Cognitive Architecture and Testbed for a Developmental AI System

Fourth Year Honours Project

2009/2010

David Fraser Bruce
06203355

Copy 1 of 2

Acknowledgements

I would like to thank my project supervisor, Dr. Frank Guerin for all his advice and assistance during the development of this project, and also for providing such an interesting and rewarding topic for the basis of my Honours project.

I would also like to thank my friends and fellow students for their words of wisdom and advice throughout the project's duration.

Declaration

I declare that this document and the accompanying code has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed: ……..

David Bruce **Date:** ……..

Abstract

A Developmental AI system can be defined as a system which can develop knowledge and skills from the environment it is situated in autonomously, by experimenting within it, and learning from the results produced.

This project aims to develop a testbed and cognitive architecture for such a system, focusing on cognition in infants, and based on Piaget's *Theory of Cognitive Development*. The system will provide the facilities for developing and debugging the learning mechanism that will ultimately be integrated into it, as well a modular and extensible architecture which facilitates future development.

The project will eventually constitute the foundation of a system used to carry out further research into Developmental AI systems, aiming to model infant competences, and more significantly, determine how those competences were learned.

Table of Contents

Acknowledgements ..... **i**

Declaration ..... **ii**

Abstract ..... **iii**

Index of Figures ..... **vii**

Index of Tables ..... **vii**

Chapter 1: Introduction **1**

1.1. Overview 1

1.2. Objectives 1

1.3. Motivations 2

1.4. Primary Goals 3

1.5. Secondary Goals 3

1.7. Structure of Document 3

Chapter 2: Background **5**

2.1. Developmental AI 5

2.2. Schema 5

2.3. Cognitive Architecture 6

2.4. Testbed 6

2.5. Existing Solutions 7

2.6. Current System 7

Chapter 3: Requirements **9**

3.1. Project Requirements 9

3.2. Performance Requirements 10

Chapter 4: Methodology & Technologies **11**

4.1. Methodology 11

4.2. Technologies 12

4.2.1. Java 12

4.2.2. IDE 12

4.2.3. Physics Engine & Graphics Package 13

4.2.4. Data Storage 13

4.2.5. Versioning 14

Chapter 5: System Design & Architecture 15
5.1. System Design 15
5.1.1. Robot Infant 15
5.1.2. Cognitive Architecture 17
5.1.2.1. Schema 17
5.1.2.2. World States 18
5.1.2.3. Response 18
5.1.2.4. Schema Library 18
5.1.2.5. Schema Manager 19
5.1.2.6. Schema Harvester 19
5.1.2.7. Forward Simulator 19
5.1.3. Transition Experience Generation 19
5.1.4. User Interfaces 20
5.1.5. Data Storage 22
5.2. System Architecture 23
5.2.1. Overview 23
5.2.2. Packages 24
Chapter 6: Implementation 25
6.1. Graphics Representation 25
6.1.1. The Infant 25
6.1.2. Joints 26
6.1.3. Hand Contact 26
6.1.4. Vision 27
6.1.4.1. Seen Objects 28
6.1.5. Reaching 28
6.1.5. Obstruction 29
6.1.6. Body Movements 30
6.1.7. Calculations 30
6.2. GUI 31
6.2.1. Integrated Graphics 31
6.2.2. Graphics Controls 31
6.2.3. GUI Features 32
6.3. Architecture 33
6.3.1. Architecture Details 33

6.3.2. Startup 33
6.3.3. Consts Class 34
6.4. Cognitive Architecture 34
6.4.1. Sensor Data 34
6.4.2. Response 35
6.4.3. Schema 36
6.4.4. Schema Library 37
6.4.5. Schema Manager 37
6.4.6. Schema Harvester 38
6.4.7. Forward Simulator 38
6.5. Transition Experience Generation 39
6.6. Learning Schemas 41
6.7. Additional Tasks 41
6.8. Summary 42
Chapter 7: Testing and Evaluation 43
7.1. Testing 43
7.1.1. Individual Component Testing 43
7.1.2. Schema Execution Testing 43
7.1.3. User Testing 45
7.1.4. Scalability Testing 45
7.1.5. Speed and Efficiency 45
7.2. Evaluation 46
7.2.1. Overall System Evaluation 46
7.3. Known Bugs 47
Chapter 8: Summary and Conclusion 49
8.1. Summary 49
8.2. Future Development 50
8.3. Conclusion 51
References/Bibliography 53
Appendix A: User Manual 55
Appendix B: Maintenance Manual 73
Appendix C: Example XML Files 80
Appendix D: Example Schemas 89
Glossary 92

Index of Figures

Figure 1 - The iCub 7
Figure 2 - Current Physics Engine 8
Figure 3 - Resting Contact of Blocks 8
Figure 4 - Gantt Chart Displaying Project Timeline 11
Figure 5 - JBox2D Rigid Physics Demo 13
Figure 6 - Labelled Infant Screenshot, Prototype Version 15
Figure 7 - Labelled Infant Screenshot, Final Version 16
Figure 8 - Cognitive Architecture Diagram 17
Figure 9 - Labelled Main GUI Screenshot 21
Figure 10 - Labelled Schema GUI Screenshot 22
Figure 11 – System Architecture Diagram 23
Figure 12 - Main Packages of System 24
Figure 13 - Hand Contact Example Scenario 27
Figure 14 - Display of Infant Vision 27
Figure 15 - Reach Example Scenario 29
Figure 16 - Obstruction Example Scenario 30
Figure 17 - Save State Functionality Summary 32
Figure 18 - Load State Functionality Summary 32
Figure 19 - Response Type Enumeration 36
Figure 20 - Tree Generation Example Output 39
Figure 21 - Transition Experience Generation Algorithm 40
Figure 22 - Transition Experience Generation Example Screenshot 40

Index of Tables

Table 1 - State of World object contents 18
Table 2 - SensorData Object Example 35
Table 3 - Example Schema Object - Grab 37
Table 4 - Schema Execution Test Results 44

Chapter 1: Introduction

This chapter provides an overview of the project, including its objectives and goals, as well as detailing the structure of this report as a whole.

1.1. Overview

Developmental AI involves the process of attempting to build a system which can develop knowledge and skills from its environment autonomously, by experimenting within it, and learning from the results produced.

Although, at the time of writing, Developmental AI systems are a relatively unexplored area within the broader AI spectrum, a significant amount of research has been carried out on the topic[1], with a number of systems emerging. One such instance of a Developmental AI system has been created by Dr. Frank Guerin, of the Department of Computing Science at the University of Aberdeen.

The current system, based upon Piaget's *Theory of Cognitive Development*, employs a basic cognitive architecture utilizing Schema objects to enable a robot infant to learn about its environment through its actions.

This system, however, has been worked on in an ad-hoc manner for a number of years, meaning that older, now redundant, sections of code still remain alongside newer parts. Additionally, the physics engine currently employed has become somewhat outdated, and hence needs to be replaced in order to handle more advanced features the current engine cannot.

The purpose of this project is to create a testbed and cognitive architecture for a similar Developmental AI system. The project will follow the same principles as the existing system, but start it from scratch, in order to resolve the presiding problems, and to provide a cleaner implementation which can be easily expanded upon in the future.

This project focuses mainly on the "software engineering" aspects of developing a Developmental AI system, creating the testbed, user interface and architecture, as well as providing the framework for incorporating **learning** and **vision** packages in a modular manner. It will ultimately be used in collaboration with the other two sections of the system, which are being worked on as two separate projects.

1.2. Objectives

A robot infant and the "world" it is situated in must be represented graphically, and should include a number of components. The infant itself should consist of a head and an arm, which should operate as it would in real-life. It should also have an indication of vision and focus in the form of an *eye* and a *fovea* respectively. The infant's world must also be populated with other objects, including blocks, and sticks, with which it can interact.

The graphical display must be created using an improved physics engine, which will replace the one currently employed. This needs to be accomplished in order to present a more realistic simulation, by providing features the current engine cannot handle properly, like the resting contact of blocks, required for creating stacks of objects.

Alongside the graphical representation of the infant, the system must have a GUI which allows the user to interact with the system. The GUI must provide a number of facilities to the user to allow manipulation of the graphical display, as well as controls for other important features, which will support development and debugging of the learning mechanism when the system is fully operational.

In order to provide a cleaner implementation of the previous system, the architecture must be re-designed and overhauled, with a focus on modularity and extensibility, to ensure future development is straightforward. The architecture must also ensure that incorporating different parts of the project, like the learning package, is an easy task.

Another key objective of the project is to implement a cognitive architecture which allows for simple integration with the learning section of the system. Like in the existing system, the cognitive architecture will be based upon Piaget's *Theory of Cognitive Development*.

The final objective of the project, once all of the above tasks have been completed, is to illustrate some basic behaviours of the infant, by creating some handcoded Schemas to be loaded into the system. The infant will ultimately be able to learn these behaviours for itself.

1.3. Motivations

The project, as a whole, when combined with learning and vision sections, aims to model the learning processes of a human infant, in order to provide an insight into the techniques an infant might use to learn.

