.getTouch()
            if last != touchp:
                last = touchp
                self.bot.senseTouch(touchp)
            time.sleep(.01)
```

redundant messages. A more cognitively plausible approach to this might use sensory fatiguing to allow the agent to come to ignore repetitive sensations.

Listing 40 `TachCheck` class used to run a thread that monitors the motion of the agent's motors

```
class TachCheck(threading.Thread):

...

    def run(self):
        self.listen()

    def listen(self):
        self.bot.senseMotion(self.direction)
        while True:
            tach  = self.bot.getTacho()
            if (self.bot.getTacho().is_near(self.target, self.inc*.05)):
                break
        self.bot.senseMotion(False)
```

`senseMotion` gets the PMLc object for the `locomotive-proprioception` modality and passes it a message corresponding to the current state of motion:

```
self.modmgr.getMCByName('locomotive-proprioception').setSense(str(moving))
```

Webcam-based Vision

The NXT agent is equipped with a webcam-based visual system capable of detecting objects and their basic properties, i.e. size, color, location in the visual field.

The PMLc for this is implemented using Python and OpenCV, but not nxt-python. It performs basic thresholding on images (arrays of pixels) produced by the webcam (SAL), to isolate simple objects in the visual field that match particular color ranges (in HSV color space). See Figure 7.1 *(a)* for a view of a red ball in HSV color space, *(b)* for a binarized version used to pick out the ball, and *(c)* for an interactive simulated version implemented in Java that allows the agent to "see" objects when it is not actually connected to the webcam.

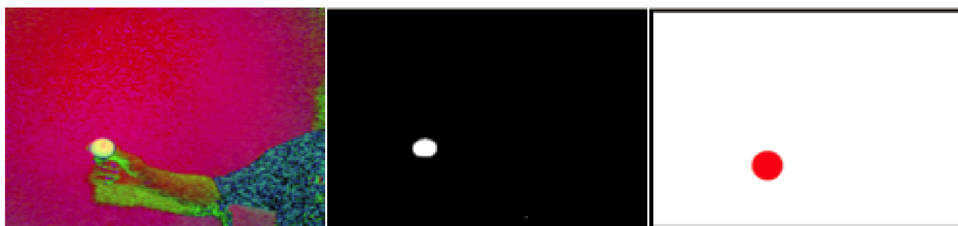

Figure 7.1: (a) a view in HSV color space, (b) a binary image in which the red ball is isolated, (c) simulated version of red ball

The PMLc/SAL for this modality runs on a separate physical computer from the rest of the agent's modalities. Like them, it is connected via a data channel to the PMLb, which has no preference, or even knowledge, of where the lower parts of its modalities are located.

This arrangement allows the agent to perceive, e.g., that there is a small red ball in the left portion of its visual system using the version of `perceive-vision` defined above.

The PMLc for this modality sends into the data channel structures like:

```
((loc .  (0.322 .  0.025)) (size .  2.59) (color .  (120.0 255.0 1.0)))
```

Simulated Vision

Figure 7.1 *(c)* shows an alternative mode of vision for the agent that produces similar sensations based not on the output of pixels from a webcam, but on a user's interactively creating objects for the agent to see. This interface is useful for testing and demonstrating the proper functioning of the agent' higher-level visual perception without the use of a temperamental webcam sensitive to small changes in lighting or other aspects of the environment.

7.1.9 Knowledge Layer

The agent perceives the world by having asserted to its knowledge layer terms that correspond to the parts of the world its sensors are detecting.

For example, the agent is capable of perceiving that it is currently moving forward, during which it believes `Moving(forward)`. When its wheels have stopped turning, the agent passively (i.e. without having to try and figure it out) perceives that it is no longer moving: `Moving(forward)` (when it is still, the agent will also believe that it's not moving backward, etc):

Listing 41 The agent's KL includes the knowledge that it can be moving in at most one direction at a time

```
MoveDir({forward, backward, left, right}).
nexists(_,1,_)(d)(MoveDir(d): Moving(d)).
```

The agent also believes, e.g., that everything has only one color (this is true in its simple model of vision), and that everything has only one location and one size at a time. As well, it believes that the distance to the nearest object in front of it at most one of `close`, `medium`, `far`, and `distant`.

The agent is able to act while simultaneously using its afferent modalities to perceive changes in the world as they happen, in real-time, without having to consciously act to check on its conditions for further action.

For instance, the following ActPlan allows the agent to infer plans for aligning itself with a ball that is positioned in the left area of its visual field: *the ActPlan for aligning with a ball is to turn in the direction of the ball and then continue aligning with it.* As the agent moves and turns, the location of the target object within its visual field changes accordingly, and these changes are immediately passively perceived and taken into consideration in carrying out the plan. The result is that the agent moves left only so long as the object is to the left of its visual field, moves right so long as it's to the right, and stops performing the `alignWith` task when the object is centered.

Listing 42 An ActPlan for aligning the agent's body with a target object

```
  ActPlan(alignWith(ball),
          snif({if(LocatedAt(ball,right), snsequence(turn(right), alignWith(ball))),
                if(LocatedAt(ball,left), snsequence(turn(left), alignWith(ball)))})).
```

Note that this rule may appear to be redundant and overly specific because of the repetition of `left` and `right`. A more general rule could be formed, but note that the fact that the turning direction here

and the symbols the agent uses to represent the position of the object in its visual field just happen to be the same symbols, and this is not the case for related terms like `center` and `forward` – i.e. there are no `LocatedAt(ball, forward)`, `turn(center)`, etc.

Listing 43 An ActPlan for going to an object

```
;; To go to the ball: if it's in the center of your vision, approach it
;; Otherwise, align the center of your vision and then go to the ball
ActPlan(goTo(ball), snif({if(LocatedAt(ball,center), approach(ball)),
                          else(snsequence(alignWith(ball), goTo(ball)))})).
```

The NXT agent uses proprioception to monitor the effectiveness and progress of its actions at the KL. The ability to sense when it is currently moving allows the agent to avoid sending act impulses to move when its body is already in motion. For example, consider the non-primitive act `goForward()`, the plan for which involved performing the primitive act `go(forward)` only if the agent is not already moving in that direction.

Listing 44 An ActPlan for moving forward & monitoring that motion with proprioception

```
;; Moving and turning informed by proprioception
ActPlan(goForward(),snif({if(~Moving(forward), go(forward))})).
```

Note that writing rules like this for every possible action will be cumbersome at best. There is room here for improvement by extending how this is handled in the architecture with an abstraction that captures the idea of monitoring the progress of an act through an agent's sensations.

7.2 CREATE Agent

I have instantiated a cognitive agent embodied in an iRobot Create, which is equipped with a basic set of built-in sensors and effectors such as a bump sensor and wheeled locomotion. The Create is intended for robot development and has cargo capacity and a serial port interface that allows operation of the system by devices other than the on-board micro-controller. The robot's basic form is similar to that of iRobot's Roomba robotic vacuum cleaners, but without the ability to clean the floor. I have physically extended the robot's body with a platform on which a small laptop sits. The laptop connects to the robot hardware through a serial port using a USB-to-serial adapter. Python software on the laptop implements the robots basic capabilities (moving, turning, sensing bumps, and so on) and connects the modalities for these capabilities to an MGLAIR agent's PMLb using the Python PMLc implementation. The laptop that drives the iRobot

has a built-in camera that can be used to implement a visual system for the agent, similar to the visual modality discussed in the previous section. While the Python/OpenCV implementation of that webcam modality is portable and could be moved directly to the iRobot agent with little or no modification, that has not been attempted at this point.

7.2.1 Vestibular Sense

I have embodied an MGLAIR agent in the iRobot Create, augmented with a tri-axial accelerometer. The accelerometer allows the agent to feel when its body is in motion, whether that motion is the result of the agent's moving its wheels, or the agent's being located in a moving vehicle, etc. This constitutes a vestibular sense for the agent.

Rather than having the features that identify different types of motion hard-coded into its PMLs perceptual function or elsewhere, the agent is trained with supervised learning to know what different locomotive actions feel like. This involves instructing the agent to drive around while training its classifier. The classifier learns what different motions feel like by paying attention to the symbolic representations of the agent's act impulses as they are performed. These symbols are used as the labels for the types of motion it learns. The features used to lean the classes include acceleration values along each of the three axes.

After training, the `vestibular` modality's classifier in the PMLc loses access to the symbols for actions as they are performed, and takes as its input only acceleration features originating in the sensor. It uses these features to classify the motion occurring at each time, and sends the result - a symbols matching the KL term for the type of motion the classifier has identified - up along the modality. The agent senses the result as things like *my body is turning*.

In the current implementation of the iRobot's vestibular modality, the (K nearest neighbor) classifier in the PMLc that learns what locomotive acts feel like is trained initially but does not continue learning after initial training. One useful modification to this would be to have the classifier continue training throughout the agent's operation. This should improve recognition and allow the agent to adapt to changes in the environment that otherwise might confuse its senses.

An accelerometer actually detects senses acceleration, not mere motion. In practice, however, remaining stationary looks different enough from moving forward in a straight line even over smooth floors at the agent's best approximation of a constant speed. When differentiating between five different motion classes (moving forward or backward, turning left or right, stopped), the current implementation is right about half of the time, with f-scores of approximately 0.5. Though it's much better than a simple guess, there is clear

room for improvement in the form of a more sophisticated classifier or use of different parameters. It has been suggested to me by Robert Platt (personal communication, April 2012) that Kalman Filters [Thrun, 2002] would be well-suited to this task.

Interesting future work on this agent includes outfitting it with a locomotive proprioceptive modality in addition to its vestibular modality and exploring the types of meta-reasoning and robust action that can be implemented by combining these two sources of information about the state of the agent's body.

7.3 Greenfoot Agent

This section describes an agent implemented in a simulated world that uses Greenfoot. The Greenfoot framework is a pedagogical tool designed for teaching introductory programming in Java to beginning students [Kölling, 2010]. Greenfoot runs on Java and is self-contained, even including its own integrated development environment. Normal use of Greenfoot involves creating scenarios consisting of a World (subclass of greenfoot.World) and Actors (subclasses of greenfoot.Actor), and modifying those Java classes to achieve the desired behavior for the scenario. The Greenfoot Tutorial [3] takes as an example a simple "Wombats" scenario – a 2D grid-based World with such Actors as Leaves and Wombats that move around the grid eating Leaves.

I have developed an interface to allow the use of Greenfoot scenarios as simple virtual worlds in which SNePS agents can operate. Creating Greenfoot MGLAIR agents involves the use of the Java PMLc described in Chapter 6 to interface with the agent's SAL, which is implemented in Greenfoot. This use of Greenfoot as an environment for embodying simple cognitive agents whose minds are implemented using MGLAIR is decidedly "off-label": neither intended, anticipated, nor supported by Greenfoot itself.

Appendix C shows how MGLAIR's PMLc connects to and uses Greenfoot agents. More details on how to access Greenfoot from SNePS/MGLAIR appear in [Bona and Shapiro, 2011].

Previous GLAIR agents that used Greenfoot connected directly to it using Allegro Common Lisp's JLinker[4], which allowed the PMLb functions to directly call Java methods implementing the PMLc or SAL. MGLAIR agents that use Greenfoot scenarios use modality data channels to establish simultaneous communication between the layers for each modality. In the Delivery Agent discussed below, the agent's PMLb uses the JLinker to initialize the Greenfoot world and SAL/PMLc. This is convenient but not strictly necessary and no communication between the layers uses this interface. An alternative would be to start the

[3] Greenfoot Tutorial: `http://www.greenfoot.org/doc/tut-1`, last accessed 30 July 2013
[4] `http://www.franz.com/support/documentation/8.2/doc/jlinker.htm`

agent's mind and the world separately each time the agent is run.

7.3.1 Agent Overview

The MGLAIR delivery agent operates in a 2D grid-based world that simulates the floor of a building with offices. The simulation consists of corridors, and rooms that have entrances on the corridors. The rooms are numbered, and some of them contain packages. The agent has a grasping effector that it uses to pick up and put down packages. Using its simple visual modality, the agent can read the room numbers and see what is in the cell it is facing (options include a wall, an entrance to a room, or part of a corridor. It can also sense the approximate distance to the nearest main corridor that it's facing using its range sensing modality. The agent's other capabilities include locomotion (it can turn left or right and go forward). Figure 7.2 shows the agent in the upper-left corner of its environment. The environment includes rooms numbered `1` through `15`, some of which contain randomly-placed packages.

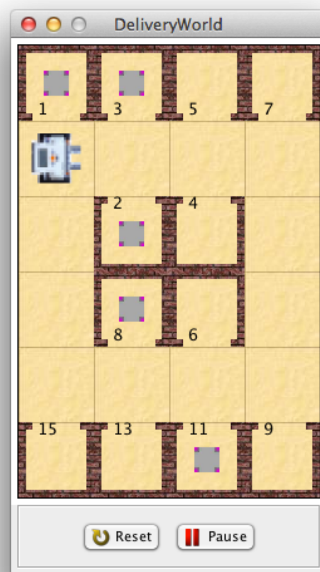

Figure 7.2: Delivery Agent in Greenfoot Delivery World

7.3.2 Agent Modalities

The Delivery agent has the following modalities:

- `vision` - an afferent modality that allows the agent to perceive what's in front of it.

- `range` - an afferent modality that allows the agent to perceive the rough distance to the next obstacle. This is used by the agent to tell whether it is facing the closest main corridor.

- `locomotion` - an efferent modality that allows the agent to move through its world.

- `grabber` - an efferent modality that the agent uses to operate its grasping effector.

- `visual-efferent` - an efferent modality used to direct the agent's visual attention to particular tasks (see §7.3.3).

7.3.3 Active and Passive Vision

Whenever the agent moves, it automatically perceives what is in front of it. In contrast, the agent only reads the room number of a room it is facing when it deliberately directs its attention to that task. That is, the agent's SAL passively produces visual sensory data that is sent up the `vision` modality automatically whenever its view changes, but only produces sensory data pertaining to the room number label when the agent explicitly acts in its `visual-efferent` modality, e.g. by performing the action `readRoomNumber()`.

7.3.4 Modality Definitions

Before they are used (attached to a PMLa primaction, used to communicate between the PMLb and PMLc, etc), modalities must be defined.

The following is a modality definition for an afferent modality named `vision` that is of type `afferent-modality`, uses the predicate `Facing` in its percepts, has an initially elevated `focus` level that cannot be modified by the agent, and uses a (socket-based) `data channel` on port 9574.

Listing 45 Defining a visual modality

```
(define-modality 'vision
  'visual-modality
  :frames '(Facing)
  :access t
  :focus '(t . elevated)
  :buffer '((exp-int . 1000))
  :channel '((port . 9574) (host . localhost))
  :sense-handler 'vision-sense-handler
  :description "Visual modality used by the agent to detect what's in front of it"
  :print t)
```

The following is a modality definition for an efferent modality named `grabber` that is of type `efferent-modality`, uses the predicates `pickUp`, `putDown` in its actions, and uses a (socket-based) `data channel` on port 9577.

Listing 46 Defining an efferent grasping effector modality

```
(define-modality 'grabber
  'efferent-modality
  :frames '(pickUp putDown)
  :channel '((port . 9577) (host . localhost))
  :focus '(nil . 0)
  :description "Used to manipulate the agent's grasping effector"
  :print t)
```

7.3.5 PMLa

The PMLa implements the agent's primitive actions. A simple example is the `pickup-act` action, which simply calls the matching function in the PMLb, passing it the name of the modality:

Listing 47

```
(define-primaction pickup-act ()
  (PMLb:pickup 'grabber))
```

This is attached to `pickUp` action nodes at the KL as follows:

```
(attach-primaction pickUp pickUp)
```

The PMLa primitive acts for `putDown` and `readRoomNumber` are similarly simple – these acts also take no arguments. The definition for `turn` below is more complicated because the act takes a direction as an argument, and the SNePS node for the direction must be processed to get a symbol for the direction that the PMLb `turn` function understands.

Listing 48 `turn-act` primitive action

```
(define-primaction turn-act ((dir))
  "Go left, or right, according to the given direction."
  (case (sneps:node-na dir)
    ('left (PMLb:turn 'locomotion 'PMLb::l))
    ('right (PMLb:turn 'locomotion 'PMLb::r))
    (otherwise (format sneps:outunit "~&***Invalid turn direction: ~S~%" (sneps:node-na dir)))))
```

7.3.6 PMLs

The PMLs implements the final transformation of perceptual data that has traveled up through the modality's data channel and perceptual buffer. PMLs functions are called by the perceptual functions that are applied to each structure as it emerges from the modality's perceptual buffer. For example, the PMLs function `perceive-distance` is called with arguments and results in the agent performing the mental act of believing that the distance to the nearest thing in front of it is near (or far).

Listing 49 PMLc `perceive-distance` perceptual function

```
(defun perceive-distance (d)
  (when (string= (symbol-name (first d)) "dist")
    (believe "~A(~A)" 'DistIs (case (rest d)
                                (0 'near)
                                (t 'far)))))
```

Another example is the PMLs `perceive-ahead` function, which translates symbols like `r` or `w` into terms like `room` or `wall`, and performs `believe(AheadIs(room))`, etc:

Listing 50 PMLs `perceive-ahead` function

```
(defun perceive-ahead (p)
  (believe "~A(~A)" 'AheadIs
           (cond ((string= p "r") 'room)
                 ((string= p "c") 'corridor)
                 ((string= p "w") 'wall)
                 (t 'other))))
```

7.3.7 PMLb

Much of the action at the PMLb is built-in functionality for using modalities. This includes functions for handling the data channel that connects the PMLb to the PMLc, modality buffer management, implementation of the focus mechanisms, and so

In afferent modalities, PMLb functions such as `PMLb:turn` translate their arguments into impulses that the PMLc will understand, then sends them on to `PMLb:execute`, as in the following.

Listing 51 PMLb `turn` passes to execute a modality name and an impulse the PMLc can handle

```
(defun turn (mod dir)
  (execute mod (format  nil "~A" (cons 'tn (if (eq dir 'l) -90 90)))))
```

Data channel handling processes - one per afferent modality - in the PMLb monitor the data channel for sensory data arriving from the PMLc, and add it to the modality's percept buffer. When these data are removed from the buffer, they are subjected to the modality's perceptual process, which performs some translation and passes the result on to the appropriate PMLs function.

PMLb sense handler functions like `range-sense-handler` are automatically applied to timestamped sensory structures in their modality (e.g. `(6454532432 . (dist . 1))`) when the structure has made it through the modality buffer. This function translates the structure into a symbol understood by the `perceive-distance` function in the PMLs, which is responsible for building the KL term that constitutes the percept and asserting it.

Listing 52 The PMLb's `range-sense-handler` is automatically applied to the `range` buffer's contents as they are removed in order

```
(defun range-sense-handler (v)
  (PMLs:perceive-distance (read-from-string (rest v))))
```

7.3.8 PMLc

The agent's PMLc is implemented in Java and connects the PMLb to the agent's body, which is implemented in Java on Greenfoot. The PMLc implementation includes a set of Java classes and methods for reading modality configuration files, which describe each modality's properties, including its name and type, and details on how to connect to the data channel at the PMLb. For instance, the Delivery agent's PMLc includes the following two lines, which create a `ModalityManager` object from a configuration file, and then connect the named modalities. The result is that each modality is assigned a `ModalityConnector` object that for afferent modalities handles watching for act impulses incoming from the PMLb, and for efferent modalities handles sending sensory data up to the PMLb when it is produced at the SAL.

```
modMgr = new ModalityManager(configFN,this);
modMgr.connectModalities("locomotion","grabber","vision", "visual-efferent", "range");
```

Each `ModalityConnector` also contains a method `setSense`, which sends a sensory impulse up along the data channel to the PMLb.

Act impulses arriving at the PMLc are passed to the `handleImpulse` method, along with the name of the modality in which they are occurring. `handleImpulse` passes the message along to specific method for the modality. Act impulses arriving at the PMLc for this agent are of the form:

``(<time> . (<action> . <arg>))''

Not shown here is the sequence of regular expressions to break these impulses into manageable chunks that can be sent to the individual modality methods.

Listing 53 Partial implementation of `PMLc.handleImpulse`

```
public void handleImpulse(String modname, String i){

...

   /* not shown here: splitting string i, which represents the act impulse
      into an action name and an array of additional arguments */

   if(modname.equals("visual-efferent")){
           handleVisualEfferent(aname);
   }else if(modname.equals("locomotion")){
           handleLocomotion(splitaction);
   }else if(modname.equals("grabber")){
           handleGrabber(aname);
   }
   modMgr.getMCByName(modname).confirmImpulse(impulse[1]);
}
```

The last thing the `handleImpulse` method does is get the modality object and confirm that the impulse has been executed. If the modality is configured to do so, this will result in a confirmation being sent up to the PMLb, which may be waiting to hear that the PMLc is free to execute the next action.

An example of a modality-specific method called by `handleImpulse` is `handleGrabber`, defined below. Converts the remaining portion of the impulse into commands like `bot.putDown()` at the SAL, which causes the agent to put down the package it is carrying. Methods like `handleGrabber` produce an error message (not shown) if they receive unexpected input.

Listing 54 Delivery Agent Java PMLc's `handleGrabber` method

```
private void handleGrabber(String aname){
   if(aname.equals("open")){
    bot.putDown();
   }
   else if(aname.equals("close")){
    bot.pickUp();
   }
...
}
```

7.3.9 Agent SAL

The agent's SAL is implemented in the `DeliveryAgent` class, which extends `greenfoot.Actor`. Greenfoot actors move in step with an internal clock in the framework. Each actor's `act` method is called once per time step. This feature makes it easy to design simple agents (which is the point of Greenfoot), but harder to implement agents that make reasonable and predictable use of concurrency, such as the NXT hardware agent described earlier.

The Delivery Agent includes method like `readRoomNumber`, which returns the number of the room the bot is currently facing (or 0). There is not a very elegant separation between the agent's SAL and the implementation of the World, and the SAL sometimes operates by simply calling the appropriate methods on the World object – there are, after all, no real sensor involved here.

If a room is ahead, the `readRoomNumber` method gets the coordinates of the cell one space ahead of the agent, and asks the world for the number of the room located in that cell.

Listing 55 Implementation of `readRoomNumber`, for actively sensing the number of the room the agent is facing

```
private int readRoomNumber(){
    if(aheadIs().equals("r")){
         Point aheadp = aheadPoint();
         Room r = theFloor.getRoom(aheadp.x, aheadp.y);
         return r.getRoomNum();
    }
    return 0;
}
```

Recall that the `visual-efferent` modality is used to implement active sensing. Every time the agent consciously acts to discover the room number, `senseRoomNumber` is called when the act impulse reaches the PMLc, and then `readRoomNumber` is used to send the result back up into the PMLc and beyond:

Listing 56 `senseRoomNumber` is used for active sensing of the room number the agent is facing