When acquired, this information could help further research in the field of Developmental AI, and hence help overcome some of the obstacles that affect every "traditional" AI program (see 2.1. Developmental AI, in the next chapter).

Providing a testbed for a Developmental AI system is imperative, as, in this system, it helps provide a more precise depiction of the problem at hand. By creating a realistic display of the infant, it becomes much easier to visualize certain states which the learning mechanism will process. Additionally, it allows for the testing of specific scenarios, meaning it is easier to experiment within the "world" using the learning mechanism.

Providing a GUI for the system is also crucial, as it will provide support for developing and debugging the learning mechanism, allowing the user to inspect the inner data structures of the system, making visible what has been learned thus far.

The testbed and cognitive architecture combined will essentially provide the fundamental foundation for the rest of the Developmental AI system.

1.4. Primary Goals

The primary goals of the project are as follows:

- To provide a new, clean implementation of the previous system with restructured architecture, with an emphasis on modularity and extensibility.
- To create a redesigned graphical representation of the robot infant and world, making use of a more advanced physics engine.
- To design and implement a new Graphical User Interface for the system, incorporating functionalities to aid the development and debugging of the learning mechanism.
- To provide a cognitive architecture framework, based on Piaget's concept of a Schema, for use with the learning mechanism, when it is incorporated into the architecture.
- To demonstrate some example behaviours, including grabbing blocks, which the robot baby will ultimately be able to learn for itself, by making use of hand-coded Schemas.

1.5. Secondary Goals

The secondary goals of the project, to be worked on if time permits after the primary goals are achieved, are as follows:

- To provide a basis for creating more complex behaviours, including pulling a block closer using a stick; pulling a longer block closer in order to obtain another block resting on top; and pulling a string closer in order to retrieve the block on the end of it.

1.7. Structure of Document

Chapter 2 of this document aims to provide an overview of the background work relating to the project. The theory behind the project is discussed; including some related works, as well as the motivation for replacing the existing system.

Chapter 3 of this document details the project requirements, both functional, and non-functional, including the performance requirements.

Chapter 4 looks at the techniques and methodology used throughout the project, with a discussion of the technologies employed and the motivation behind their selection.

Chapter 5 details the preliminary design work behind individual aspects of the project, including the design of the graphical display, GUI and cognitive architecture, as well as the system architecture.

Chapter 6 details the implementation stage of the project, with an in-depth discussion of how each section of the project was constructed, including details of problems and challenges encountered in doing so.

Chapter 7 discusses the testing of the system and its performance, as well as detailing existing bugs. The chapter also provides an overall evaluation of the system implemented in this project.

Chapter 8 summarises the project, detailing the future work on the system, and provides a conclusion to the report.

Appendices containing the Maintenance Manual and the User Manual for the system are provided at the end of this document, as well as Appendices containing sample XML documents and Schemas as created during the course of the project.

Chapter 2: Background

This chapter examines some of the background work relating to the project, including a brief analysis of the current system.

2.1. Developmental AI

Studies in Artificial Intelligence (AI) to date have produced some fascinating findings, resulting in systems which are able to interact with humans in an intelligent manner.

However, one of the major shortcomings of "traditional" AI at present is that no real progress has been made in creating a program which possesses all the "common sense" knowledge an average human learns from experience; only what it has been programmed to "know".

Hence, research has commenced in the field of Developmental AI, which takes its inspiration from human nature, particularly the way humans learn about their surroundings as infants.

Developmental AI involves trying to create a program which can continuously, and autonomously, build upon its current knowledge and skills through its environment, by taking an action upon it and learning from the results.

In theory, such a program will then be able to perpetually increase what it knows; therefore obtaining a far more sophisticated knowledge base, meaning it can interact in a more intelligent manner.

2.2. Schema

The concept of a Schema, as used extensively throughout this project, was first defined by Jean Piaget[2] in 1936, in his *Theory of Cognitive Development*[3]. Piaget's theory, detailing the development of human intelligence, essentially described how children reason and represent about the world they inhabit.

Taking inspiration from infants, Piaget attempted to theorise how humans acquire information from their surroundings and experiences, building upon their current knowledge, in order to utilise it in similar situations in the future.

After observations, Piaget postulated that all children, as they grow older, progress through four stages of development; *The Sensorimotor Period*, *Pre-operational Thought*, *Concrete Operations*, and *Formal Operation*s. It is these four stages that compose Piaget's *Stage Theory of Cognitive Development* and, it is during these stages, Piaget theorised, that children make use of inner Schemas.

The Schema object that Piaget defined is essentially a snapshot of internal knowledge about a child's surroundings relating to a specific behaviour at a particular moment in time. It contains everything that the infant currently knows about their circumstances, their environment, and the result of their actions at present, based on their past experiences in similar situations.

In theory, many of these Schema objects would then be used by the child as it develops, in order to determine how to react to different experiences. Piaget defined this process as *Assimilation*, which is essentially the child reacting to a situation based on its pre-existing beliefs.

If the child encounters an entirely new situation, which it has no prior knowledge of it would then alter its existing Schemas in order to take in this new information. This action was referred to by Piaget as *Accommodation*, which is the process of changing existing beliefs as a result of new experiences.

Piaget defined the balance of these two processes, *Assimilation* and *Accommodation* as being achieved by a mechanism called *Equilibration.* This balance is essentially the balance between using previous knowledge to react to situations, and altering current knowledge based on new experiences. Piaget believed that all children maintain this balance, and that it is necessary for them to do so in order to progress to the next stage of development.

The cognitive architecture and learning section of the system being implemented in this project is based primarily on this aspect of Piaget's *Theory of Cognitive Development*.

2.3. Cognitive Architecture

A key goal of this project is to implement a c*ognitive architecture* for a Developmental AI system. A cognitive architecture, in this situation is an attempt to provide a plausible model of the main internal components of the mind, and how they are linked.

A number of attempts at cognitive architectures already exist, including Soar[4] and ACT-R[5], which focus on the internal information processing of a system, and ICARUS[6], which also looks at learning, and storing learned rules in long-term memory.

The majority of the existing cognitive architectures, including Soar[7] and ACT-R[8], are rule-based systems, and employ "chunking" techniques as part of their learning mechanism. Chunking primarily involves remembering a solution to a problem, so that it can be called and utilized in a similar situation at a later stage. This varies from the process used within this system, which aims to learn brand new rules for each problem encountered.

Also, all three of the above cognitive architectures attempt to model human competence in some specific area. However, this project diverges from those attempts in that it aims to create a cognitive architecture which focuses on modelling human infant competences, and specifically how those competences were learned; following the Developmental AI approach, and taking inspiration from Piaget's theory on Schemas.

2.4. Testbed

A testbed is essentially a platform to allow for the experimentation and testing of software projects. It is similar to the concept of a "sandbox" in software development, in that it allows separate parts of a project to be tested individually, isolated from the system as a whole.

For this project, a testbed is required to provide facilities for the rigorous testing of particular states. Additionally, the testbed will ultimately assist in the development and debugging of the learning mechanism.

2.5. Existing Solutions

As mentioned previously, some research in Developmental AI has already been carried out, resulting in a number of existing solutions to the creation of a Developmental AI system.

However, this project focuses specifically on creating Developmental AI system with a cognitive architecture for infancy; something which has not yet been fully explored.

The CogX (*Cognitive Systems that Self-Understand and Self-Extend*) project[9], for instance, is developing a unified theory of self-understanding and self-extension, with an implementation of this theory in a robot. The resulting cognitive architecture could certainly be incorporated in a Developmental AI system, but is more suited to modelling an adult than an infant.

The RobotCub[10] project, however, focuses specifically on child cognition, making use of a humanoid robot, with sensory inputs, representing a 3.5 year old child; the iCub, pictured in Figure 1, to the right. The key aims of the RobotCub project are to support community research on embodied cognition, and to advance understanding in several key issues in cognition.

Figure 1 - The iCub

The RobotCub project differs from this project in a number of ways. For instance, it has a strong focus on robotics, and the technology employed to provide multiple sensory inputs. Additionally, the RobotCub project is not based specifically on the work of Piaget alone, but on child cognition in general.

This system provides a more faithful model of Piaget's theory than anything previously attempted. Making use of the Schema object, which in itself is an entity, able to assimilate everything in the world, and take a specific action based on this, differs from "traditional" AI systems; which usually have some sort of central planning module, deciding on a goal and performing searches in order to reach it.

It is apparent that none of the existing systems have yet focused entirely on creating a cognitive architecture for infancy, and it is also visible there is not currently a prevalent solution which is totally faithful to the work of Piaget. Therefore, this system can be seen as providing a cognitive architecture which is more authentic, given the background of the subject area.

2.6. Current System

As explained previously, this project will be based upon an existing system, which has been developed by Dr. Frank Guerin and a number of students over the past few years.

However, because this system has been worked on in such an ad-hoc manner, new pieces of code have been added alongside old sections, meaning key parts of the system no longer function correctly, if at all.

The design of the previous system was by no means modular, with some classes growing exponentially with no real regard for structure of the system. Additionally, no considerations were in place for future extension of the system, meaning classes needed to be continually altered and added to in order to accommodate changes.

Because the system has fallen into such a state of disrepair, it is therefore necessary to re-start it from scratch, whilst retaining the majority of the concepts used throughout.

Producing a new, cleaner implementation of the system means it will be a lot easier to build upon at a later stage, and so it will be able to facilitate further work and research for the foreseeable future.

Another of the main parts of the system which needs to be revamped is the graphical depiction of the infant and the world, including the physics engine used to create it. The current physics engine is limited in what it is able to process, and so cannot handle more complex situations which are required in order to provide a realistic simulation (see Figure 2, to the right).

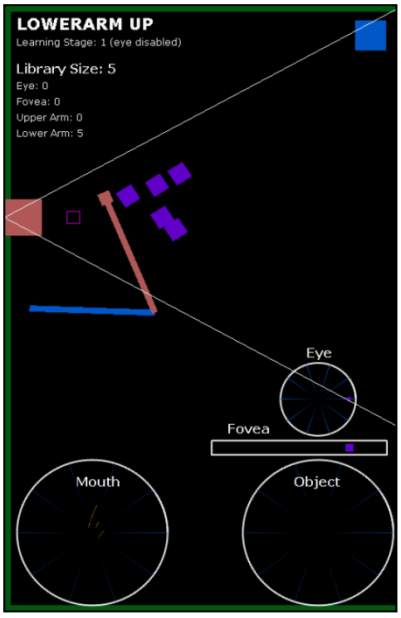

Figure 2 - Current Physics Engine

Some of the behaviours that the infant should ultimately be able to learn require features from a physics engine that the current version simply cannot provide. The current physics engine has been used for a number of years, and hence open-source software of a similar nature is now far more advanced, offering many desirable features to the project.

For instance, one of the behaviours the infant must ultimately learn using the learning mechanism is to recognise when a block is sitting on top of another, and attempt to obtain this block by moving the one underneath closer.

This behaviour requires resting contact of blocks (see Figure 3, below); something which is not provided by the current physics engine, but is available on newer, more advanced versions. Resting contact allows the simulation to create and display stacks of blocks, which can be useful for depicting certain scenarios.

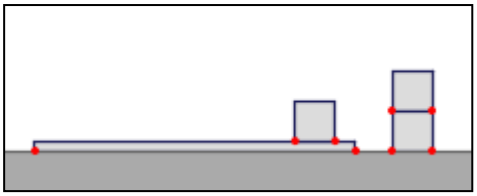

Figure 3 - Resting Contact of Blocks

By re-designing the graphical representation, and updating the physics engine, the system will be able to provide a more accurate, realistic depiction of the infant and the world it inhabits.

Chapter 3: Requirements

This chapter details the requirements for the project, both functional, and non-functional.

3.1. Project Requirements

The main functional requirements of the project are as follows:

- To replace the current physics engine with a more advanced one.
- To provide a graphical representation of an infant situated in a world, with a moveable arm, and some indication of vision, alongside other objects like blocks.

  The graphical representation must also provide the user with the functionality to manipulate the infant and the objects in world, using either mouse or keyboard controls.
- To revamp the system architecture, providing a cleaner implementation, whilst retaining the majority of the same functionalities as the previous version with an emphasis on extensibility and modularity.
- To create a cognitive architecture framework to allow the learning and vision sections of the system to be integrated. This should be based on Piaget's *Theory of Cognitive Development*.
- To create a Graphical User Interface (GUI), providing a number of functionalities to the user allowing them to interact with the graphics display, in order to stage experiments.

  The GUI must also provide the user the means to inspect the contents of the currently-loaded Schema Library, as well as view and/or execute the individual Schemas contained within it, in order to help develop and debug the learning mechanism.
- To display some basic, elementary behaviours using handcoded Schemas, such as:
  - Looking at an object, picking it up, and pulling it towards itself. This is the most basic chained-Schema behaviour.

  Additionally, the system must provide the basis for displaying more complex behaviours, like recognising when a block is resting on top of another, or realising there is an obstruction in the path of an object to be grabbed.

The system implemented in this project will provide the overall Developmental AI system with a testbed, user interface and cognitive architecture; essentially building the foundations of the system. The learning and vision sections, which will be mentioned throughout this report, are being implemented as two individual projects by two other students.

3.2. Performance Requirements

The graphical display and physics engine used in the system will require a significant amount of processing power. Additionally, as a simulation runs, the size of the Schema Library will increase exponentially, as new Schemas are obtained and added; this again will require a large amount of processing power and memory.

Hence, the performance of the system must be given significant consideration, as it is unsatisfactory for it to crash or slow down considerably when under high strain. Therefore, it is necessary to ensure that in such situations, the system will continue to operate at an acceptable speed, with no real impairment to the functionalities provided.

However, the efficiency of the system with regards to speed is a lower priority; it is important to focus on implementing features which function correctly first, before the efficiency of the processes is considered.

Chapter 4: Methodology & Technologies

This chapter discusses the methodology used throughout all the stages of the project, as well as the technologies employed, explaining the motivations behind their selection.

4.1. Methodology

Prior to beginning the design and implementation stages, the related topics behind the project were researched, to gain an in-depth understanding of the underlying theory involved. This included literatures detailing Piaget's work with Schemas, as well as papers written on other cognitive architectures.

Additionally, as many of the fundamental concepts from the previous system were to remain consistent, it was imperative to examine the existing source code, so any previously used techniques could at least be considered.

After the system had been divided into three sections by the project supervisor, the requirements and project plan were composed, including the main goals of the project, and a time-line (see Figure 4, below.)

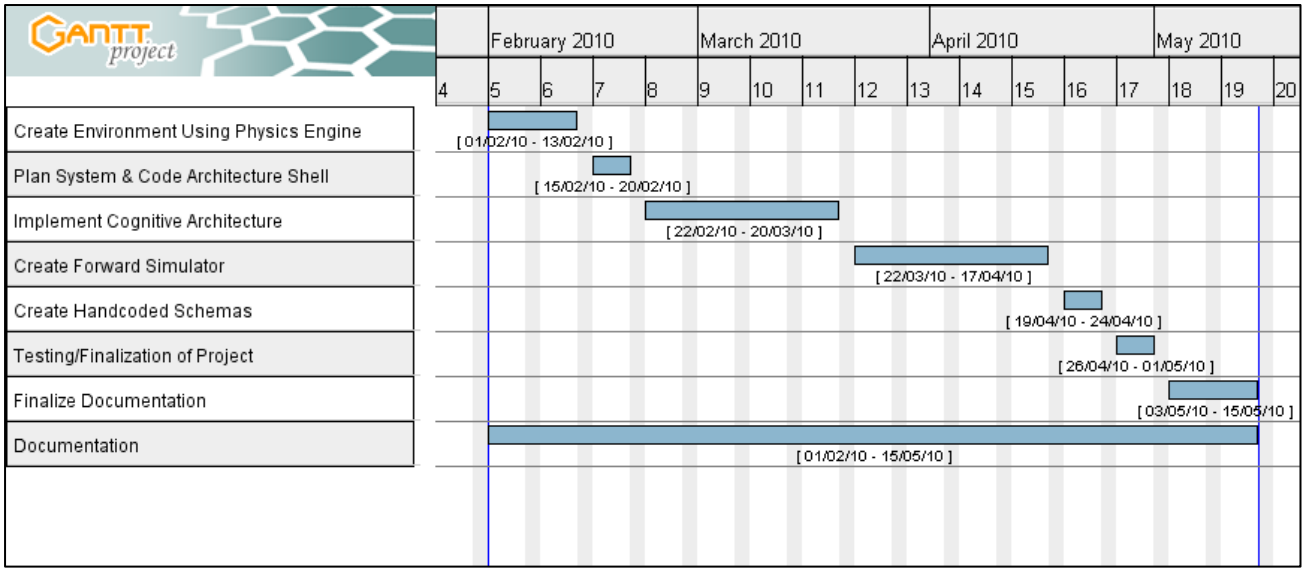

Figure 4 - Gantt Chart Displaying Project Timeline

In the design stage of the project, the individual parts of the testbed were designed, including the GUI and graphical display, the cognitive architecture, and the system architecture.

Prior to commencing the implementation stage of the project, the technologies to be employed were studied in-depth; particularly the graphics package and physics engine, which were not used previously.

The AGILE approach was employed during the implementation phase of the project, as new requirements emerged during weekly meetings with the project supervisor and other students working on different sections.

To ensure the project could recover from incidents of data loss or corruption, regular backups of the project were made, both on a Subversion server, and to a location outwith the University, thus, there was always an up-to-date version of the project available to work on.

Documentation was produced on a regular basis, in order to ensure that progress was being made on the final report, as well as the project itself.

4.2. Technologies

This section details the technologies selected to implement this project, explaining the motivations behind their selection.

4.2.1. Java

The entire project was written using the Java programming language. After some initial research, it was apparent the system could also have been implemented using C++, but ultimately, Java was selected for a number of beneficial reasons.