```
public void senseRoomNumber(){
    pmlc.handleRoom(readRoomNumber());
}
```

7.4 Conclusions

This chapter has described in some detail implemented MGLAIR agents in a variety of platforms and environments. For each of these, the concurrent use of afferent and efferent modalities is key to its operation. The agents make use of modalities in combination to monitor the effects of their own actions, to learn what particular types of actions feel like, and to implement active sensing. These examples show the versatility of MGLAIR, as well as the relative ease with which an agent in a new embodiment can be instantiated with the use of provided PMLc implementations to interface with the SAL of the agent's body.

Chapter 8

Discussion

I have presented the theory and implementation of MGLAIR, and example agents that showcase its features. MGLAIR provides a model of modality as instantiable objects to be used as parts of agent implementations. The behavior of each modality is determined by its properties specified as part of its definition. These determine how a modality is used and can be used by the agent, and how sense data or act impulses within the modality are handled under various circumstances (e.g. when a sensor is generating data faster than it can be perceived, when the agent attempts to perform an action that uses a modality that is already in use, or when different modalities differ in their importance to the agent's current task and the agent must allocate its resources accordingly). Though the upper and lower layers of the architecture may even run on separate physical machines, the architecture itself takes care of this, initiating and using abstract data channels that the layers use to communicate. The various processes that go into running a modality and managing the flow of information up or down through the layers of the architecture - through the data channels, afferent and efferent buffer, etc - are set up automatically and run concurrently and invisibly to the agent architect. The result is agents that sense and act simultaneously in multiple modalities, and that integrate sensing and acting with reasoning.

By allowing agents to act while perceiving and to perceive more than one thing at a time, MGLAIR opens the door to more effective planning and action based on the agents' having better real-time knowledge of the world, and in particular of the state of its own body. The development of a model for proprioceptive modalities in MGLAIR agents that allow the agent to actually sense what its effectors are doing is of relevance here, as is the basic accelerometer-based vestibular modality. Implemented agents use proprioception, e.g., to avoid action toward some goal that their body is already acting towards. This is an issue that is hard

to address otherwise if the agent has iterative plans that check some condition and then perform an action based on its beliefs: such plans don't know to wait until the old action is completed before initiating the same action again, and the beliefs may not have changed if, e.g. the agent is currently moving toward some target location but has not yet reached it. Configuring modality buffers to block multiple access attempts is another way of handling this. A possible area for future work here is to modify the acting system itself so that actions are not believed to be completed until they have actually completed. However, the most straightforward possible implementations of such a mechanism might end up blocking the execution of unrelated actions.

The use of modality focus allows agent architects to create agents that will behave reasonably even when tasked with some intensive modality tasks that might otherwise destroy its performance on other tasks. Particularly when the result of such interference might run contrary to the agent's goals or, worse, result in damage to the agent itself (e.g. a texting drive failing to notice that they have drifted into opposing traffic), it is worth customizing modality focus. Complementary effects can be achieved by adjusting modality buffers' properties (size, expiration, etc).

8.1 Possibilities for Future Work

There are a number of interesting directions in which work on MGLAIR can go from here. It's model of modality provides a great deal of flexibility. There is much potential for customizations that extend the basic model, e.g. by adding unique types of processing within the PML.

8.1.1 Connectionist Extensions

MGLAIR's afferent and efferent modalities are more general in nature than the perceptual and motor modules included in most cognitive architectures. Though there implemented examples of specific modalities for vision, etc (see any of the agents in Chapter 7, for instance), the implementation of MGLAIR does not ship with a standard modalities for specific types of perception. That is, there's no standard visual modality that commits to a particular way of performing machine vision. This allows the agent architect a great deal of flexibility in implementing low-level processing in modalities for the sensors that a particular agent is equipped with, but it means that MGLAIR lacks a specific account for how vision (audition, etc) is handled. One possibility for future work is to develop and commit to a particular, probably connectionist, explanation for how arrays of pixels turn into sets of features suitable for digestion by the upper levels of the PML. Of course, such an account could be presented as an extension, or even just as one instance of an

MGLAIR agent. However, if the work were general enough for use across multiple agents, the result could be reasonable seen as a new iteration of MGLAIR that increases the level of hybridism in the architecture itself – and perhaps its performance and cognitive plausibility.

The simple classifier used for the learning vestibular modality discussed in Chapter 7 is a good candidate for extension or replacement with a more sophisticated technique. In general the basic idea of including different types of processing within the PML of particular modalities to implement learning is a sound one.

8.1.2 Focus & Attention

As noted in earlier chapters, MGLAIR's model of *focus* does not claim to approach a general model of *attention*. In fact, there likely is no sufficiently realistic general model: it seems that attention functions differently in different perceptual modules, even in humans.

However, MGLAIR's focusing mechanisms would be a good starting point for the development of a model of attention, as would the sort of top-down control over afferent modalities exemplified by the MGLAIR model for active sensing. An agent that needs to selectively attend more carefully to certain visual feature, for instance might do so by modifying its own perceptual functions, consciously or otherwise, to suit its current task. Such capabilities would be of obvious use for an agent that needs to focus on identifying radically different kinds of objects using the same modality at different times. For example, consider an agent that sometimes needs to identify addressed parcels and determine to whom they are addressed, and sometimes needs to perform face detection and face recognition to determine whether the person it's looking at is the intended recipient. Such an agent's performance will suffer if it is constantly trying to recognize faces when it should actually be reading words and numbers written on a parcel.

8.1.3 Integration with SNePS Concurrent Reasoning

Current changes to SNePS include the addition of concurrent reasoning via *inference graphs* [Schlegel and Shapiro, 2013]. In order to support agents that fully take advantage of the concurrent use of multiple modalities and the ability to perform concurrent inference on the contents of the knowledge base, future versions MGLAIR should be integrated with this work.

8.1.4 Inter-layer Communication

In the current MGLAIR implementation, communication between the PMLb and PMLc via data channels is largely automated. For instance, in an afferent modality, modality-specific PMLb functions call

`PMLb:execute`, passing it an act impulse, which (after a trip through the modality buffer and data channel) arrives at the PMLc's `handleImpulse` method in turn – assuming it isn't blocked or discarded for any reason. The PMLc then examines this message and turns it into a series of low-level commands for the SAL. Currently, the agent architect must know when implementing the PMLb for that modality the format of the message that the PMLc is expecting. Or, another way of looking at it: when writing the PMLc, the agent architect must know the formatting of messages it will get from the PMLb. The recommendation is to send into a data channel a string that will be easy for the other side to handle. For instance, in some example agents the Lisp PMLb constructs a string that looks like a Python form that can be easily evaluated by the (Python) PMLc. The implementation would be improved and made easier for agent architects to use if information about the formatting of messages passing through the data channel could be provided as part of the modality description, which both the PMLc and PMLb would access and use to handle forming and parsing data.

A related issue is that the agent architect must currently define in code what happens to act impulses and sensory data within each modality at each level of the architecture. It is not a coincidence that the current implementation of the Lisp PMLb, where much of the "plumbing" for modalities is located in the architecture, also contains the simplest modality- and agent-specific functions (i.e. just sense handlers and helper functions).

8.1.5 Decay of PML structures

Another interesting direction for future work MGLAIR is exploring the role of perception and memory in agents. Currently, agents remember everything they perceive unless they decide not to believe it, e.g., because of a conflicting more recent belief. Humans, on the other hand, constantly forget things that we've seen (the alternative seems horrifying [Borges, 1962]). One possible extension to MGLAIR would link conscious percepts more closely with the structures that resulted in the agent's coming to believe them, and allow those structures to decay over time in inverse proportion to how frequently the percepts are used or thought about by the agent. The result would be increasingly fuzzy memories of things perceived – unless the agent spends a great deal of time recalling them.

8.1.6 Conclusion

These are just a few areas in which MGLAIR might soon be expanded to improve its performance, usability, or cognitive plausibility. Others include: developing ROS (Robot Operating System) [Quigley et al., 2009]

MGLAIR agents; using external ontologies like DOLCE to provide background information about modalities in the persistent modality descriptions themselves; exploring imagination and illusions through MGLAIR modalities; and the use of MGLAIR as a pedagogical tool, which would involve the development of novice-friendly interfaces. It is my hope that I and others will find the architecture's flexible model of modality, and the provided examples uses of the model, to be a good starting point for the development of such extensions.

References

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

[Anderson, 2007] Anderson, J. (2007). *How can the human mind occur in the physical universe?* Oxford University Press, USA.

[Anderson et al., 2004] Anderson, J., Bothell, D., Byrne, M., Douglass, S., Lebiere, C., and Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, 111(4):1036–1060.

[Anderson et al., 2007] Anderson, J. R., Qin, Y., Jung, K.-J., and Carter, C. S. (2007). Information-processing modules and their relative modality specificity. *Cognitive Psychology*, 54(3):185 – 217.

[Anstey et al., 2009] Anstey, J., Seyed, A., Bay-Cheng, S., Pape, D., Shapiro, S., Bona, J., and Hibit, S. (2009). The agent takes the stage. *International Journal of Arts and Technology*, 2(4):277–296.

[Arrabales et al., 2009] Arrabales, R., Ledezma, A., and Sanchis, A. (2009). A Cognitive Approach to Multimodal Attention. *Journal of Physical Agents*, 3(1):53.

[Barsalou, 2010] Barsalou, L. (2010). Grounded Cognition: Past, Present, and Future. *Topics in Cognitive Science*, 2(4):716–724.

[Barto et al., 1981] Barto, A. G., Sutton, R. S., and Brouwer, P. S. (1981). Associative search network: A reinforcement learning associative memory. *Biological cybernetics*, 40(3):201–211.

[Best and Lebiere, 2006] Best, B. and Lebiere, C. (2006). Cognitive agents interacting in real and virtual worlds. *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*, pages 186–218.

[Bona and Shapiro, 2011] Bona, J. P. and Shapiro, S. C. (2011). Creating sneps/greenfoot agents and worlds, snerg technical note 46. Technical report, Department of Computer Science and Engineering, University at Buffalo, The State University of New York.

[Borges, 1962] Borges, J. L. (1962). *Funes the memorious*, pages 59–66. New Directions Publishing.

[Byrne, 2001] Byrne, M. D. (2001). Act-r/pm and menu selection: applying a cognitive architecture to hci. *International Journal of Human-Computer Studies*, 55(1):41 – 84.

[Calvert et al., 2004] Calvert, G., Spence, C., and Stein, B. (2004). *The handbook of multisensory processes.* MIT Press.

[Chandrasekaran et al., 2005] Chandrasekaran, B., Kurup, U., and Banerjee, B. (2005). A diagrammatic reasoning architecture: Design, implementation and experiments. In *AAAI Spring Symposium: Reasoning with Mental and External Diagrams: Computational Modeling and Spatial Assistance*, pages 108–113.

[Dalton, 2000] Dalton, P. (2000). Psychophysical and behavioral characteristics of olfactory adaptation. *Chemical Senses*, 25(4):487–492.

[Derbinsky and Laird, 2010] Derbinsky, N. and Laird, J. (2010). Extending Soar with Dissociated Symbolic Memories. In *Symposium on Remembering Who We Are–Human Memory for Artificial Agents, AISB.*

[Fodor, 1983] Fodor, J. A. (1983). *The Modularity of Mind.* The MIT Press.

[Goldfain, 2008] Goldfain, A. (2008). *A computational theory of early mathematical cognition.* PhD thesis, Department of Computer Science and Engineering, State University of New York at Buffalo.

[Harnad, 1990] Harnad, S. (1990). The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1):335–346.

[Haugeland, 1989] Haugeland, J. (1989). *Artificial intelligence: The very idea.* The MIT Press.

[Hexmoor et al., 1993a] Hexmoor, H., Lammens, J., and Shapiro, S. (1993a). An autonomous agent architecture for integrating 'unconscious' and 'conscious', reasoned behaviors. In *Computer Architectures for Machine Perception*, volume 93. Citeseer.

[Hexmoor et al., 1993b] Hexmoor, H., Lammens, J., and Shapiro, S. (1993b). Embodiment in GLAIR: a grounded layered architecture with integrated reasoning for autonomous agents. In *Proceedings of The Sixth Florida AI Research Symposium (FLAIRS 93)*, pages 325–329. Citeseer.

[Hexmoor, 1995] Hexmoor, H. H. (1995). *Representing and Learning Routine Activities.* PhD thesis, University at Buffalo, The State University of New York.

[Kandefer et al., 2007] Kandefer, M., Shapiro, S., Stotz, A., and Sudit, M. (2007). Symbolic reasoning in the cyber security domain. In *Proceedings of MSS 2007 National Symposium on Sensor and Data Fusion.*

[Kieras and Meyer, 1997] Kieras, D. and Meyer, D. (1997). An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction*, 12(4):391–438.

[Kokinov, 1994] Kokinov, B. N. (1994). The dual cognitive architecture: A hybrid multi-agent approach. In *ECAI*, pages 203–207.

[Kölling, 2010] Kölling, M. (2010). The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):14.

[Kumar et al., 1988] Kumar, D., Ali, S., and Shapiro, S. C. (1988). Discussing, using and recognizing plans in SNePS preliminary report—SNACTor: An acting system. In Rao, P. V. S. and Sadanandan, P., editors, *Modern Trends in Information Technology: Proceedings of the Seventh Biennial Convention of South East Asia Regional Computer Confederation*, pages 177–182. Tata McGraw-Hill, New Delhi, India.

[Kumar and Shapiro, 1991] Kumar, D. and Shapiro, S. C. (1991). Architecture of an intelligent agent in SNePS. *SIGART Bulletin*, 2(4):89–92.

[Kurup and Chandrasekaran, 2007] Kurup, U. and Chandrasekaran, B. (2007). Modeling memories of large-scale space using a bimodal cognitive architecture. In *Proceedings of the Eighth International Conference on Cognitive Modeling*, pages 267–272.

[Laird, 2008] Laird, J. (2008). Extending the Soar cognitive architecture. In *Artificial General Intelligence 2008: Proceedings of the First AGI Conference*, page 224. IOS Press.

[Laird et al., 1987] Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: an architecture for general intelligence. *Artificial intelligence*, 33(1):1–64.

[Lakemeyer et al., 2010] Lakemeyer, G., Levesque, H., Pirri, F., Lakemeyer, G., Levesque, H., and Pirri, F. (2010). 10081 abstracts collection–cognitive robotics. *Cognitive Robotics.*

[Lakoff, 1987] Lakoff, G. (1987). *Women, fire, and dangerous things: What categories reveal about the mind.* Cambridge Univ Press.

[Langley et al., 2009] Langley, P., Laird, J., and Rogers, S. (2009). Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, 10(2):141–160.

[McGurk and MacDonald, 1976] McGurk, H. and MacDonald, J. (1976). Hearing lips and seeing voices. *Nature*, 264(5588):746–748.

[Minsky, 1988] Minsky, M. (1988). *Society of mind*. Simon and Schuster (New York).

[Nayak et al., 2004] Nayak, T., Kandefer, M., and Sutanto, L. (2004). Reinventing the reinvented shakey in sneps. Technical report, Department of Computer Science and Engineering, University at Buffalo, The State University of New York.

[Newell, 1994] Newell, A. (1994). *Unified theories of cognition*, volume 187. Harvard University Press.

[Newell et al., 1959] Newell, A., Shaw, J. C., and Simon, H. A. (1959). Report on a general problem-solving program. In *IFIP Congress*, pages 256–264.

[Newell and Simon, 1976] Newell, A. and Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126.

[Nilsson, 2007] Nilsson, N. J. (2007). The physical symbol system hypothesis: status and prospects. In *50 years of artificial intelligence*, pages 9–17. Springer.

[Pew and Mavor, 1998] Pew, R. and Mavor, A. (1998). *Modeling human and organizational behavior: Application to military simulations*. National Academies Press.

[Pylyshyn, 1999] Pylyshyn, Z. (1999). Is vision continuous with cognition?: The case for cognitive impenetrability of visual perception. *Behavioral and Brain Sciences*, 22(03):341–365.

[Quigley et al., 2009] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3.

[Rickel and Johnson, 2000] Rickel, J. and Johnson, W. (2000). Task-oriented collaboration with embodied agents in virtual worlds. *Embodied conversational agents*, pages 95–122.

[Rohrer, 2011] Rohrer, B. (2011). An implemented architecture for feature creation and general reinforcement learning. In *Workshop on Self-Programming in AGI Systems, Fourth International Conference on Artificial General Intelligence*.

[Rohrer, 2012] Rohrer, B. (2012). Becca: Reintegrating ai for natural world interaction. In *AAAI Spring Symposium on Designing Intelligent Robots: Reintegrating AI*.

[Rumelhart and McClelland, 1986] Rumelhart, D. E. and McClelland, J. L. (1986). *Parallel distributed processing: explorations in the microstructure of cognition. Volume 1. Foundations*. MIT Press.

[Samsonovich et al., 2010] Samsonovich, A., Stocco, A., Albus, J., Grossberg, S., Rohrer, B., Chandrasekaran, B., Kurup, U., Arrabales, R., Sun, R., Goertzel, B., Ritter, F., Evertsz, R., Mueller, S., Shapiro, S. C., Hawkins, J., Noelle, D. C., Franklin, S., Wang, P., Brom, C., Cassimatis, N., Laird, J., and Thórisson, K. R. (2010). Feature-by-feature comparison of cognitive architectures. `http://members.cox.net/bica2009/cogarch/architectures.htm` [Last Accessed Wednesday, May 26, 2010].

[Santore, 2005] Santore, J. F. (2005). *Identifying Perceptually Indistinguishable Objects*. PhD thesis, University at Buffalo.

[Schlegel and Shapiro, 2013] Schlegel, D. R. and Shapiro, S. C. (2013). Concurrent reasoning with inference graphs. In *Proceedings of the Third International IJCAI Workshop on Graph Structures for Knowledge Representation and Reasoning (GKR 2013)*.

[Searle et al., 1980] Searle, J. R. et al. (1980). Minds, brains, and programs. *Behavioral and brain sciences*, 3(3):417–457.

[Shapiro, 1992] Shapiro, S. C. (1992). Artificial intelligence. In Shapiro, S. C., editor, *Encyclopedia of Artificial Intelligence*, pages 54–57. John Wiley & Sons, second edition.

[Shapiro, 1998] Shapiro, S. C. (1998). Embodied cassie. In *Cognitive Robotics: Papers from the 1998 AAAI Fall Symposium, Technical Report FS-98*, volume 2, pages 136–143.

[Shapiro et al., 2005] Shapiro, S. C., Anstey, J., Pape, D. E., Nayak, T. D., Kandefer, M., and Telhan, O. (2005). The Trial The Trail, Act 3: A virtual reality drama using intelligent agents. In Young, R. M. and Laird, J., editors, *Proceedings of the First Annual Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*, pages 157–158. AAAI Press, Menlo Park, CA.

[Shapiro and Bona, 2010] Shapiro, S. C. and Bona, J. P. (2010). The GLAIR cognitive architecture. *International Journal of Machine Consciousness*, 2(2):307–332.

[Shapiro and Ismail, 2001] Shapiro, S. C. and Ismail, H. O. (2001). Symbol-anchoring in cassie. In *Anchoring Symbols to Sensor Data in Single and Multiple Robot Systems: Papers from the 2001 AAAI Fall Symposium*, pages 2–8.

[Shapiro and Ismail, 2003] Shapiro, S. C. and Ismail, H. O. (2003). Anchoring in a grounded layered architecture with integrated reasoning. *Robotics and Autonomous Systems*, 43(2–3):97 – 108. ¡ce:title¿Perceptual Anchoring: Anchoring Symbols to Sensor Data in Single and Multiple Robot Systems¡/ce:title¿.

[Shapiro and Kandefer, 2005] Shapiro, S. C. and Kandefer, M. (2005). A sneps approach to the wumpus world agent or cassie meets the wumpus. In *IJCAI-05 Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC'05): Working Notes*, pages 96–103.

[Shapiro and The SNePS Implementation Group, 2007] Shapiro, S. C. and The SNePS Implementation Group (2007). *SNePS 2.7 User's Manual*. Department of Computer Science and Engineering, University at Buffalo, The State University of New York, Buffalo, NY. Available as `http://www.cse.buffalo.edu/sneps/Manuals/manual27.pdf`.

[Stein and Meredith, 1993] Stein, B. and Meredith, M. (1993). *The merging of the senses.* The MIT Press.

[Steinmetz et al., 1970] Steinmetz, G., Pryor, G. T., and Stone, H. (1970). Olfactory adaptation and recovery in man as measured by two psychophysical techniques. *Perception & Psychophysics*, 8(5):327–330.

[Sun, 2004] Sun, R. (2004). Desiderata for cognitive architectures. *Philosophical Psychology*, 17(3):341–373.

[Sun, 2007] Sun, R. (2007). The importance of cognitive architectures: An analysis based on CLARION. *Journal of Experimental and Theoretical Artificial Intelligence*, 19(2):159.

[Taatgen and Anderson, 2010] Taatgen, N. and Anderson, J. R. (2010). The past, present, and future of cognitive architectures. *Topics in Cognitive Science*, 2(4).

[The SNePS Research Group, 2010] The SNePS Research Group (2010). Sneps research group homepage. `http://www.cse.buffalo.edu/sneps/` [Last Accessed Wednesday, February 10, 2010].

[Thrun, 2002] Thrun, S. (2002). Probabilistic robotics. *Communications of the ACM*, 45(3):52–57.

[Turing, 1950] Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236):pp. 433–460.

[Wilson, 2002] Wilson, M. (2002). Six views of embodied cognition. *Psychonomic Bulletin & Review*, 9:625–636. 10.3758/BF03196322.

Appendix A

Implementation Details of MGLAIR Modalities and Mechanisms

A.1 MGLAIR in Common Lisp

This section shows the organization and details of my MGLAIR Lisp implementation. The structures and functions are defined in Lisp files in subdirectories of the `mglair` directory. Creating an MGLAIR agent involves first loading SNePS and then loading the agent-independent `pmla.cl`, and loading `mods.cl` from the agent's `pmb` file, which loads `mglair/mods.cl` and the rest of the system.