First, Java is the programming language consistent amongst all members of the group working on this system. Using Java ensured no complications arose from the eventual integration of the three sections of the project, as no considerations needed to be made for different languages.

Second, because the previous system was also implemented using Java, it was therefore clear that all of the requirements for this system were feasible using the same language. Because a lot of the requirements remained the same, there was no question to whether they could be accomplished in the new system if Java was used again.

Additionally, Java allows the system to be device independent, meaning it theoretically be run from any machine with the Java Runtime Environment installed. This was important to the development of this project in particular, as the three sections of the system were being developed on Linux, Windows, and Apple operating systems, in parallel.

4.2.2. IDE

The project was written in the NetBeans IDE 6.8. NetBeans was selected mainly due to personal preference, but it also provides a number of other advantages over other IDEs where this project is concerned.

Perhaps the most visible benefit that NetBeans provided to the implementation of this project is that it has an embedded GUI builder, which allows GUIs to be designed in a simple and efficient manner. Each GUI can be designed by placing each object visually, as opposed to writing the positioning into the code, something which can be very useful when trying to design the overall appearance.

4.2.3. Physics Engine & Graphics Package

In order to improve the physics engine in the new system, both the old graphics and physics libraries needed to be replaced with updated ones, which could handle more complex features than the previous ones.

It was decided at the very beginning of the project that a possible replacement physics engine was *JBox2D*[11], which is available to download free at the project's Source Forge page[12]. JBox2D provides all of the features which are required for the representation of the infant in this project; including rigid body physics, stable stacking of blocks, and motors, which are required for movement of the body joints. A demonstration program displaying some of these features is displayed in Figure 5, to the right.

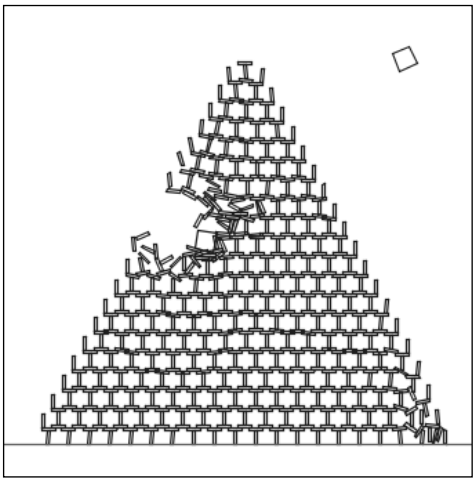

Figure 5 - JBox2D Rigid Physics Demo

As JBox2D is a Java port of a C++ physics engine, *Box2D*, it would again have been possible to implement the physics engine using C++ instead. However, as mentioned previously, Java was the consistent language amongst the group, and hence it was deemed suitable to continue working with it for this section of the project, and make use of JBox2D.

Unfortunately, at the time of writing, no real documentation exists for the JBox2D library, so the documentation for the C++ Box2D library[13], and source code from the demonstration programs provided by the developers were used for reference.

Along with JBox2D, the project makes use of *Processing*[14], a free API, which provides the graphics rendering. Processing was used alongside the JBox2D demonstration programs given online, hence it was visibly the best choice for graphics rendering, as it had already been used in a similar situation.

Clearly, the main benefit both JBox2D and Processing provide is that they are open source, and so free to use by anyone who requires them, which means no money needed to be spent in obtaining a high quality physics engine for the project.

4.2.4. Data Storage

As well as using Java, XML (Extensible Markup Language) was also used to satisfy some of the project requirements involving data storage. The saving and loading of states was implemented using XML files which recorded the individual positions of each object in the world.

It would of course have been possible to use a database for this purpose, making use of SQL queries to write to and read from the storage location. However, XML provides a far more lightweight solution to data storage in this instance, and also means the user does not need to be connected to a network, or require access rights, in order to load and save data.

A number of different file types could have been used to save data, but XML was selected mainly because of its simplicity in comparison to the alternatives. Additionally, XML it is so extensible in its creation of tags, meaning it is also very readable for users when trying to debug errors, or find specific information they might require.

4.2.5. Versioning

Throughout the project, daily backups were kept, using the Subversion version-control software[15]. Subversion provides the means to maintain both current and historical versions of source code, hence allowing the user to backtrack to a previous version of a project, if deemed necessary.

Clearly, using Subversion provided the standard advantages that can be said of any backup system; there was always another copy of the project in existence, in case the original was lost or corrupted. Also, it was relatively easy to return to a previous version of the project, if it was required.

Additionally, making use of Subversion allowed for collaboration amongst multiple users on the project. Because parts of the system were worked on by several people, and each individual's work needed to be integrated together, creating a single Subversion repository allowed each person to obtain and work on the latest version of the project whenever they required it.

Chapter 5: System Design & Architecture

This chapter details the preliminary design considerations behind the system, including the design of the robot infant, user interfaces, and cognitive architecture, as well as the architecture employed within the system.

5.1. System Design

This section describes some of the decision decisions that were made prior to the actual implementation stage of the system.

5.1.1. Robot Infant

It was vital that the graphical depiction of the infant included all the features detailed in the project requirements.

Initially, all that was required of the display was to present the robot infant itself, without any of the vision indicators. After some detailed analysis of the usage of the physics and graphics packages, the graphical prototype of an infant and world as displayed in Figure 6 was produced.

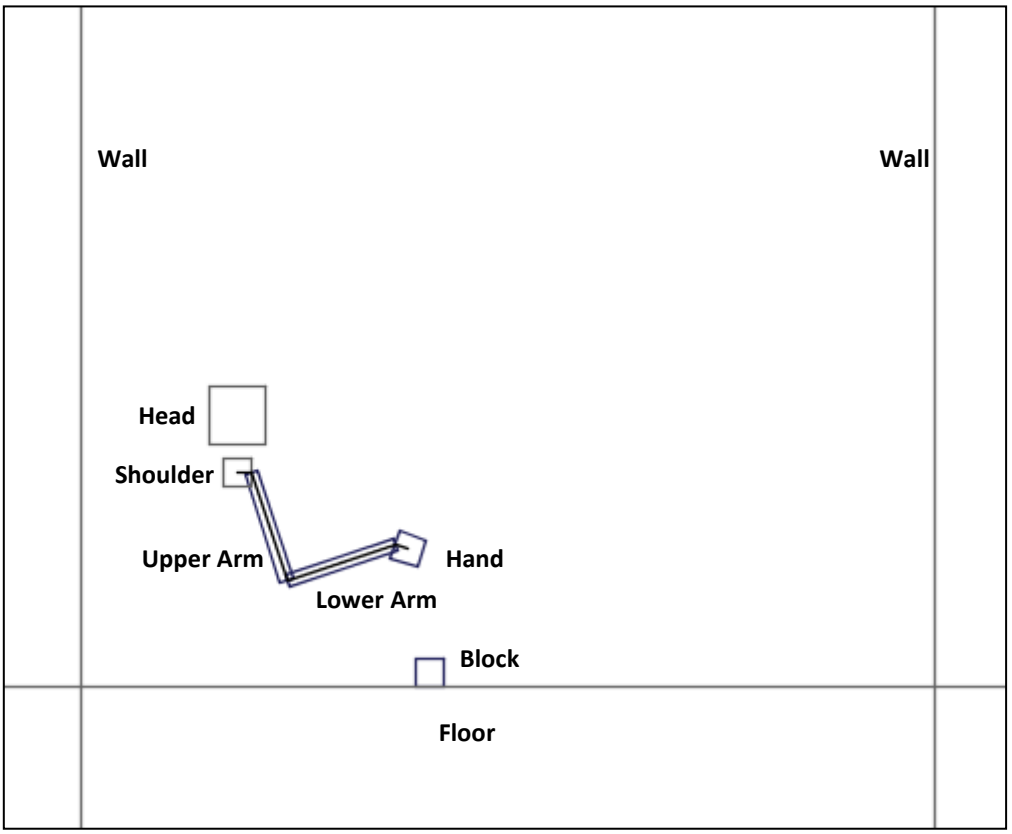

Figure 6 - Labelled Infant Screenshot, Prototype Version

The graphics display above meets all of the initial requirements; the infant is made up of a head, shoulder, upper-arm, lower-arm and hand; the arm segments can all be moved individually.

The infant is contained within a world, with wall and floor boundaries in place to prevent objects becoming completely out of the infant's field of vision and reach.

Once the initial prototype was produced, it was deemed necessary by the project supervisor to incorporate a number of other features to the graphical display. First, and perhaps most importantly, the display was required to indicate the infant's field of vision in some way; both the general direction, as well as its point of focus.

Additionally, as a side requirement for another student's work on the vision section of the system, the graphical display was required to distinguish between the components of the world – the infant, the blocks – and the background, by introducing some colour to allow this differentiation.

After making the above refinements to the initial design considerations, the infant and its world were designed using the physics engine as displayed in Figure 7.

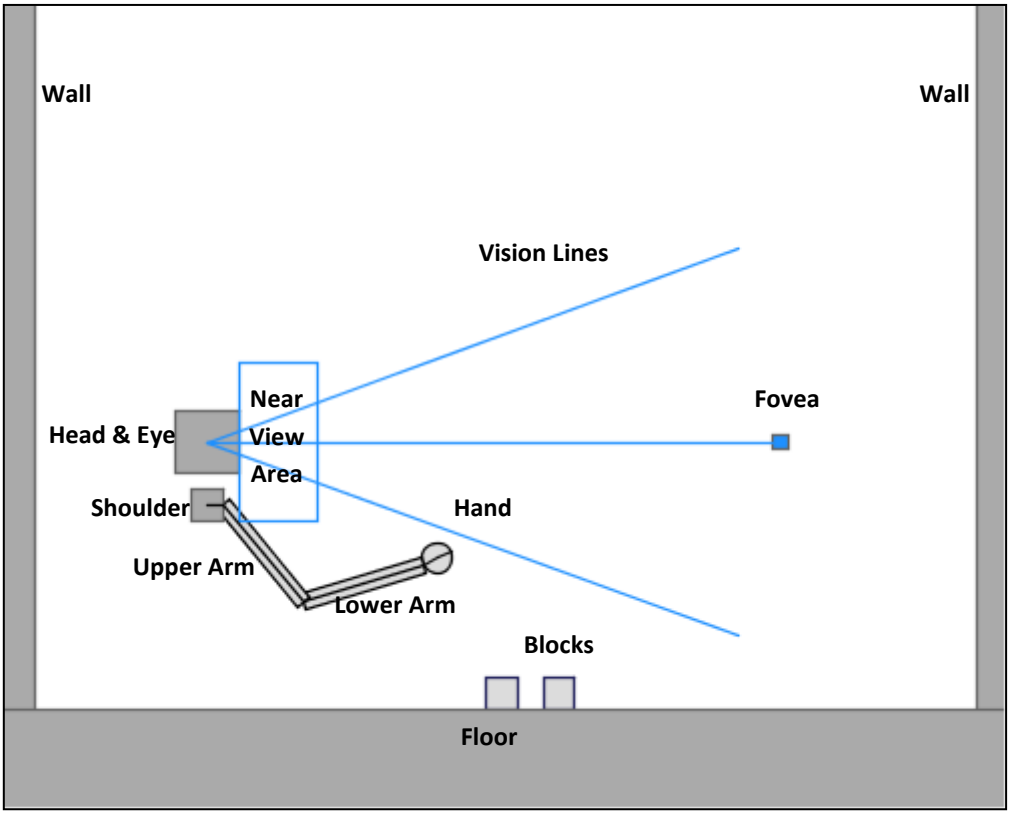

Figure 7 - Labelled Infant Screenshot, Final Version

The vision lines, represented in blue, can be moved up and down, and the fovea in and out, to indicate movement in the infant's field of vision.

The *Near View Area*, in front of the infant's face is the key area of interest in its field of vision. It is within this region the infant must aim to pull objects when learning new behaviours, and hence is pivotal to the learning section of the system.

Colour was also introduced into the display, distinguishing between moveable objects (represented in a light grey), stationary objects (represented in dark grey), and the background (represented in white).

Keyboard controls for the movement of objects within the graphics display were also designed, with the numeric keypad being used to control the infant's arm and field of vision, along with basic actions, like grabbing and dropping objects. The keyboard and mouse controls are significantly important, as they will allow the user to set up specific scenarios when using the learning mechanism in the future.

5.1.2. Cognitive Architecture

The cognitive architecture to be implemented will be based on Piaget's concept of a Schema, as discussed previously.

The cognitive architecture will require a number of components for storing and executing Schema objects, as illustrated in Figure 8, below.

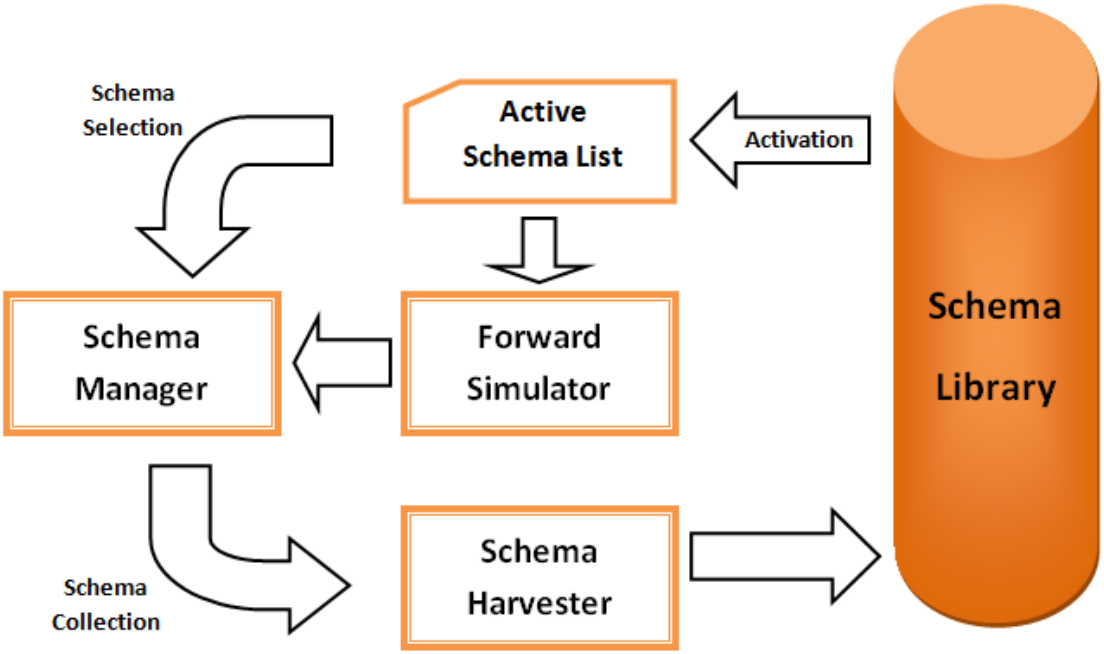

Figure 8 - Cognitive Architecture Diagram

Schema objects will be stored in the Schema Library, which will generate a list of all Schemas which can be activated given the current state of the world. The execution of Schemas will be managed by the Schema Manager, which will also obtain lists of future Schema activations from the Forward Simulator. The Schema Harvester will collect new Schemas and store them in the Schema Library. It is necessary for each of the described components to be included in the system.

5.1.2.1. Schema

Behaviours which the infant can perform in the world are to be stored as Schema objects; these objects are also used to record and store the behaviours the infant has learned.

Based on Piaget's representation, the Schema defined in this project is a triple (S1, Response, S2) comprising of an Initial State (S1), an action to take (Response), and the state resulting from this

action (S2). The initial and resulting states are a representation of the state of the world when they were recorded, and hence store a list of appropriate attributes.

5.1.2.2. World States

The states of the world will be held in the form of a single object, which will store the following information:

Attribute	Description
Eye Angle	The current angle of the eye/vision in the environment.
Fovea Distance	The current distance of the fovea from the centre of the eye.
Upper-Arm Angle	The angle the upper arm is currently making in relation to the shoulder.
Lower-Arm Angle	The angle the lower arm is currently making in relation to the upper arm.
Hand Angle	The angle the hand is currently making in relation to the lower arm.
In Contact	Whether the hand is currently in contact with an object or not.
Grabbing Item	Whether the hand is currently grabbing an item or not.
Near View	Whether an object is present in the area of interest in front of the eye.
Object In Fovea	The type of object present in the Fovea – 1 for hand, 2 for object, or 0 for nothing.
Seen Objects	A list of all the objects currently in vision, with their appropriate attributes.

Table 1 - State of World object contents

It was decided after careful consideration that the attributes above were a sufficient representation for the state of the world. These values provide enough information to determine the position of all significant objects in the world, and so enable the infant to make informed movements when running a Schema.

5.1.2.3. Response

The action to take in a Schema will be stored in the form of a Response object, which is required to have several variations. A Schema should be able to have any of the following objects as its Response:

- A basic Response, with some type of elementary action to take.
- A Network object, defining some sort of action to take, using a neural net.
- A list of sub-Schema objects, which can then be activated in a chain.

5.1.2.4. Schema Library

A Schema Library will be implemented as some form of data structure, storing all the Schemas which can be executed in the system.

Additional provisions will be in place to store Reflex Schemas; basic Schemas with only a Response, which must be present in the system at all times.

5.1.2.5. Schema Manager

A Schema Manager is required to manage the execution of individual Schemas, based on their Response type; hence, the method which manages the Schema will need to account for this.

5.1.2.6. Schema Harvester

A Schema Harvester is required to harvest new Schemas when a Schema is run. It will check the Schema Library object, to determine if a Schema is already present, and if not, will add it to the Library.

5.1.2.7. Forward Simulator

The Forward Simulator class will generate a Tree structure, used to provide the learning mechanism with a forward simulation of what could possibly happen given the execution of specific Schemas. The Tree will consist of several layers of Node objects, which contain a Schema and the state in which it was activated.

When the system is fully operational, this forward simulation Tree will be used to develop more complex behaviours, like moving an obstructing object out of the way to reach for another one, for example.