```
mglair
 ├─ modality
 │   ├─ modality-classes.cl
 │   ├─ modality-timing.cl
 │   ├─ modality-buffer.cl
 │   └─ modality-focus.cl
 ├─ mods.cl
 └─ patches
     └─ patch.cl
```

A user manual and downloadable examples are forthcoming.
These will be made available at http://www.cse.buffalo.edu/ jpbona/mglair/.

A.1.1 mods.cl

`mods.cl` loads the rest of the source files for defining and handling modalities. It also contains functions that aren't specific to focus, buffer management, or other pieces that reside in their own files.

```
;;; mods.cl - Implementation of MGLAIR modalities
;;; define-modality user function is defined here,
;;; as are a number of odds and ends dealing with modalities

(in-package :PMLb)

; Modalities Hash
(defparameter *modalitiesHash* (make-hash-table))
(defvar *warnings* t)

;;; see modality-classes, modality-focus, etc for specific mechanisms
(load "mglair/modality/modality-classes") ; defclass, methods for modality classes
(load "mglair/modality/modality-data-channel") ; defclass, methods for data channels
(load "mglair/modality/modality-focus") ; functions for modality focus
(load "mglair/modality/modality-buffer") ; classes, methods for modality buffers
(load "mglair/modality/modality-timing.cl") ; functions for modality buffer timing

(defun define-modality  (name type
                         &key frames access (focus '(nil . 0)) conflict
                         (description "a modality - insert a better description")
                         channel buffer sense-handler delay-running  (pmlc-confirm nil)
                         print)
  "Function to define a modality. Adds defined modality to *modalitiesHash*"
  ;; there's much room for improvement and simplification here
  (if (not (subtypep type 'modality))
      (error "In define-modality, ~A is not a type of modality~%" type)
      (let ((mod-inst (make-instance type :buffer
                                     (if (subtypep type 'efferent-modality)
                                         (make-instance 'efferent-buffer)
                                         (make-instance 'afferent-buffer)))))
        (if name (setf (name mod-inst) name)
            (error "define-modality needs a non-nil name"))
        (when frames
          (setf (frame mod-inst)
                (mapcar (lambda (x) (intern (symbol-name x) :snepslog)) frames)))
        (when (and (subtypep type 'efferent-modality)
                   (or pmlc-confirm (not (eq 'queueing conflict))))
          (setf (confirm-acting mod-inst) pmlc-confirm))
        (when access (setf (access mod-inst) access))
        ;;; e.g. ((capacity . 10) (exp-int 1))
        (when buffer
          (let ((b (make-instance (if (subtypep type 'efferent-modality)
                                      'efferent-buffer
                                      'afferent-buffer))))
            (when (assoc 'capacity buffer)
              (setf (capacity b) (rest (assoc 'capacity buffer))))
            (when (assoc 'exp-int buffer)
              (setf (exp-int b) (rest (assoc 'exp-int buffer))))
```

```
            (setf (buffer mod-inst) b)))
        (when focus (setf (focus-modifiable mod-inst) (first focus)
                          (focus mod-inst) (focus-num (rest focus))))
        (if (member conflict '(squeeze blocking queue suspend interrupt))
            (setf (conflict (buffer mod-inst)) conflict)
            (when *warnings* (warn "Possibly invalid conflict in define-modality: ~A") conflict))
        (when description (setf (description mod-inst) description))
        (when (and (subtypep type 'afferent-modality) sense-handler)
          (set-sense-handler mod-inst sense-handler))
        (when channel
          (if (or (eq 'socket (rest (assoc 'type channel)))
                  (and (null (assoc 'type channel))
                       (not (null (assoc 'port channel)))
                       (not (null (assoc 'host channel)))))
              (setf (channel mod-inst)
                    (make-instance 'modality-data-channel
                                   :socket (socket:make-socket
                                            :connect :passive
                                            :local-port (rest (assoc 'port channel)))))
              ;otherwise, it's not a supported type
              (error "Unsupported socket type: ~A~%" (rest (assoc 'type channel)))))
        (when print (print-modality mod-inst))
        (unless delay-running (run-modality mod-inst))
        (setf (gethash (name mod-inst) *modalitiesHash*) mod-inst))))

;;; Keeping track of whether this efferent modality is currently acting
(defmethod acting ((emod efferent-modality))
  (pprint "getting acting")
  (slot-value emod 'acting))

(defmethod (setf acting) (value (emod efferent-modality))
  (format t "setting acting to ~A: " value)
  (setf (slot-value emod 'acting) value))

(defun remove-acting (emod &key discard)
  "Used by run-act-monitor to watch for the confirmation of acts in efferent mods"
  (if discard
      (read-line (socket (channel emod))) ; also need to clear acting?
      (let ((msg (read-from-string (read-line (socket (channel emod))))))
(when *warnings* (warn "in remove-acting, got a confirmation: ~A" msg))
    (if (eql msg (first (acting emod))) ; if this is a confirmation for the act
(setf (acting emod) nil) ; clear the thing
; otherwise..... odd - the modality sent something we weren't expecting
(when *warnings* (warn "in remove-acting, not eq: ~A, ~A " msg (first (acting emod))))))))

(defmethod kill-act-monitor ((emod efferent-modality))
  "Kills the act monitor for a process"
  (when (act-monitor (emod ))
    (mp:process-kill (act-monitor emod))))

(defmethod run-act-monitor ((emod efferent-modality) a &key discard)
  ;;; going to monitor emod for a confirmation that a has been performed
```

```
  (setf (acting emod) a)
  (setf (act-monitor emod)
(mp:process-run-function
 (concatenate 'string (symbol-name (name emod)) "-act-monitor" (symbol-name (gensym)))
 (lambda ()
   (setf (gethash a *impulses*) nil) ; assume a has not been confirmed
   (remove-acting emod)
   (setf (gethash a *impulses*) a)))))

(defun clear-register (mod)
  "Clears an afferent or efferent modality's register"
  (setq (register mod) nil))

;;;;;;;;;;;;;;;;;;;;;;;;;; Modality utils ;;;;;;;;;;;;;;;;;;;;;;;;;
;;; use of print-object would be preferred here
(defmethod print-modality ((m modality))
  "Prints out a modality's properties"
  (format t "Modality named ~A~%" (name m))
  (format t "~t predicates: ~A~%" (frame m))
  (format t "~t agent conscious access?: ~A~%" (access m))
  (format t "~t adjustable focus?: ~A~%" (focus-modifiable m))
  (format t "~t initial focus: ~A~%" (focus m))
  (format t "~t conflict handling mode: ~A~%" (conflict (buffer  m)))
  (format t "~t description: \"~A\"~%" (description m))
  (format t "~t data channel: ~A~%" (channel m))
)

(defun list-modalities ()
  "Print a list of all the modalities"
  (loop for key being the hash-keys of *modalitiesHash*
     do (let ((mod (gethash key *modalitiesHash*)))
  (format t "modality ~A running on socket ~A is of type ~A~%"
  (name mod) (socket (channel mod)) (type-of mod)))))


(defun run-all-modalities ()
  "Run all defined modalities. Usually modalities run themselves on definition"
  (loop for key being the hash-keys of *modalitiesHash*
     do (let ((mod (gethash key *modalitiesHash*)))
  (pprint mod)
  (format t "Starting modality ~A on socket ~A~%" (name mod) (socket (channel mod)))
  (run-modality mod))))

(defmethod get-modality ((name symbol))
  "Given a symbol naming a modality, return the modality object"
  (gethash (intern (symbol-name name) :PMLb) *modalitiesHash*))
(defmethod get-modality ((mod modality)) mod)

(defmethod run-modality ((amod modality))
  "Starts the internal processes for an afferent modality"
  (run-dc-reader amod) ; start the data channel reader
  (run-sense-handler amod)) ; start the sense handler
```

```
(defmethod run-modality ((emod efferent-modality))
  "Run an efferent modality"
  (setf (dc-process emod)  ; data channel process
(mp:process-run-function
 (symbol-name (name emod) )
 (lambda ()
            (setf (socket (channel emod))
                  (socket::accept-connection (socket (channel emod)))))))
  (run-efferent-buffer emod)) ;run the efferent buffer processes


(defmethod set-sense-process ((amod afferent-modality) (fn function))
  "Kills the sense handler currently monitoring amod and runs fn in a new process"
  (mp:process-kill (pproc mod))
  (setf (pproc mod) (mp:process-run-function (symbol-name (name mod)) fn)))


(defun get-modalities-by-type (mtype)
  "Returns a list of modalities whose type is mtype"
  (loop for key being the hash-keys of *modalitiesHash*
       when (typep (gethash key *modalitiesHash*) mtype)
       collect (gethash key *modalitiesHash*)))

(defun act-modality (aname)
  "Given the symbol that names a sneps act node,
   return the modality with which it's associated"
  (loop for emod in (get-modalities-by-type 'efferent-modality)
       do
       (if (member aname (frame emod)
   :test (lambda (x y) (string= (symbol-name x) (symbol-name y))))
   (return-from act-modality emod))))

(defun primact-modality (pa-name)
  "Given the symbol that names a primitive action,
   get the modality with which it's associated"
  (let ((aname
 (sneps::node-na (gethash (intern (symbol-name pa-name) :snepslog)
  snip::*primitive-action-functions*))))
    (act-modality aname)))



(defmethod run-sense-handler ((amod afferent-modality))
  "Runs an afferent modality's sense handler"
  (setf (pproc amod)
        (mp:process-run-function
         (symbol-name (name amod)) (lambda () (funcall #'sense-handler-fn amod)))))

(defun sense-handler-fn (amod)
  ;;; runs a looping process that checks this modality's  buffer for incoming senses
  ;;; & manages expiration
```

```
  (cl-user::while t
   ;; grab the next structure from the modality buffer
   (let ((sensed (buffer-pop (buffer amod))))
     ;; get rid of data older than the deadline
     (unless  (or (null sensed)
                  (expired? sensed (exp-int (buffer amod))))
       ;; whether it's expired depends on the exp-interval for this modality
       ;; timestamp :(first sensed)
       (setf (register amod) (funcall (sense-handler amod)  sensed)))
     (sleep (sleep-inc amod)))))

(defmethod execute ((mname symbol) impulse); &rest args)
  "Execute impulse in the efferent modality named by mname"
  (let ((emod (get-modality mname)))
    (if emod (execute emod impulse)
        (when *warnings*
          (warn "Trying to execute in modality ~A failed... does it exist?" mname)))))


(defmethod execute ((mod efferent-modality) impulse)
  (if (eq (conflict (buffer mod)) 'queueing)
      (add-to-buffer (buffer mod) impulse) ; make sure add-to checks for room
      (data-channel-add mod impulse)))
```

A.1.2 modality/modality-classes.cl

```
;;; Modality Classes
(defclass modality ()
   ;; the modality's name -- used to uniquely identify it
  ((name :initform 'Modality :initarg :name :accessor name)
   ;; predicate/frame associated with the modality
   (frame  :accessor frame :initarg :frame :initform 'Sensed)
   ;; can the agent consciously access the modality? (t or nil)
   (access :accessor access :initarg :access :initform nil)
   ;; stores this modality's data channel process
   (dc-process :accessor dc-process)
   ;; modality percept/act buffer
   (buffer :initform
   (make-instance 'modality-buffer) :initarg :buffer :accessor buffer)
   ;; focus level - initially normal (0)
   (focus :initform 0 :accessor focus)
   ;; is focus modifiable? (t or nil)
   (focus-modifiable :initform nil :accessor focus-modifiable)
   ;; sleep-inc is used internally to manipulate focus
   ;; Not intended to be modified by the user.
   (sleep-inc :initform .1 :accessor sleep-inc)
   ;; run immediately if t
   (runmod :initarg t :accessor runmod)
   ;; Modality register stores KL term for last act/percept
   ;; Modified by execute-primaction, and by perceptual processes
   (register :initform nil :accessor register)
   ;; Human-readable description of the modality
   (description :initarg :description :initform "A modality" :accessor description)
   ;; Modality data channel
   (channel :initarg :channel :initform
    (make-instance 'modality-data-channel ) :accessor channel)))

(defclass afferent-modality (modality)
  ;; sense-handler implements the modality's perceptual function
  ;; The default is perhaps too simple to be used
  ((sense-handler :accessor sense-handler :initform (lambda (x) (snepslog::tell x)))
   ;; perceptual process runs sense handler, which calls PMLs::believe
   (pproc :accessor pproc)
   (buffer :initform
   (make-instance 'afferent-buffer) :initarg :buffer :accessor buffer)))

(defclass efferent-modality (modality)
  ((frame  :accessor frame :initarg :frame :initform 'Move)
   ; used to monitor whether the modality is currently in use at the PMLc
   (acting :accessor acting :initarg :acting :initform nil)
   (act-monitor :accessor act-monitor :initarg :act-monitor :initform nil)
   (confirm-acting :accessor confirm-acting :initarg :confirm :initform nil)
   ;; buffer management process
   (bproc :accessor bproc)
   (buffer :initform
   (make-instance 'efferent-buffer :capacity 0) :initarg :buffer :accessor buffer)))
```

```
;; accessor for sense-handler for afferent modalities
(defmethod sense-handler ((amod afferent-modality))
  "If the sense-handler is function, return it;
   if a symbol that names a function: compile, set, and return it;
   else error"
  (let ((shand (slot-value amod 'sense-handler)))
    (if (functionp shand)
shand
(if (and (symbolp shand) (fboundp shand))
    (setf (slot-value amod 'sense-handler)
  (symbol-function (compile shand (symbol-function shand))))
    (error "sense-handler \"~A\" for modality ~A not defined" shand (name amod))))))

(defmethod set-sense-handler ((m symbol) fn)
  "Sets the modality named m to have sense data handler fn"
  (let ((mod (get-modality m)))
    ;; if this symbol names a modality, set its sense handler to the given fn
    (if mod
(set-sense-handler mod fn)
(error (format nil "set-sense-handler got a modality name that doesn't exist: ~A" m)))))

(defmethod set-sense-handler ((m modality) fn)
  "Sets the modality m to have sense handler fn"
  (setf (slot-value m 'sense-handler)
(cond ((functionp fn) fn)   ;if fn's a function, store it
      ((fboundp fn) (compile fn (symbol-function fn))) ; if fn's a fbound symbol, store it
      ((symbolp fn)
       (progn
 (when *warnings*
   (warn "set-sense-handler called with a symbol that does not now name a function: ~A" fn))
 fn))))) ; otherwise, error
```

A.1.3 modality/modality-data-channel.cl