The Forward Simulator will also need to determine, by some means, the "best" possible branch of the Tree to take; allowing the learning mechanism to activate the Schema most likely to lead to success at each stage.

5.1.3. Transition Experience Generation

As part of the learning mechanism that will ultimately be integrated into the system, the testbed was required to provide Transition Experience generation for the learner's neural network. This was not one of the initial project requirements, but was added at the request of the project supervisor, and the student working on the learning section of the system.

A Transition Experience is a record of movement of the arm at a specific location. It is a 4-tuple, (s,a,s',r), consisting of an initial state (s), a response taken (a), a resulting state (s'), and a reward value for taking this response (r).

Transition Experiences, in their definition, look very similar to Schema objects, but differ significantly in their purpose. Transition Experiences are at a much lower level than Schemas, and it is from these objects that a Schema will be trained; to move the hand into the *Near View Area*, for instance.

A single Transition Experience displays the resulting state when one of four basic arm actions (upper-arm up, upper-arm down, lower-arm up, lower-arm down) is taken at an initial state. When repeated a large number of times in a variety of locations, it is possible to produce an indication of

what effects certain actions have in specific areas; hence aiding learning by means of a neural network.

Therefore, system is required to generate a large amount of Transition Experiences, saving them to a file for later use. The facility to load these Transition Experiences into memory is also required, to ensure they can be accessed easily afterwards.

An ideal Transition Experience generation method would sweep the arm across the entire available area in front of the infant, taking each of the four arm movements at every position. This would ensure that every possible location was covered. However, this simply would not be possible given a limited amount of time, and so it is sufficient to have the arm cover as much of the space as possible, allowing the learning mechanism to estimate the response in the gaps in between.

5.1.4. User Interfaces

One of the key differences between this system and the previous version is that the GUI is required to be more user-interactive.

As well as allowing user manipulation of the graphical display itself, the main GUI needs to provide a number of additional functionalities to the user, both for interacting with the graphics, and for helping the user to debug and further develop the learning mechanism.

The key aim in designing the user interface for this project was to try and keep everything as simple, and as intuitive as possible; each functionality needs to be clearly labelled, and visible to the user where they expect it.

The Robot GUI was designed using the NetBeans UI Designer tool, which allows the positioning of each individual aspect of the interface. The final version of the Robot GUI design is presented in Figure 9 below, with an explanation of design decisions underneath.

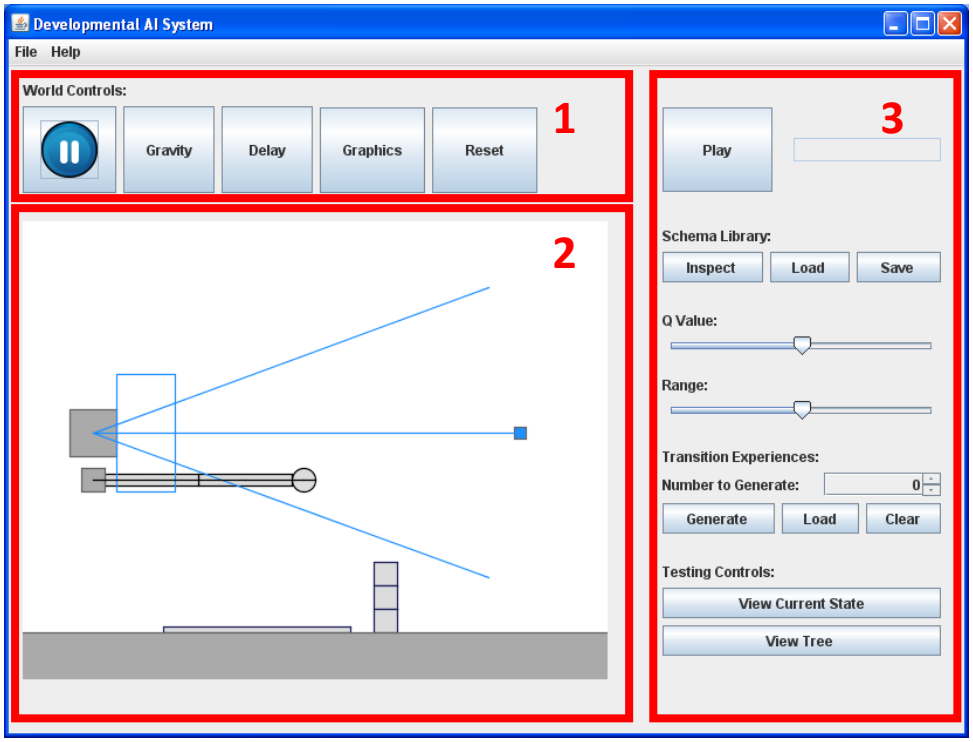

Figure 9 - Labelled Main GUI Screenshot

The Robot GUI was divided into three distinct sections during design; the controls for interacting with the graphics display (**1**), the graphical representation of the infant (**2**), and the tools for the learning section of the system (**3**). Grouping the GUI into these sections ensured that the required controls were immediately visible to the user, regardless of their previous experience with the testbed.

Additionally, Save/Load State controls are located in the *File* menu, as would be expected by the user in any window, and a brief user manual is provided under the *Help* menu. This user manual is used purely to provide details of keyboard controls for the system, because the system will only ever be used for research, and so full in-system user manual is not necessary.

As well as the main GUI, which displays the graphical representation of the infant, and its associated controls, an additional GUI was designed, to enable the inspection of the contents of the Schema Library.

The Schema GUI is provided as a pop-up window, displaying the current contents of the Schema Library, and providing the means to execute individual Schemas within it.

Like the Main GUI, the Schema GUI was designed with an emphasis on simplicity and intuitiveness, ensuring that any user could operate the functionalities it provides, regardless of their previous experience.

The final version of the Schema GUI design is displayed in Figure 10 below, with an explanation of related design decisions underneath.

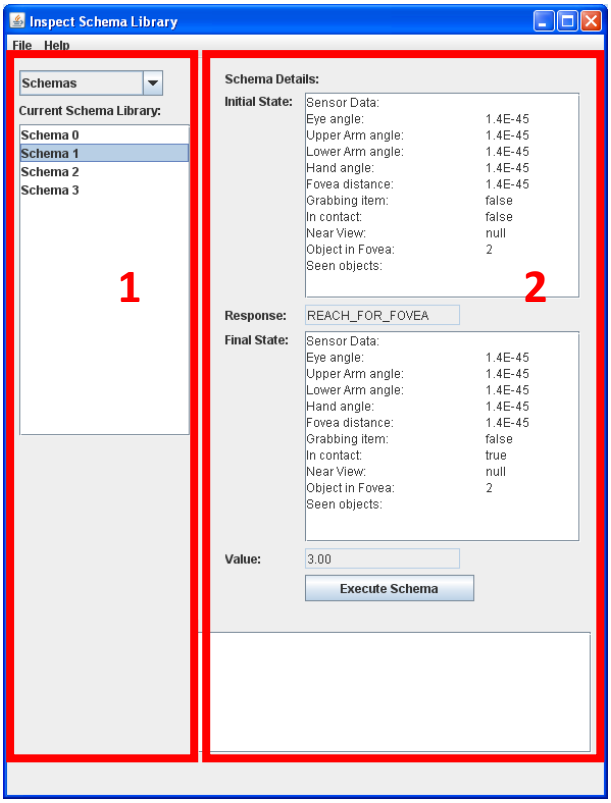

Figure 10 - Labelled Schema GUI Screenshot

The Schema GUI is divided into two main sections; the Schema display section (**1**) and the Schema inspection component (**2**).

The Schema display section allows the user to select the part of the Schema Library they wish to view (i.e. Schema list or Reflex Schema list), from which they can then select an individual Schema. The Schema inspection component allows the user to inspect the Schema they have selected, as well as execute it, and obtain results from the execution in the terminal box at the bottom of the window.

5.1.5. Data Storage

There are a number of situations in the system where some form of data storage is required; including the saving/loading of world states, the saving/loading of Transition Experiences during their generation, and the saving/loading of Schema Libraries.

As discussed previously, XML documents were used to store the required information; mainly due to the extensibility of tag creation. Hence, three different XML documents were designed to suit each type of data to be recorded; graphics display state, transition experience data, and Schema Library contents.

A sample XML document for each of the three functionalities mentioned are provided as part of Appendix C.

5.2. System Architecture

This section describes the overall architecture of the system, as well as the design of the packages contained within it.

5.2.1. Overview

While the system architecture in the previous system did not conform to any "standard" design pattern as such, the new system loosely follows the 3-tier architecture, with an interface layer, a processing layer, and a storage layer, as displayed in Figure 11.