```
;;;;;;;;;;;;;;;;;;;;;;;;;; Modality data channel class and methods ;;;;;;;;;;;;;;;;;;;;;;
(defclass modality-data-channel ()
  ((socket :initform nil :initarg :socket :accessor socket)
  (channel-stream :initform nil :initarg :channel-stream :accessor channel-stream)))

(defmethod connectedp ((m modality))  (connectedp (channel m)))
(defmethod connectedp ((mname symbol)) (connectedp (gethash mname *modalitiesHash*)))
(defmethod connectedp ((c modality-data-channel))  (socket c))

(defmethod activep ((m modality)) (activep (channel m)))
(defmethod activep ((mname symbol)) (activep (gethash mname *modalitiesHash*)))
(defmethod activep ((c modality-data-channel))
  (typep (socket c) 'acl-socket:socket-stream-internet-active))

(defmethod attach-channel ((m modality) (c modality-data-channel))
   ;; given a modality data channel that is not connected, associate it with a modality
   (unless (connectedp (channel m))
   (setf (channel m) c)))

(defmethod data-channel-add ((emod efferent-modality) msg)
  (let ((tsmsg (cons (get-internal-real-time) msg)))
    (case (conflict (buffer emod))
      (blocking
       (if (and (confirm-acting emod) (acting emod))
   ;; if the modality's currently acting, discard
   (when *warnings* (warn "Modality ~A is busy, discarding impulse ~A" (name emod) msg))
   ;; otherwise, bypass the buffer and send
   (dc-send emod tsmsg)))
      (otherwise
       (if (acting emod)
   (progn
     (kill-act-monitor emod) ; kill the act monitor
     (dc-send emod (acting emod))
     (setf (acting emod) nil)) ; clear acting
   (dc-send emod tsmsg))))))

(defmethod dc-send ((emod efferent-modality) msg)
  "Send a message into emod's data channel"
  (when (confirm-acting emod)
    (run-act-monitor emod msg))
  (format (socket (channel emod)) "~A~%" msg)
  (force-output (socket (channel emod)))
  msg)

(defmethod run-dc-reader ((amod modality))
  ;;; the channel reader reads incoming data out of the data channel
  ;;; and adds it to this modality's buffer
  (setf (dc-process amod)
(mp:process-run-function
 (concatenate 'string (symbol-name (name amod) ) "-channel-reader")
```

```
(lambda ()
  (let ((sock (socket::accept-connection (socket (channel amod)))))
    (setf (socket (channel amod)) sock)
    (cl-user::while t
    (funcall #'add-to-buffer (buffer amod)
     (string-trim '(#\newline #\return #\null)
  (read-line sock))))))))
```

A.1.4 modality/modality-focus.cl

```
;;; define constants for the focus level
(defconstant lowest -3)
(defconstant low -2)
(defconstant reduced -1)
(defconstant normal 0)
(defconstant elevated 1)
(defconstant high 2)
(defconstant highest 3)

(defun focus-num (f)
  "Given a symbol naming a focus level, return the corresponding number"
  ;; for internal use only
  (if (numberp f) f
      (case f
( lowest -3)
( low -2)
( reduced -1)
( normal 0)
( elevated 1)
( high 2)
( highest 3)
( otherwise 0))))

;;; set the minimum sleep time to 10 millis (default is 75)
;;; (trying to sleep for less than the minimum sleep time breaks things)
(setf (sys::thread-control :clock-event-delta) 10)

;;; Used as the base time to determine how long each modality buffer process sleeps
;;; NOT modality-specific -- focus is adjusted through adjust-focus, which uses this
(defparameter *mod-sleep-base* .1)

(defun raise-focus (mod)
  (adjust-focus mod))

(defun lower-focus (mod)
  (adjust-focus mod :decrease? t))

(defmethod adjust-focus ((amod symbol) &key decrease?)
  "Increase the focus of the modality named by amod unless :decrease"
   (let ((modality (get-modality amod)))
     (when modality (adjust-focus modality :decrease? decrease?))))

(defmethod adjust-focus ((amod afferent-modality) &key decrease?)
  "Increase or decrease the modality amod's focus unless
   it's either highest or lowest respectively.
   Warns the user and returns the focus or nil if adjusting was not successful."
  (if (and (not decrease?) (= (focus amod) highest))
      (warn "Can't increase focus any more")
      (if (and decrease? (= (focus amod) lowest))
  (warn "Can't decrease focus any more")
```

```
  (let ((newfocus (funcall (if decrease? #'1- #'1+) (focus amod))))
    (progn
      (setf (focus amod) newfocus)
      (setf (sleep-inc amod) (focus-sleep newfocus))
      (focus amod))))))

(defmethod focus-sleep ((mod modality))
  "Given a modality, calculate the amount of time for it to sleep"
  (focus-sleep (focus mod)))
(defmethod focus-sleep ((f number))
  "Given a focus level, calculate the amount of seconds to sleep"
  (* *mod-sleep-base* (expt 2 (* -1 f))))

(defmethod ignore-modality ((amod symbol))
  "Ignore the modality named by the symbol amod"
  (let ((modality (get-modality amod)))
     (when modality (ignore-modality modality))))

(defmethod unignore-modality ((amod symbol) &key sense-handler)
  "Stop ignoring the modality named by the symbol amod"
  (let ((modality (get-modality amod)))
     (when modality (unignore-modality modality sense-handler))))

(defmethod ignore-modality ((amod afferent-modality))
  "Ignore the modality amod by kill its process,
  starting one that just reads off messages and ignores them"
   (set-modality-process amod
 (lambda () (cl-user::while t (read-line (socket (channel amod)))))))

(defmethod unignore-modality ((amod afferent-modality) &key sense-handler)
"Kills the process currently monitoring amod and runs the new sense-handler,
or the original one for the modality if no new one is specified"
(set-modality-process
 amod
 (lambda () (cl-user::while
     t
     (let ((percept (string-trim '(#\newline #\return #\null)
 (read-line (socket (channel amod)))))
       (funcall (if sense-handler sense-handler (sense-handler amod)) percept))))))
```

A.1.5 modality/modality-buffer.cl

```
(defclass modality-buffer ()
  ;; atomic enqueue, dequeue
  ((queue :initform (make-instance 'mp:queue) :accessor queue)
   ;; buffer expiration interval < 0 means none
   (exp-int :initform -1 :initarg :exp-int :accessor exp-int)
   ;; what happens when a buffer is full?
   (conflict :initform 'squeeze :initarg :conflict :accessor conflict)
   ;; capacity < 0 means unlimited
   (capacity :initform -1 :initarg :capacity :accessor capacity)))
;; future work: consider sized queues
;; http://www.franz.com/support/documentation/9.0/doc/multiprocessing.htm

(defclass afferent-buffer (modality-buffer) ())
(defclass efferent-buffer (modality-buffer)
   ((conflict :initform 'queueing)
    (capacity :initform 5 :accessor capacity :initarg :capacity)))


(defmethod clear-buffer ((emname symbol))
  (clear-buffer (get-modality emname)))
(defmethod clear-buffer ((emod efferent-modality))
  (cl-user::while (> (size buffer) 0) (buffer-pop emod)))
(defmethod clear-buffer ((amod afferent-modality))
  (warn "Clear-buffer got an afferent modality. This is not supported."))

;; not to be confused with capacity, gets the current amount of stuff in the buffer
(defmethod size ((buffer modality-buffer))
  "Get the amount of stuff currently in a buffer"
  (1- (length (mp::queue-head (queue buffer)))))

(defmethod vacancyp ((buffer modality-buffer))
  "Returns t if a modality has available space in its buffer"
  (or (< (capacity buffer) 0) (< (size buffer) (capacity buffer))))

(defmethod run-efferent-buffer ((emod efferent-modality))
  "Runs the buffer handling process for efferent modalities,
   and store it in the modality's process slot"
  (setf (bproc emod)
        (mp:process-run-function
         (symbol-name (name emod))
         (lambda () (funcall #'efferent-buffer-handler-fn emod)))))

(defun efferent-buffer-handler-fn (emod)
  "Efferent buffer handler gets (unexpired) stuff out of the buffer
   and adds it to the data channel"
  (cl-user::while
   t (let ((nextact (buffer-pop (buffer emod))))
       (unless (or (null nextact)
                   (expired? nextact (exp-int (buffer emod))))
         (dc-send emod nextact))
```

```
      (sleep (sleep-inc emod)))))

(defun clear-efferent-buffer (emod)
    (cl-user::while (> (size (buffer emod)) 0)
      (when *warnings*
          (warn (format nil "clearing buffer, threw away: ~A"
                              (buffer-pop (buffer emod)))))))

(defun expired? (sensed exp-interval)
  "True if a sensory datum or act impulse is expired"
  (and (> exp-interval 0)
    (> (- (get-internal-real-time) (first sensed)) exp-interval)))

(defmethod add-to-buffer ((buffer afferent-buffer) item)
  (let ((impulse (cons (get-internal-real-time) item))) ;; add timestamp
    (case (conflict buffer)  ; adding to buffer depends on conflict-handling
      ('squeeze
       (unless (vacancyp buffer) ; unless there's space, throw away the first thing
         (buffer-pop buffer))
       ;; then add the new structure with timestamp
       (mp::enqueue (queue buffer)  impulse))
       (otherwise  ; for afferent, the other option is blocking
        (if (vacancyp buffer)
            (mp::enqueue (queue buffer) impulse)
            (warn (format nil "add-to-buffer blocking new item ~A~%"
                              (not (vacancyp buffer)))))))
    impulse))

(defmethod add-to-buffer ((buffer efferent-buffer) item)
  (let ((impulse (cons (get-internal-real-time) item))) ;; add timestamp
    (if (vacancyp buffer) ;room
       (mp::enqueue (queue buffer)  impulse)
       ;; no room
       (if (eq (conflict buffer) 'squeeze)
   (progn (buffer-pop buffer)
  (mp::enqueue (queue buffer)  impulse))
   (warn (format nil "add-to-buffer blocking new item ~A~%"
                             (not (vacancyp buffer))))))
    impulse))

(defmethod buffer-pop ((b modality-buffer))
  "Remove and return the top item from the modality buffer b"
  (unless (mp::queue-empty-p (queue b))
        (mp::dequeue (queue b))))

;;; used for timing calculations
(defmethod buffer-status ((m modality))
  (format nil "~A buffer status: ~%~TCapacity: ~A~%~TSize: ~A~%~%"
          (name m)
          (if (>= (capacity (buffer m)) 0) (capacity (buffer m)) "unlimited")
          (size (buffer m))))
```

```
(defun print-all-buffer-stats ()
  "Prints the buffer status for all modalities"
  (loop for key being the hash-keys of *modalitiesHash*
     do (pprint (buffer-status (gethash key *modalitiesHash*)))))
```

A.2 Python-based PMLc Implementation

This section shows the agent-independent part of the implementation of a Python-based PMLc. Its use is illustrated in the `NXTBot` PMLc, the implementation of which is given in Appendix B.

A.2.1 `PMLc.py`

```
import threading
import socket
import time

from MGLAIR.agentconfig.ConfigReaders import AgentConfigReader

class MGLAIRActorI:

    def handleImpulse(self, i):
        raise 'override handleImpulse to get stuff done'

class ModalityConnector:

    def __init__(self, modConfig, mglairActor):

        self.modConfig = modConfig
        self.bot = mglairActor
        self.name = modConfig.getName()
        self.port = modConfig.getDConfig().getPort()
        self.hostname = modConfig.getDConfig().getHostname()

    def confirmImpulse(self, i):
        self.setSense(str(i))

    def processImpulse(self, text):
        print "Calling handleImpulse with", text
        self.bot.handleImpulseM(self.getName(),text);
        #self.bot.handleImpulse(text);

    def setSense(self, s):
        # write to socket
        self.sock.send(s.strip()+"\n")

    def getName(self):
        return self.name

    def connect(self):
        print "Connecting modality at port ", self.port

        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.connect((self.hostname, self.port))

        self.watcher = SocketWatcher(self.sock, self)
        self.watcher.start()
```

```
class SocketWatcher(threading.Thread):

    def run(self):
        self.listen();

    def __init__(self, socket, connector):
        self.socket = socket;
        self.connector = connector;
        threading.Thread.__init__(self)

    def getSocket(self):
        return self.socket;

    def listen(self):
        # listen on the data channel socket
        print "listening " + " to modality " + self.connector.getName()
        line = str(self.socket.recv(1024))
        while(len(line) > 0):

            lines = line.split("\n");
            print lines
            for l in lines:
                if len(l.strip()) > 0:
                    self.connector.processImpulse(l)

            time.sleep(.00001)
            line = str(self.socket.recv(1024));

class ModalityManager():

    def __init__(self, configFN, actor):
        self.actor = actor
        self.configFN = configFN
        self.connectors = {}

    def connectModalities(self, *mnames):
        mcs = AgentConfigReader.getConfig(self.configFN)
        for mname in mnames:
            try:
                mcon = ModalityConnector(mcs[mname],self.actor)
                mcon.connect()
                self.connectors[mname] = mcon
            except KeyError:
                print "KeyError: can't find modality ", mname

    def getMCByName(self, name):
        return self.connectors[name]
```

A.2.2 ConfigReaders.py

```
from xml.dom import minidom

class DataChannelConfig:

    def __str__(self):
        return "Data Channel\n" + "Port: " + str(self.port) + "\n";

    def __init__(self, port, hostname = None):
        self.port = port;
        self.hostname = hostname;

    def getPort(self):
        return self.port;

    def setPort(self, port):
        self.port = port;

    def getHostname(self):
        return self.hostname

    def setHostname(self,hostname):
        self.hostname = hostname

class ModalityConfig:

    def __str__(self):
        return "Modality:\n" + "Name: " + self.name + "\n" + self.dcconfig.__str__()

    def __init__(self, name, dcconfig, hostname='localhost', frame='', type='',confirm=False):
        self.name = name;
        self.frame = frame;
        self.type = type;
        self.dcconfig = dcconfig;
        self.confirm = confirm

    def getConfirm(self):
        return self.confirm;
    def setConfirm(self,c):
        self.confirm = c

    def getName(self):
        return self.name;

    def setName(self, name):
        self.name = name;

    def getDConfig(self):
        return self.dcconfig;

    def setDConfig(self, dcc):
```

```
        self.dcconfig = dcc;

class AgentConfigReader():

    @staticmethod
    def getConfig(cfile):
        modalities = {}
        config = minidom.parse(cfile)
        aconfigs = config.getElementsByTagName('agent')
        if aconfigs.length > 0:
            if aconfigs.length > 1:
                print "Ignoring all but the first agent config"

            mconfigs = aconfigs[0].getElementsByTagName('modality')

            for mconfig in mconfigs:
                mc = ModalityConfigReader.getConfig(mconfig)
                mc.getDConfig().setHostname(aconfigs[0].attributes['address'].value)
                modalities[mc.name] = mc

        return modalities

class ModalityConfigReader():