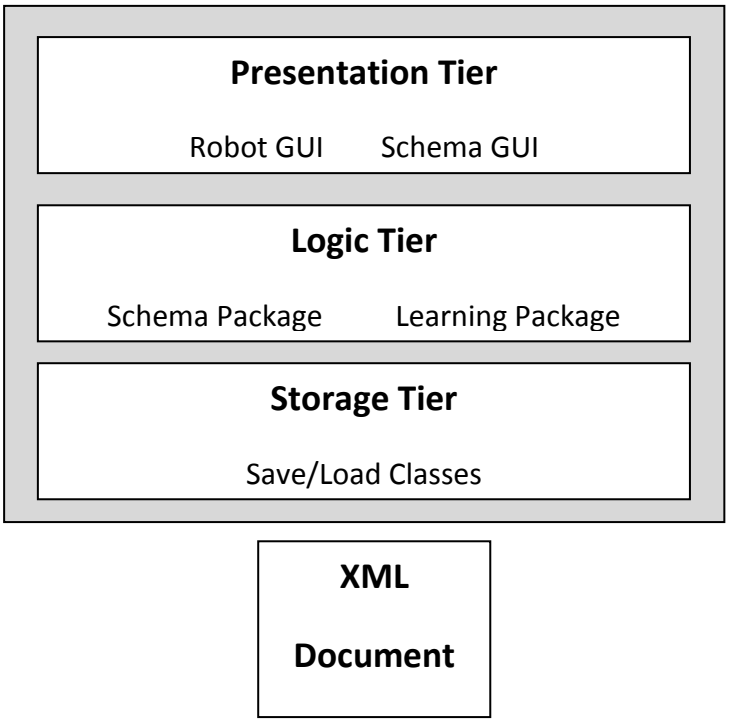

Figure 11 – System Architecture Diagram

Unlike the traditional definition of a 3-tier architecture, however, the Storage Tier in this project is not an integral part of the system. The data storage facilities were provided to allow the user to perform save/load operations at multiple points in the system, including the state of the world, transition experiences, and the Schema Library.

However, none of these features were key requirements, and were added mainly to allow for more in-depth testing of specific environmental scenarios and circumstances. Unlike most systems which use a 3-tiered architecture, this system would still function correctly without the Storage Tier; hence the project follows the 3-tier architecture only loosely.

5.2.2. Packages

The system was divided into a number of packages, with an emphasis on the organisation of related classes, to ensure that future work could be carried out with a minimum amount of inconvenience to the user.

Additionally, the primary aim was to create a more modular architectural design for this system than the previous version had, to prevent the new system from falling into a similar state of disrepair as before.

The key packages of the system are displayed in Figure 12, below, with an explanation of each package following underneath.

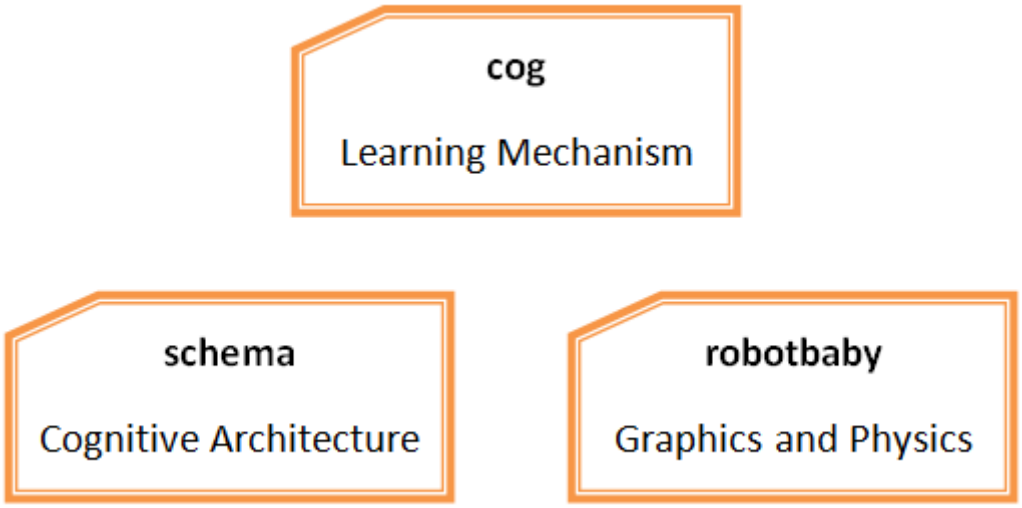

Figure 12 - Main Packages of System

The graphical components of the testbed part of the project, including the classes that produce the infant using the physics engine, are stored in the **robotbaby** package.

The cognitive architecture implemented is stored within the **schema** package, which defines the Schema object and its associated components, as well as classes for storing and managing Schemas.

The **cog** package will ultimately hold all of the classes associated with the learning section of the system. However, the majority of classes contained within **cog** were implemented by the student working on the learning mechanism, hence only a few classes were created for this package during this project.

The classes used to save and load data to XML files in the system are stored within **storage** sub-packages of each of the above main packages, acting as the storage tier of the architecture.

Chapter 6: Implementation

This chapter details how each part of the system was implemented, including how the design considerations discussed in the previous chapter were accounted for, and the problems encountered in doing so.

6.1. Graphics Representation

The robot infant and its world were created using the Processing 1.0 graphics renderer, along with the JBox2D physics engine, which provided suitable real-world features; including object collision, friction, gravity, and resting contact.

From the Processing graphics renderer, a ***PApplet*** object is produced, which is an applet object containing the aforementioned sketch of the infant and world.

The PApplet is refreshed continuously, at a designated frequency, and so all changes within the display are visible almost immediately. Therefore, every time an object is moved, like a block for example, the applet displays the each step of the movement instantly, in a similar style to an animation.

A considerable amount of time was spent researching and experimenting with the usage of JBox2D; initially, it was incredibly difficult to understand the demos provided, as no documentation was provided, and the sample code was sparsely commented. Hence, commencing the creation of the graphics display took substantially longer than the time initially assigned to the task, meaning the later stages of the project were delayed momentarily; however, it didn't take long for the project to get back on track.

The entire display can be altered manually, either using the keyboard controls provided for creating body movements (e.g. move upper arm up), or by clicking and dragging with the mouse. The keyboard controls use Key Listeners to detect changes, and the applet updates accordingly.

6.1.1. The Infant

The infant was created using the shape framework provided as part of the **collision** package in the JBox2D physics engine.

Once a shape has been defined, it is assigned to a ***Body*** object, part of the JBox2D **dynamics** package, which defines how objects in a world behave and react with other objects. The Body object allows for the customization of a number of attributes, including the frictional forces inflicted upon the object in the world, as well as its density and positioning.

Individual Body objects were created for each segment of the infant, as well as other objects, like blocks, and the boundaries of the world.

Each Body object can be assigned some form of user-defined data. Hence, in the display, each Body was given a String ID, allowing it to be identified at a later stage; this ID became particularly important when differentiating between objects during hand contact simulation.

6.1.2. Joints

After some experimentation with different methods of creating a realistic arm object, a solution was reached by attaching each arm segment using joints, which are provided in JBox2D as ***RevoluteJoint*** objects.

A RevoluteJoint object allows for two objects to be connected, and to limit the angle of movement each object has in relation to the other. This provided an ideal representation of shoulder, elbow and wrist joints, as it meant the relevant segments could be attached to one another, whilst limiting un-realistic movements.

The head and shoulder objects remain stationary in the world, as they are not required to move in the infant representation. The movements of the upper-arm, lower-arm and hand are all limited as are realistic – e.g. the elbow joint does not allow the lower-arm to bend backwards on the upper-arm.

6.1.3. Hand Contact

The JBox2D physics engine provides methods for detecting contact on objects. Each Body has a list of Contact objects, which store information on the other Body objects it is in contact with, if any.

From this information, it is possible to obtain the String ID of the Body in contact, as mentioned previously, and hence is possible to discriminate contact between different objects. For instance, it is possible to determine if the object in contact is a block, the floor, or another part of the infant's body, based on the String ID of the Body.

To provide continuous contact checking, the display has a method running in an individual Thread, listening for hand contact, supplying the infant with a "touch" ability; essentially providing a Contact Listener. The display can then detect when the hand collides with an object in the world, and if is possible, can then pick it up.

The sequence in Figure 13 below displays the process of picking up a block. First, contact (indicated by contact points, in red) is detected between the hand and the block (**1**). The contact detection method will distinguish that the item in contact is a block, and can so be picked up, with a joint between the two items being created (**2**). The block can then be moved around with the hand (**3**), without falling out of the hand's grasp.

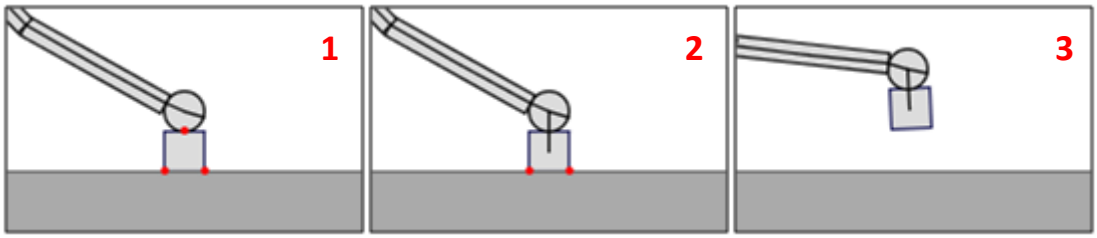

Figure 13 - Hand Contact Example Scenario

Implementing the method to detect touch took considerably longer than was planned; this was due mainly to the lack of any real examples of the Contact abilities JBox2D possesses, and so a number of attempts were made before reaching the solution detailed above.