    @staticmethod
    def getConfig(mconfig):
        name = mconfig.attributes['name'].value
        confirming  = False;
        try:
            confirming = mconfig.attributes['confirm-acting'].value
            print "yes confirm-acting for ", name, " it's ", confirming
        except KeyError:
            print "no confirm-acting for ", name

        dchans = mconfig.getElementsByTagName('data-channel')
        if dchans.length > 0:
            chan = dchans[0]
            port = int(chan.attributes['port'].value)
            type = chan.attributes['type'].value

            dc = DataChannelConfig(port)
            mc = ModalityConfig(name,dc,confirm=confirming)
            print mc
            return mc
```

A.3 Java PMLc Implementation

A.3.1 ModalityConnector

```
/**
 * Class responsible for connecting to a single modality's data channel.
 * A loop watches for input from the data channel from afferent modalities.
 */
public class ModalityConnector{

  Socket outSock = null;
  PrintWriter outWriter = null;
  String hostname = "localhost";
  int port = 9674;

  SocketWatcher sw;

  private   String name = "";
  private String type = "";
  private boolean busy = false;
  private boolean confirmActing = false;

  private PMLcI pmlc;

  public ModalityConnector(ModalityConfig mconf, PMLcI c){
    this.pmlc = c;
    this.name = mconf.getName();
    this.port = mconf.getDc().getPort();
    this.type = mconf.getType();
    this.confirmActing  = mconf.isConfirmActing();
  }

  public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

  public int getPort() {
        return port;
    }
```

```
  public void setPort(int port) {
      this.port = port;
  }

public void setSense(String input){
  outWriter.println(input);
}


public void processImpulse(String text){
  pmlc.handleImpulse(this.name, text);
}

public boolean confirmImpulse(String i){
  if(confirmActing){
      setSense(i);
      return true;
  }
  return false;
}

public boolean connect(){
  System.out.println("Connecting modality DC at port " + port);
  try {
    outSock = new Socket(hostname, port);
    outWriter = new PrintWriter(outSock.getOutputStream(), true);
    System.out.println("connected");
  } catch (UnknownHostException e) {
    System.err.println("Unknown host: " +hostname);

  } catch (IOException e) {
    System.err.println("IOException connecting to: " + hostname);

  }
  sw = new SocketWatcher(outSock, this);
  sw.start();
  return true;
}

public boolean sleepwatcher(){
  return true;
}

class SocketWatcher extends Thread{
  BufferedReader breader;

  private Socket s;
  private ModalityConnector mc;
  SocketWatcher(Socket sock, ModalityConnector mc){
    this.mc = mc;
    s = sock;
    try {
```

```
         breader = new BufferedReader(new InputStreamReader(s.getInputStream()));
      } catch (IOException e) {
         e.printStackTrace();
      }
    }

    @Override
    public void run() {
      while(true){
        try {
            String line = breader.readLine();
            busy = true;
            System.err.println("about to process impulse: " + line);
            mc.processImpulse(line);
            busy = false;
        } catch (IOException e) {
            e.printStackTrace();
        }
        Thread.yield();
      }
    }
  }
}
```

A.3.2 ModalityManager

```
/**
 * Class for managing connections to modality data channels.
 * Maintains a reference to a PMLcI instance that is agent specific.
 * Also takes as an argument to its constructor a String that
 * names a file in which modality definitions are stored.
 */
class ModalityManager{

  private PMLcI pmlc;
  private String configFN;
  private AgentConfigReaderI acfgReader;
  private java.util.HashMap<String, ModalityConnector> modConns =
                                  new HashMap<String, ModalityConnector>();

  Map<String, ModalityConfig> mcs = new HashMap<String,ModalityConfig>();

  public void getConfig(){
    mcs = acfgReader.getConfig(configFN);

    for(String mname : mcs.keySet()){
      ModalityConnector mcon = new ModalityConnector(mcs.get(mname), pmlc);
      modConns.put(mcon.getName(),mcon);
      System.out.println("Added a modalityconnector " + mcon.getName());
    }
  }

  /**
   * Connects all the modalities named in mnames
   * @param mnames a Collection of strings naming modalities to be connected
   */
  public void connectModalities(String... mnames){
    getConfig();
    for(String mname : mnames){
      modConns.get(mname).connect();
    }
  }

  public ModalityManager(String infile, PMLc pmlc){
    this(infile,pmlc,new AgentConfigReader());
  }

  public ModalityManager(String infile, PMLcI actor, AgentConfigReaderI aconfig){
    this.acfgReader = aconfig;
    this.pmlc = actor;
    this.configFN = infile;
  }

  public ModalityConnector getMCByName(String mname){
    return modConns.get(mname);
  }
```

```
}
```

A.3.3 Modality Configuration Readers

```
interface AgentConfigReaderI{
  public Map<String,ModalityConfig> getConfig(String cfile);
}

class AgentConfigReader implements AgentConfigReaderI{

  /**
   * Create a Map from names of modalities to configuration objects
   */
  public Map<String,ModalityConfig> getConfig(String cfile){

    HashMap<String,ModalityConfig> mcs = new HashMap<String, ModalityConfig>();
      try {

      File fXmlFile = new File(cfile);
      DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
      DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
      Document doc = dBuilder.parse(fXmlFile);
      doc.getDocumentElement().normalize();

      // if this is, in fact, an agent specification
      if(doc.getDocumentElement().getNodeName().equalsIgnoreCase("agent")){
        // get all the modalities
        NodeList nList = doc.getElementsByTagName("modality");
        for (int temp = 0; temp < nList.getLength(); temp++) {
          ModalityConfig mc = ModalityConfigReader.getConfig(nList.item(temp));
          mcs.put(mc.getName(), mc);
        }
      }
      }catch (Exception e) {
        e.printStackTrace();
      }
      return mcs;
  }
}

class ModalityConfigReader{

  public static ModalityConfig getConfig(Node node){
     if (node.getNodeType() == Node.ELEMENT_NODE) {

        Element eElement = (Element) node;

        String name =  eElement.getAttribute("name");
        String confirm = eElement.getAttribute("confirm-acting");
        boolean ce = Boolean.parseBoolean(confirm);

        DataChannelConfig dc = new DataChannelConfig();
```

```
        NodeList channels = eElement.getElementsByTagName("data-channel");

        if(channels.getLength() > 0){
          Element cElt = (Element) channels.item(0);

          String channelType = cElt.getAttribute("type");
          String channelPort = cElt.getAttribute("port");
          dc.setPort(Integer.parseInt(channelPort));
        }
        ModalityConfig mconf = new ModalityConfig(name, dc);
        mconf.setConfirmActing(ce);
        return mconf;
      }
      System.out.println("ModalityConfigReader.getConfig() returning null");
      return null;
  }
}
```

Appendix B

Implementation Details for Python-NXT Agent

B.1 Knowledge Layer

```
;;; NXTAgent.snepslog
;;; KL for simple agent embodied in NXT-based robot with
;;; locomotive, locomotive-proprioception, grasping effector,
;;; and ultrasonic range-sensing modalities
set-mode-3

;; doing nothing
define-frame nothing(action)
;; sleep for a short while
define-frame sleep(action).

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Primitive acts
;;; grab() : the act of grabbing with the grabbing effector
define-frame grab(action).
;;; release() : the act of releasing with the grabbing effector
define-frame release(action).
;;; turn(d) : the act of turning in direction d (left or right)
define-frame turn(action d)
;;; go(godir) : the act of going in direction godir (forward or backward)
define-frame go(action godir)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Proposition-valued Functions
;;; [[Ball(b)]] = [[b]] is a ball
define-frame Ball(nil b).
;;; [[Object(o)]] = [[o]] is an object
define-frame Object(nil o).
;;; [[SizeOf(o s)]] = The size of object [[o]] is [[s]]
define-frame SizeOf(nil o s).
;;; [[ColorOf(o c)]] = The color of object [[o]] is [[cs]]
define-frame ColorOf(nil o c).
```

```
;;; [[LocatedAt(o l)]] = The object [[o]] is located at [[l]]
define-frame LocatedAt(nil o l).
;;; [[DistIs(dist)]] = [[dist]] is a distance
define-frame DistIs (nil dist).
;;; [[Moving(dir)]] = The agent is moving in direction [[dir]]
define-frame Moving (nil dir).
;;; [[MoveDir(md)]] = The [[md]] is a direction in which the agent can move
define-frame MoveDir(nil md).
;;; [[Feeel(bump)]] = The agent feels [[bump]]
define-frame Feel (nil bump)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Frames for the non-primitive locomotive acts
;;; (these use proprioception/range-sensing to monitor progress)
;;; There are probably better names to distinguish them from the primitives the resemble

;;; Going forward, backward, etc while monitoring locomotive-proprioception
define-frame goForward(action)
define-frame goBackward(action)
define-frame turnLeft(action).
define-frame turnRight(action).

;;; toTo(object) : high-level action for centering on and approaching
define-frame goTo(action object).
;;; approach(object) : action for approaching a centered object
define-frame approach(action object).
;;; alignWith(object) : align with an object by turning until it's centered
define-frame alignWith(action alignobj).
;;; approachObstacle() : move towards an obstacle until it's close
define-frame approachObstacle(action).
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

^^
(cl:load "./pmla.cl")
(attach-primaction
 adopt adopt unadopt unadopt
 believe believe disbelieve disbelieve
 do-all do-all do-one do-one
 snif snif sniterate sniterate
 snsequence snsequence
 withall withall withsome withsome
 senseFor senseFor
 go go-act
 turn turn-act
 nothing noop
 sleep sleepasec
 start start
 stop stop
 say say
 grab grab-act
 release release-act
)
```

```
^^

;;; Turn tracing on/off
untrace inference acting

;;; Can only be moving in one direction at a time
andor(0,0){Moving(forward), Moving(left), Moving(right), Moving(backward)}.
;;; The closest obstacle is at only one distance [at a time]
andor(0,1){DistIs(close), DistIs(medium), DistIs(far), DistIs(distant)}.

;; a ball has one size, location, and color
all(x)(Ball(x) =>
         andor(0,1){SizeOf(x,small),SizeOf(x,medium),SizeOf(x,large)})!
all(x)(Ball(x) =>
         andor(0,1){LocatedAt(x,center),LocatedAt(x,right),LocatedAt(x,left)})!
all(x)(Ball(x) =>
         andor(0,1){ColorOf(x,red), ColorOf(x,blue), ColorOf(x,unknown)})!

Ball(ball)!
MoveDir({forward, backward, left, right}).
nexists(_,1,_)(d)(MoveDir(d): Moving(d)).

;; Approaching, e.g. a wall - move forward until it's close
ActPlan(approachObstacle(),
        snif({if(~DistIs(close),snsequence(goForward(),approachObstacle()))})).

;; Moving and turning informed by proprioception
ActPlan(goForward(),snif({if(~Moving(forward), go(forward))})).
ActPlan(turnLeft(), snif({if(~Moving(left), turn(left))})).
ActPlan(turnRight(), snif({if(~Moving(right), turn(right))})).


;; To go to the ball: if it's in the center of your vision, approach it
;; Otherwise, align the center of your vision and then go to the ball
ActPlan(goTo(ball), snif({if(LocatedAt(ball,center), approach(ball)),
                             else(snsequence(alignWith(ball), goTo(ball)))})).

;; To approach the ball, keep moving toward it unless
;; it appears large or you've felt a bump
;; When you feel the ball, grab it
ActPlan(approach(ball),
        snif({if(~SizeOf(ball,large),
               snsequence(goForward(),
                        snsequence(adopt(whendo(Feel(bump), grab())),
                                   approach(ball))))})).

;; To align with the ball:
;; turn right while it's to your right, left while it's to your left
ActPlan(alignWith(ball),
        snif({if(LocatedAt(ball,right), snsequence(turnRight(), alignWith(ball))),
              if(LocatedAt(ball,left), snsequence(turnLeft(), alignWith(ball)))})).
```

B.2 Modality Definitions

```
(define-modality 'vision
   'afferent-modality
  :frames '(DistIs)
  :focus '(nil . 1)
  :buffer '((capacity . 1))
  :channel '((port . 9574) (host . localhost))
  :sense-handler 'sense-distance
  :description "Range sensing modality used by the agent to detect what's in front of it"
  :print t)

(define-modality 'vision2
   'afferent-modality
  :frames '(BallIs SizeOf LocatedAt)
  :focus '(nil . 1)
  :channel '((port . 9573) (host . localhost))
  :buffer '((capacity . 5))
  :conflict 'squeeze
  :sense-handler 'sense-vision
  :description "Visual modality used by the agent to detect objects"
  :print t)

(define-modality  'locomotion
   'efferent-modality
  :frames '(turn goForward go)
  :description "Used by the agent to move and turn"
  :channel '((port . 9576) (host . localhost))
  :conflict 'blocking
  :pmlc-confirm t
  :print t)

(define-modality 'locomotive-proprioception
   'afferent-modality
  :frames '(Moving)
  :channel '((port . 9579) (host . localhost))
  :buffer '((capacity . 1))
  :focus '(nil . 0)
  :sense-handler 'sense-motion
  :description "motor feedback"
  :print t)

(define-modality 'grabber
   'efferent-modality
  :frames '(grab release)
  :channel '((port . 9577) (host . localhost))
  :focus '(nil . 0)
  :description "Used to manipulate the agent's grasping effector"
  :print t)

(define-modality 'tactile
   'afferent-modality
```

```
:frames '(Feel)
:channel '((port . 9575) (host . localhost))
:focus '(nil . 0)
:buffer '((capacity . 1))
:sense-handler 'sense-touch
:description "used to feel bumps, etc"
:print t)
```

B.3 PMLa

```
;;; NXT agent pmla
(in-package :snepslog)
(cl:load "./pmlb.cl")        ; load agent pmlb
(cl:load "./mglair/pmla.cl") ; load mglair pmla functions

(define-primaction grab-act ()
 (sneps:clear-infer)
 (PMLb:grab 'grabber t))

(define-primaction release-act ()
 (sneps:clear-infer)
 (PMLb:grab 'grabber nil))

(define-primaction go-act ((godir))
  (cond ((eq godir (sneps:node 'forward))
         (PMLb:forward 'locomotion))
        ((eq godir (sneps:node 'backward))
         (PMLb:backward 'locomotion))
        (otherwise (format sneps:outunit "~&***Invalid go direction: ~S~%" godir)))
  (mp::sleep .1))

(define-primaction turn-act ((d))
  (sneps:clear-infer)
  (cond ((eq d (sneps:node 'left))
         (PMLb:turnLeft 'locomotion))
        ((eq d (sneps:node 'right))
         (PMLb:turnRight 'locomotion))
        ((eq d (sneps:node 'random))
         (if (> 1 (random 2))
             (PMLb:turnLeft)
             (PMLb:turnRight)))
        (t (format sneps:outunit "~&***Invalid turn direction: ~S~%" d)))
  (mp::sleep .1))


(define-primaction sleepasec ()
  (mp:process-sleep .1))

(define-primaction noop ())
```

B.4 PMLb

```
;;; PMLb for NXT agent
(defpackage :PMLb
  (:shadow common-lisp:find)
  (:export #:perform
           #:senseBall?
           #:forward #:backward #:pickUp #:putDown
           #:turnLeft #:turnRight #:exit
           #:senseBall
           #:grab #:release
           #:senseDistance))

(in-package :PMLb)
(cl:load "./pmls.cl")  ; load pmls
(cl:load "./mglair/mods.cl") ; load modality mechanisms


;;; PMLb actions
(defun turnLeft (mod) (execute mod (make-tuple-str `(t left))))
(defun turnRight (mod) (execute mod (make-tuple-str `(t right))))
(defun grab (mod &optional close?)
  (execute mod (make-tuple-str `(g ,(if close? "True" "False")))))

(defun forward (mod &optional (units 1))
  (execute 'locomotion (make-tuple-str `(f ,units))))
(defun backward (mod &optional (units 1))
  (execute 'locomotion (make-tuple-str `(b ,units))))


;;; PMLb sensing
(defun sense-touch (input)
  (PMLs::perceive-touch  (rest input) ))
(defun sense-motion (input)
  (PMLs::perceive-motion (rest input)))
(defun sense-distance(d)
  (PMLs::perceive-distance (rest (read-from-string (rest d)))))
(defun sense-vision(d)
  (PMLs::perceive-vision (read-from-string (rest d))))


;;; Load modality definitions
(load "modality-defns.cl")
```

B.5 PMLs

```
;;; PMLs - PMLa sensory sublayer
(defpackage :PMLs
  (:export #:see #:hear #:distance))

(in-package :PMLs)

(load "mglair/pmls.cl") ; load agent-independent mglair PMLs

(defun interpret-distance (dist)
  (symbol-name (cond ((< dist 30) 'close)
        ((< dist 60) 'medium)
        ((< dist 120) 'far)
        (t 'distant))))

(defun perceive-distance (d)
  "takes an int between 0 and 255, asserts the distance (close, far, etc)"
  (believe "DistIs(~A)" (interpret-distance d)))

(defun perceive-motion (motion)
  (format t "perceiving motion: ~A" motion)
  (cond
    ((or (null motion) (string= motion "False"))
      (snepslog::tell
        "perform withall(?x, Moving(?x), believe(andor(0,0){Moving(?x)}), nothing())."))
    (t (snepslog::tell (format nil "perform believe(Moving(~A))." motion)))))

(defun perceive-touch (touching)
  (if (string= touching "True")
      (believe "Feel(bump)")
      (believe "andor(0,0){Feel(bump)}")))

(defun perceive-vision (visuals)
  (let ((loc (interpret-location (rest (first visuals))))
        (size (interpret-size (rest (second visuals))))
        (color (interpret-color (rest (third visuals))))
        (obj "ball"))

(defun interpret-size (coverage)
  ;; takes values corresponding to the % of the fov the object uses 0...100
 (if (> 5 coverage) 'small
     (if (> 15 coverage) 'medium
         'large)))

(defun interpret-location (xypair)
  (if  (isabout 0 (first xypair) .1) 'center
       (if (> (first xypair) 0) 'right
       'left)))

(defun interpret-color (hsv)
  ;; takes HSV values in (0..180,0..255, 0..255)
```

```
  (if (isabout 120 (first hsv)) 'blue
      (if (isabout 0 (first hsv)) 'red
          'unknown)))

(defun isabout (target value &optional (e 5))
  (and (< value (+ target e)) (> value (- target e))))
```

B.6 Python PMLc and SAL

B.6.1 NXTBot Class

The NXTBot agent uses the agent-independent PMLc code shown in Appendix A. Act impulses from efferent modalities make their way instances of this class because the `ModalityManager` has a reference to it, and calls `handleImpulseM` on arriving impulses.

```
import nxt.locator, thread
import threading
from nxt.motor import *
from nxt.sensor import *
from math import fabs
import ast
from sys import stdout
from MGLAIR.PML.PMLc import MGLAIRActorI

class NXTBot(MGLAIRActorI):

    def setModalityManager(self,mmgr):
        self.modmgr = mmgr

    def confirmAction(self,modname,c):
        mc = self.modmgr.getMCByName(modname);
        mc.setSense(c)

    def senseMotion(self, moving):
        if self.modmgr:
            self.modmgr.getMCByName('locomotive-proprioception').setSense(str(moving))
        else:
            print "proprioception modality not connected"
        print "sensed motion ", moving

    def senseTouch(self, touchp):
        if self.modmgr:
            self.modmgr.getMCByName('tactile').setSense(str(touchp))
        else:
            print "tactile modality not connected"
        print "sensed touch ", touchp

    def senseDistance(self, dist):
        if self.modmgr:
            self.modmgr.getMCByName('vision').setSense("(d . " + str(dist) + ")")
        else:
            print "distance sensing modality not connected"

    def timestamp(self,str):
        print "str", str
        print "str[0]",str.split(str)[0]
        along = str.split(".")[0].strip(" (")
        print "along", along
        return long(along)
```

```
def iToTup(self,str):
    tupstr = str.split(".")[1]
    return ast.literal_eval(tupstr.strip(" ")[:len(tupstr)-2]);


def confirmLoc(self,time):
    return lambda : self.modmgr.getMCByName("locomotion").confirmImpulse(time)

def handleLocomotion(self,tup,time=-1):
    print "got time"
    print time

    if(tup[0] == "t"):
        if(tup[1] ==  "left"):
            self.turnLeft(cfn=self.confirmLoc(time))
        elif(tup[1] == "right"):
            self.turnRight(cfn=self.confirmLoc(time))
    elif(tup[0] == "f"):
        self.forward(cfn=self.confirmLoc(time))
    elif(tup[0] == "b"):
        self.backward(cfn=self.confirmLoc(time))
    else:
        print "got a bad locomotion impulse: " + tup

def handleImpulseM(self, mod, i):
    ts = self.timestamp(i)
    if mod == 'locomotion':
        tup = self.iToTup(i)
        self.handleLocomotion(tup,time=ts)
    elif mod == 'grabber':
        # here, open or close the grabber
        tup = self.iToTup(i)
        print tup
        if(tup[0] == "g"):
            self.moveGrabber(ast.literal_eval(tup[1]))
            return
    else:
        print mod + "is not any of the modalities handleImpulseM expected"
        self.handleImpulse(i);

def __init__(self, mmgr=None):

    self.modmgr = mmgr
    self.b = nxt.locator.find_one_brick()

    self.mx = nxt.Motor(self.b, nxt.PORT_C)
    self.my = nxt.Motor(self.b, nxt.PORT_B)
    self.motors = [self.mx, self.my]

    self.instructions = (
            [0, 0, 20, 360],
```

```
        [0, 1, 20, 360])

    #how long from start until the last instruction is ended
    self.length = 4

    self.us = UltraSonicSensor(self)
    self.us.start()

    self.touch = TouchSensor(self)
    self.touch.start()

def turnmotor(self, m, power, degrees):
    try:
        m.turn(power, degrees)
    except BlockedException:
        print "BlockedException in turnmotor."

def runinstruction(self, i):
    motorid, speed, degrees = i
    # start a thread to run the motor command
    thread.start_new_thread(
            self.turnmotor,
            (self.motors[motorid], speed, degrees))

def turnRight(self,cfn=None):
    tc = TachCheck(self,"right",inc=20,confirmfn=cfn)
    tc.start()
    for i in ([0, 0, 10, 20], [0, 1, -10, 20]):
                self.runinstruction(i[1:])

def turnLeft(self,cfn=None):
    tc = TachCheck(self,"left",inc=-20, confirmfn=cfn)
    tc.start()
    for i in ([0, 1, 10, 20], [0, 0, -10, 20]):
                self.runinstruction(i[1:])

def getTouch(self):
    return Touch(self.b,PORT_1).get_sample()

def getDistance(self):
    try:
        return Ultrasonic(self.b, PORT_4).get_sample()
    except nxt.error.I2CError:
        print "getDistance had a I2CError"

def moveGrabber(self, close):
    print "movegrabber: ", close
    val = -10 if close else 10
    print "val: " , val
    Motor(self.b, PORT_A).turn(-10 if close else 10, 25)

def openGrabber(self):
```

```
        Motor(self.b, PORT_A).turn(10, 25)

    def closeGrabber(self):
        Motor(self.b, PORT_A).turn(-10, 25)

    def getTacho(self):
        return nxt.Motor(self.b, nxt.PORT_C).get_tacho()

    def backward(self,cfn=None):
        self.instructions = (
            [0, 0, -20, 360],
            [0, 1, -20, 360] )

        tc = TachCheck(self,"backward",confirmfn=cfn)
        tc.start()

        seconds = 0
        while 1:
            print "Tick %d" % seconds
            for i in self.instructions:
                    if i[0] == seconds:
                            self.runinstruction(i[1:])
            seconds = seconds + 1
            break
            time.sleep(1)

        tc.stop()

    def forward(self,cfn=None):
        # proprioception
        tc = TachCheck(self,"forward",confirmfn=cfn)
        tc.start()

        seconds = 0
        while 1:
            print self.instructions # ([0, 0, 20, 360], [0, 1, 20, 360])
            print "Tick %d" % seconds
            for i in self.instructions:
                    if i[0] == seconds:
                            self.runinstruction(i[1:])
                            print i[1:][2]
            seconds = seconds + 1
            break
            time.sleep(1)
        tc.stop()
```

B.6.2 NXT Sensor & Motor Classes

```
# runs a thread that polls the touch sensor
class TouchSensor(threading.Thread):
    def run(self):
```

```
        self.listen()

    def __init__(self, bot):
        self.bot = bot
        threading.Thread.__init__(self)

    def listen(self):
        last = False
        while True:
            touchp = self.bot.getTouch()
            if last != touchp:
                last = touchp
                self.bot.senseTouch(touchp)

            time.sleep(.01)

# Runs a thread that polls the ultrasonic sensor
class UltraSonicSensor(threading.Thread):
    def run(self):
        self.listen()

    def __init__(self, bot):
        self.bot = bot
        threading.Thread.__init__(self)

    def listen(self):
        last = 0
        while True:
            dist = self.bot.getDistance()
            if dist != last:
                last = dist
                self.bot.senseDistance(dist)
            time.sleep(.01)

# runs a thread to check the tachomoter,
# compare to target value; used for
# "proprioceptive" sense
class TachCheck(threading.Thread):

    def run(self):
        self.listen()

    def __init__(self, bot, direction="forward", inc=360,confirmfn=None,
                                        motorid=1, threshold=.1,timeout=1000):
        self.inc = inc
        self.bot = bot
        self.direction = direction
        self.threshold = threshold
        self.timeout = timeout
        self.target = self.bot.getTacho().get_target(inc,1)
        self.stopped = False
        self.confirmfn = confirmfn
```

```
        threading.Thread.__init__(self)

    def millis(self):
        return int(round(time.time()*1000))

    def listen(self):
        startmillis = int(round(time.time()*1000))
        self.bot.senseMotion(self.direction)

        # want to store the last tach value every timeout millis
        # if, after timeout the tach value is unchanged, we're done
        lasttach = self.bot.getTacho()
        checktime = self.millis()

        while True:
            tach  = self.bot.getTacho()
            nowmillis = self.millis()

            if (self.bot.getTacho().is_near(self.target, fabs(self.inc)*self.threshold)):
                break

            # if we haven't checked the tach value in more than timeout seconds, do so
            if(nowmillis > checktime + self.timeout):
                checktime = self.millis()
                nowtach = self.bot.getTacho()

                if(nowtach.tacho_count == lasttach.tacho_count):
                    # we're done here
                    break
                else:
                    lasttach = nowtach

        self.bot.senseMotion(False)

        # if we need to send a confirmation within the modality
        if self.confirmfn:
            apply(self.confirmfn)

    def stop(self):
        self.stopped = True
```

Appendix C

Implementation Details for Greenfoot Delivery Agent

The MGLAIR version of the Delivery Agent discussed in 7 is based on an unpublished GLAIR version (mentioned in [Shapiro and Bona, 2010]) that originated with efforts to implement a version of Shakey the robot in GLAIR [Nayak et al., 2004].
The following section presents some of the MGLAIR version's `ActPlan`s, which make use of its proprioceptive modality to verify that it has moved as intended before proceeding. This behavior is essential to the proper operating of the agent. Without a way of waiting to verify that it has moved, the agent would, by default believe that it has finished each act as soon as the primitive action that implements it has finished evaluating. The result is an agent that will act repeatedly to move forward, e.g., when its body is still carrying out the action. This is prevented by linking its primitive acts to its proprioceptive modality at the knowledge layer, and by setting the locomotive modality's buffer to block impulses when it is already acting.

C.1 Knowledge Layer Excerpt

```
;;; To go to the end of the hallway, perform a verified go [forward]
;;;  (using proprioception) as long as there's a corridor ahead
;;;  (using passive visual sensing)
ActPlan(goToEnd(),
        sniterate({if(AheadIs(corridor),goAndVerify())})).

;;; To perform a verified turn, turn and then verify it
ActPlan(turnAndVerify(left),
        snsequence(turn(left), verifyTurn())).
ActPlan(turnAndVerify(right),
        snsequence(turn(right), verifyTurn())).

;;; To verify that you've turned: if you haven't turned,
;;; pause and then verify that you've turned
ActPlan(verifyTurn(),
        snif({if(~Turned(), snsequence(sleep(),verifyTurn()))})).

;;; To verify that you went, if you didn't then pause and verify
ActPlan(verifyGo(),
        snif({if(~Went(forward), snsequence(sleep(),verifyGo()))})).
```

```
;;; To perform a verified goForward, go and then verify
ActPlan(goAndVerify(),
        snsequence(goForward(), verifyGo())).
```

C.2 Modality Definitions

```
(define-modality 'range
  'afferent-modality
  :conflict 'squeeze
  :frames '(DistIs)
  :buffer '((capacity . 2)
  :description "Range-sensing modality used to detect the distance to the next corridor"
  :channel '((port . 9575) (host . localhost))
  :sense-handler 'range-sense-handler
  :print t)

(define-modality 'vision
  'visual-modality
  :frames '(Facing)
  :access t
  :focus '(t . elevated)
  :buffer '((exp-int . 1000))
  :channel '((port . 9574) (host . localhost))
  :sense-handler 'vision-sense-handler
  :description "Visual modality used by the agent to detect what's in front of it"
  :print t)

(define-modality  'visual-efferent
  'efferent-modality
  :frames '(readRoomNumber)
  :description "Used for active visual sensing, i.e. reading room numbers"
  :channel '((type . socket) (port . 9572) (host . localhost))
  :focus '(nil . high)
  :print t)

(define-modality  'locomotion
  'efferent-modality
  :frames '(turn goForward)
  :conflict 'blocking
  :description "Used by the agent to move and turn"
  :channel '((port . 9576) (host . localhost))
  :print t)

(define-modality  'locomotive-proprioception
  'afferent-modality
  :frames '(moving)
  :sense-handler 'proprioception-sense-handler
  :description "Senses motion"
  :channel '((port . 9573) (host . localhost))
  :print t)

(define-modality 'grabber
  'efferent-modality
  :frames '(pickUp putDown)
  :conflict 'blocking
  :channel '((port . 9577) (host . localhost))
```

```
:focus '(nil . 0)
:description "Used to manipulate the agent's grasping effector"
:print t)
```

C.3 PMLa

```
(in-package :snepslog)
(cl:load "./pmlb.cl")
(cl:load "./mglair/pmla.cl")

(define-primaction pickup-act ()
  (PMLb:pickup 'grabber))

(define-primaction forward-act ()
  (snepslog::tell "perform believe(~Went(forward))")
  (PMLb:forward 'locomotion))

(define-primaction putdown-act ()
  (PMLb:putdown 'grabber))

(define-primaction read-act ()
  (PMLb:read-room-number 'visual-efferent))

(define-primaction turn-act ((dir))
  "Go left, or right, according to the given direction."
  (snepslog::tell "perform believe(~Turned(right))")
  (case (sneps:node-na dir)
    ('left (PMLb:turn 'locomotion 'PMLb::l))
    ('right (PMLb:turn 'locomotion 'PMLb::r))
    (otherwise (format sneps:outunit
                 "~&***Invalid turn direction: ~S~%" (sneps:node-na dir)))))
```

C.4 PMLb

```
;;; PMLb for Greenfoot-based MGLAIR Delivery Agent
(defpackage :PMLb
  (:shadow common-lisp:find)
  (:export #:forward #:pickup
           #:putdown #:read-room-number #:turn))
(in-package :PMLb)

(cl:load "./pmls.cl")
(cl:load "./mglair/mods.cl")

;;; when t, causes the world to be automatically created and connected
(defvar *create-delivery-world* t)

;;; PMLb act implementations;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun read-room-number (mod)
  (execute mod "(read . room)"))

(defun turn (mod dir)
  (execute mod (format  nil "~A" (cons 'tn (if (eq dir 'l) -90 90)))))

(defun forward (mod)
  (execute mod "f"))

(defun pickup(mod)
  (execute mod "close"))

(defun putdown (mod)
  (execute mod "open"))

;;; Load modality definitions
(load "modality-defns.cl")


;;; PMLb sensing ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun vision-sense-handler (v)
  (PMLs:perceive-vision (read-from-string (rest v))))

(defun range-sense-handler (v)
  (PMLs:perceive-distance (read-from-string (rest v))))

(defun proprioception-sense-handler (v)
  (PMLs:perceive-proprioception (read-from-string (rest v))))

;;; Initialize the simulated world
(if *create-delivery-world*
    (load "create-delivery-world.cl"))
```

C.5 PMLs

```
;;; PMLs - PML sensory sublayer
(defpackage :PMLs
  (:export #:perceive-vision
           #:perceive-distance
           #:perceive-proprioception))
(in-package :PMLs)

(load "mglair/pmls.cl")

(defun perceive-vision (p)
  ;; takes forms like (ahead . r), (room . 1), etc
  (let ((type (symbol-name (first p))))
    (cond ((string= type "ahead") (perceive-ahead (rest p)))
  ((string= type "room") (perceive-room (rest p)))
  ((string= type "on") (perceive-package (rest p)))
  (t (warn (format nil "perceive-vision got unexpected input: ~A" p))))))


(defun perceive-package (p)
  (believe "~A(package)" (if (string= p "p") "On" "~On")))

(defun perceive-ahead (p)
  (believe "~A(~A)" 'AheadIs
   (symbol-name (cond ((string= p "r") 'room)
 ((string= p "c") 'corridor)
 ((string= p "w") 'wall)
 (t 'other)))))

(defun perceive-room (rnum)
  (believe "~A(~A(~A))" 'Facing 'room rnum))

(defun perceive-distance (d)
  (when (string= (symbol-name (first d)) "dist")
    (believe "~A(~A)" 'DistIs (case (rest d)
(0 'near)
(t 'far)))))

(defun perceive-proprioception (d)
  (if (numberp d)
      (believe "Turned(right)")
      (if (string= d "forward")
  (believe "Went(forward)"))))
```

C.6 Agent PMLc

```
class PMLc implements PMLcI{
    private ModalityManager modMgr;
    private DeliveryAgent bot;

    /**
     * Set up the agent's PMLc
     * @param configFN The filename containing modality data channel specifications
     * @param b The bot
     */
    public PMLc(String configFN, DeliveryAgent b){
    if (configFN != null){
        try{
            // initialize a new modality manager with the config file,
            // and connect modalities by name
            modMgr = new ModalityManager(configFN,this);
            modMgr.connectModalities("locomotion","grabber","vision",
                                     "visual-efferent", "range",
                                     "locomotive-proprioception");
        }catch(Exception e){
            System.err.println("Couldn't connect the Delivery Agent's modalities!");
            e.printStackTrace();
        }
    }
    this.bot = b;
    }

    public void handleSense(String modname, String s ){
        modMgr.getMCByName(modname).setSense(s);
    }

    public void handleDist(int dist){
        handleSense("range","( dist . " + Integer.toString(dist) + ")");
    }
    public void handleRoom(int roomnum){
        handleSense("vision","( room . " + Integer.toString(roomnum) + ")");
    }
    public void handleAhead(String aheadIs){
        handleSense("vision","( ahead . " + aheadIs + ")");
    }

    private void handleTurn(int deg){
    if(deg == -90){
      try{
        bot.turnLeft();
      }catch(InterruptedException ie){
        ie.printStackTrace();
      }
    }else if(deg == 90){
      try{
        bot.turnRight();
```

```
  }catch(InterruptedException ie){
    ie.printStackTrace();
  }
}else{
    System.err.println("bad turn direction in handleTurn: " + deg);
}

handleSense("locomotive-proprioception",Integer.toString(deg));

}

private void handleGrabber(String aname){
if(aname.equals("open")){
    bot.putDown();
}
else if(aname.equals("close")){
    bot.pickUp();
}else{
    System.err.println("bad actname in PMLc.handleGrabber " + aname);
}
}

private void handleVisualEfferent(String actname){
if(actname.equals("read")){
    bot.senseRoomNumber();
}else{
    System.err.println("bad actname in handleVisualEfferent " + actname);
}
}

private void handleLocomotion(String[] splitaction){
    if(splitaction[0].equals("tn")){
        int deg = Integer.parseInt(splitaction[1]);
        handleTurn(deg);
    }else if(splitaction[0].equals("f")){
      try{
        bot.goForward();
        handleSense("locomotive-proprioception","forward");
      }catch(InterruptedException ie){
        ie.printStackTrace();
      }
    }else{
        System.err.println("bad actname in handleLocomotion " + splitaction[0]);
    }
}

/**
 * Called on arriving impulses from efferent modalities.
 * Examines the impulse and calls the appropriate method(s) at the SAL.
 */
public void handleImpulse(String modname, String i){
    /*
```

```
        act impulses for this agent look like "(<time> . (<action> . <arg>))"
        where <time> is a long value used by the PMLb & data channel,
        <action> corresponds to the action (e.g. "tn" is for turning), and
        <arg> is its argument (e.g. 90)
         */

        String spliton = "(\\.|\\(|\\)|\\s)+";
        String[] impulse = i.split(spliton, 3);
        String[] splitaction = impulse[2].split(spliton);
        String aname = splitaction[0];
        //Long time = Long.parseLong(impulse[1]);

        if(modname.equals("visual-efferent")){
            handleVisualEfferent(aname);
        }else if(modname.equals("locomotion")){
            handleLocomotion(splitaction);
        }else if(modname.equals("grabber")){
            handleGrabber(aname);
        }
        //once the action has completed, confirm it in the same modality
        modMgr.getMCByName(modname).confirmImpulse(impulse[1]);

    }
}
```

C.7 SAL

```
import java.awt.Point;
import java.util.Arrays;
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

/**
 * MGLAIR SAL for Greenfoot Delivery Agent
 * @author Jonathan P. Bona
 */
public class DeliveryAgent extends Actor// implements MGLAIRActorI
{
  private boolean isCarrying = false;
  private PMLc pmlc;
  private int direction;
  private Package carriedPackage;
  private Floor theFloor = null;

  private final static int FORWARD_PAUSE_MS = 300;
  private final static int TURN_PAUSE_MS = 200;

  GreenfootImage nxtCarrying = getImage();
  GreenfootImage nxtImg = getImage();
  // keep track for each tick whether the agent has recently moved
  private boolean moved = false;

  /**
   * Act - do whatever the DeliveryAgent needs to do.
   * This method is called whenever the 'Act' or 'Run'
   * button gets pressed in the environment, and with every tick of the clock.
   */
  public void act()
  {
    if(moved){
        // sense every time the robot moves
        // an alternative would be to update all the "sensors" every m milliseconds
        senseAhead();
        senseDist();
//        sensePackage();
        moved = false;
    }
  }
  /**
   * Constructs a Delivery Agent by setting up the PMLc,
   * and handling initial direction and display
   * @param configFN the name of a file containing modality specifications
   */
  public DeliveryAgent(String configFN){
    super();
    setDirection(Direction.EAST);

    getImage().scale(40,40);
```

```
  getImage().rotate(-90);
  nxtImg = getImage();

  setImage("./NXTBOT-CARRYING.PNG");
  getImage().scale(40,40);
  getImage().rotate(-90);
  nxtCarrying = getImage();
  setImage(nxtImg);
  pmlc  = new PMLc(configFN,this); // set up the PMLc
}

/**
 * Gives the bot a reference to the Floor (i.e. the World)
 * @param f
 */
public void setFloor(Floor f){
  theFloor = f;
}

/**
 * If the bot is facing the entrance to a room, gets its number
 * @return the room number
 */
private int readRoomNumber(){
  if(aheadIs().equals("r")){
      Point aheadp = aheadPoint();
      Room r = theFloor.getRoom(aheadp.x, aheadp.y);
      return r.getRoomNum();
  }
  return 0;
}

/**
 * Gets the number of the room that the bot is facing (if any),
 * and passes it to the PMLc
 */
public void senseRoomNumber(){
  pmlc.handleRoom(readRoomNumber());
}

/**
 * Gets a description of what the bot is facing and passes it to the PMLc
 */
public void senseAhead(){
  pmlc.handleAhead(aheadIs());
}

/**
 * Gets the distance to the nearest main corridor in the direction
 * the bot is facing and passes it to the PMLc
 */
public void senseDist(){
```

```
  pmlc.handleDist(distIs());
}

/**
 * Returns true if the bot has moved from a location or facing
 */
public boolean movedFrom(Point loc, int facing){
  boolean result =!(getX() == loc.x && getY() == loc.y && facing == direction);
  return result;
}

/**
 * Moves the bot ahead one cell
 * @throws InterruptedException
 */
public void goForward () throws InterruptedException
{
  // run a thread that will let the modality know when it's done running
  Point ap = aheadPoint();
  setLocation(ap.x, ap.y);
  Thread.sleep(FORWARD_PAUSE_MS);
}

/**
 * Set the bot's location to cell <i>,<j>
 */
public void setLocation(int i, int j){
  moved=true;
  if(isCarrying) // if the bot is carrying a package, move that too
    carriedPackage.setLocation(i,j);
  super.setLocation(i,j);
}

private int getDirection(){
  return direction;
}

/**
 * Rotate the bot to the left
 */
public void turnLeft() throws InterruptedException
{
  setDirection((getDirection()+3)%4);
  Thread.sleep(TURN_PAUSE_MS);
}

/**
 * Rotate the bot to the right
 */
public void turnRight() throws InterruptedException{
  setDirection((getDirection()+1)%4);
  Thread.sleep(TURN_PAUSE_MS);
```

```
}

/**
 * Change the bot's direction to <dir>
 */
private void setDirection(int dir)
{

  this.direction = dir;
  switch(direction) {
    case Direction.SOUTH :
    setRotation(90);
    break;
    case Direction.EAST :
    setRotation(0);
    break;
    case Direction.NORTH :
    setRotation(270);
    break;
    case Direction.WEST :
    setRotation(180);
    break;
    default :
    break;
  }
  moved =true;
}


/**
 * Returns true if the bot is carrying a person, false otherwise
 */
private boolean isCarrying(){
  return isCarrying;
}

/**
 * Causes the bot to pick up a package if
 * it's not already carrying one and there's one in the current square
 */
public void pickUp() {
  if(!isCarrying && theFloor.containsPackage(getX(),getY())){
    carriedPackage = theFloor.liftPackage(getX(), getY());
    isCarrying = true;
    setImage(nxtCarrying);
  }
}

/**
 * Puts down the currently-carried person in the bot's current location
 */
public void putDown() {
```

```
  if(isCarrying){
    carriedPackage.setLocation(getX(),getY());
    isCarrying = false;
    setImage(nxtImg);
  }
}

/**
 * @return The cell one space ahead of the bot
 */
private Point aheadPoint(){
  return aheadPoint(1);
}

/**
 * Returns the cell n spaces ahead of the bot in the direction it's facing
 */
private Point aheadPoint(int n){
  switch(direction){
    case Direction.EAST:  return new Point(getX()+n,getY());
    case Direction.WEST:  return new Point(getX()-n,getY());
    case Direction.NORTH:   return new Point(getX(), getY()-n);
    case Direction.SOUTH:   return new Point(getX(), getY()+n);
    default: return null;
  }
}

/**
 * @return 0 if the agent is facing a main corridor that's near;
 * 1 if far;
 * -1 in exceptional cases
 */
public int distIs(){
  Point twoAhead = aheadPoint(2);
  // if the agent's facing east or west toward a main corridor,
  // want to know whether the corridor is near or far
  if(aheadIs().startsWith("c") && direction == Direction.EAST ||
                                   direction == Direction.WEST){
      return atPoint(twoAhead).startsWith("w") ? 0 : 1;
  }
  return -1;
}

/**
 * Returns a String naming the thing at Point p.
 * @param p a Point on the world grid
 * @return String naming the object at Point p
 */
public String atPoint(Point p){
  if(theFloor.containsRoom(p.x,p.y)) {
    //if the room is facing this way, it's a room
    if (theFloor.getRoom(p.x,p.y).opensTo(getX(),getY())){
```

```
        return "r";
      }else{ // otherwise it's a wall
        return "w";
      }
    }else if(theFloor.containsCorridor(p.x, p.y)){
      return "c";
    }else if(theFloor.outOfBounds(p.x, p.y)){
      return "w";
    }
    return null;
  }

  /**
   * @param n
   * @return a String describing what's ahead n spaces
   */
  public String aheadIs(int n){
    Point ahead = aheadPoint(n);
    return atPoint(ahead);
  }

  /**
   * Returns a String describing what's in the adjacent cell in the dir it's facing
   */
  public String aheadIs(){
    return aheadIs(1);
  }
}
```