6.1.4. Vision

The infant's vision indicators should not interfere with the rest of the world, and therefore could not be constructed using objects, or linear structures, which would cause an obstruction.

Hence, the vision display was implemented by drawing the appropriate features on top of the graphical display, as lines and boxes, a feature provided as part of the PApplet object.

It was decided before implementation that the infant should be able to see anything 20° above and below the direction it was looking straight at. Hence, using trigonometry, a triangle of vision was created, emerging from the infant's eye, located in the centre of the head object, as displayed in Figure 14.

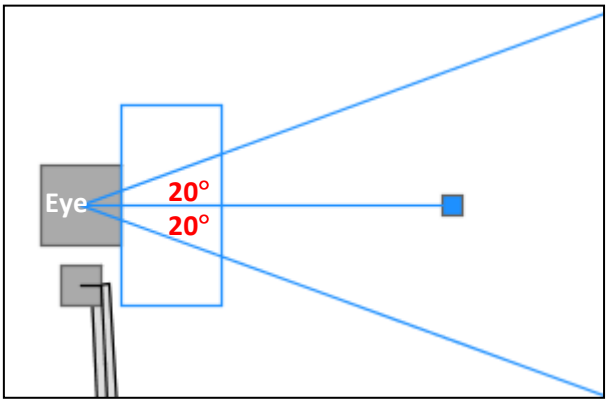

Figure 14 - Display of Infant Vision

By defining the "points" of vision using trigonometric formulae in relation to the other points, it was possible to enable the movement of the field of vision, without altering the actual shape of the triangle. Movement of the infant's field of vision was restricted, so that it cannot look behind itself.

Creating the vision indicators took a number of attempts to successfully complete, as it was difficult to draw the lines whilst retaining the same angle of vision during movement.

The point of focus, or fovea, of the infant was implemented in a similar manner. Indicated on the display as a box, it too is drawn on top of the graphics, with a line, leading from the eye to its

location. The position of the fovea was also defined in relation to the other vision points, meaning it could move, whilst staying in correct formation. Again, movement of the fovea is restricted, and it cannot move backwards beyond a short distance in front of the infant's face.

The vision indicators are updated as the display refreshes, in a similar manner to the movement of items in the world, meaning any change in position is immediately apparent.

The region immediately in front of the infant's head was implemented by drawing a box in a similar manner to the rest of the vision, but it has no movement; it is simply there to indicate the area of importance for the learner.

Both the current angle the eye is looking in, and the current distance of the fovea from the eye are recorded, and can be retrieved for use.

Functionality has also been provided to determine which object the infant is currently focusing on, by calculating the distance of each object from the fovea's location; if the distance is almost 0 (i.e. < 0.05), it is certain that the infant's focus is directed towards this object.

6.1.4.1. Seen Objects

When a Body falls within the infant's vision boundaries, it is "seen", and is stored as a **SeenObject** object.

A SeenObject has a String ID, a reference to the Body seen, the distance between the Fovea and the Body, and the angle of the Body relative to the eye.

The distance was calculated using the Mathematical distance formula, and the angle was obtained using trigonometry based on the angle of the eye.

6.1.5. Reaching

A key ability required by the infant is to reach for the fovea, or for objects the fovea is focussing on, and hence the representation of the infant was required to facilitate this.

The reaching behaviour was implemented by finding the position of the fovea, and then finding the position of the hand object relative to it. The arm then makes an appropriate movement towards the fovea, based on this position, and whether the fovea was within the hand's reach.

For instance (see Figure 15 - Reach Example Scenario, below), if the hand is positioned below the fovea with the fovea within reaching distance(**1**), the upper arm will be moved up until the hand lies on the fovea line(**2**), at which point the hand will be moved in until it touches the fovea (**3**). Similar movements are made when the hand is originally positioned above the fovea, or already on the fovea line.

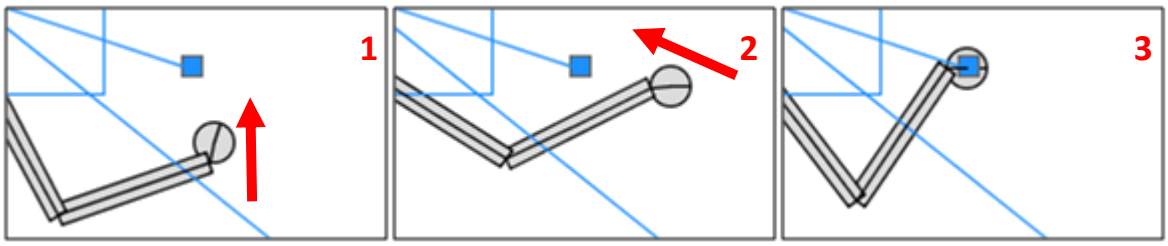

Figure 15 - Reach Example Scenario

The reach method will run the appropriate movement on a timer, so that it is displayed like an animation of the movement of the hand and arm, instead of just jumping into place.

If the fovea is out of reach, this will be recognised in the method, and the arm will stretch itself out and position itself on the fovea line, getting as close as it can to the object being focussed on.

A number of attempts were made at reaching for the fovea before the solution described above was arrived at. There were a number of difficulties in determining the position of the fovea relative to the hand, and even more so, determining the appropriate movements to apply to the arm to let the hand reach the position.

Eventually, the timer was used, which checks the position of the hand on a regular basis, meaning an arm movement can be made specific to the current circumstances, and not just a single, general movement.

6.1.5. Obstruction

One requirement of the learning mechanism is that the infant must have the ability to recognise when an object is obstructed by another object, so it can react accordingly. Hence, this functionality needed to be added to the graphics display.

This requirement was not present in the initial specification, but was added during the course of the project, when it became apparent this was a situation that was necessary to consider.

Obstruction detection was implemented by performing mathematical calculations upon the coordinates of the hand, the obstructing object, and the object being focussed on.

To determine if an object is obstructing the hand's reach capability, first, the angles of the obstructing object and the object focussed on with relation to the hand are first obtained. If the difference between these angles is negligible, i.e. they are almost aligned, the position of the object to be reached in relation to the hand is recorded.

Depending on the position of the object focussed on in relation to the hand, calculations are performed upon the obstructing object's coordinates to determine if it lies between them.

If the obstructing object lies between the hand and the object focussed on, and the angle between the two objects is negligible, then there is found to be an obstruction, and the infant will act as specified by the Schema. This is illustrated in Figure 16, below, which is assumed to be looking at the system from a top-down perspective.

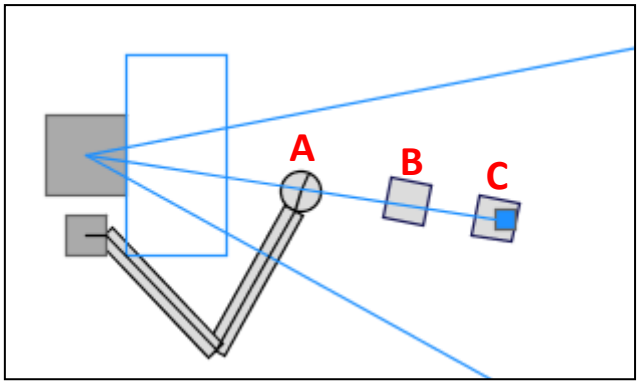

Figure 16 - Obstruction Example Scenario

Because B lies between A and C, and to the infant's vision the angle is almost identical, B is causing an obstruction, and this information will register in the system, when the method is called.

6.1.6. Body Movements

Each component of the arm (hand, upper-arm and lower-arm) has two associated methods which provide movement up and down. This movement is achieved by applying a torque to the required Body, with a standard rate of movement, causing it to rotate on its respective joint.

All of the methods for moving parts of the body – including the reach and vision methods – are stored in a single class, **BodyMovements**, allowing them to be referenced easily from any location in the project.

All of the Body movements, including the basic movements of the arm, reach, and detecting obstruction were required as part of the Response in a Schema object.

6.1.7. Calculations

A number of mathematical formulae were required for important calculations within the graphical display.

To calculate the distance between Body objects in the world, for instance, the distance between the fovea and the eye, the distance formula was used:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Also, to calculate the angle between a Body objects, or between a Body object and the eye, trigonometric functions were used, including sine, cosine and tangent. These functions were provided by using the Java Math package, which provides methods for making use of them.

6.2. GUI

The GUI was required to include a number of features, some of which were not present in the initial requirements specification, but were deemed to be significantly useful during the course of the project, if it was possible to incorporate them.

6.2.1. Integrated Graphics

In the previous system, the user interface was divided into two separate windows; one for the graphical display, and one for the Schema controls. Although no real problematic issues arose from this arrangement, it was apparent it would be more convenient for the user to have both elements present on a single GUI.

Hence, the infant representation was incorporated into the GUI by embedding the PApplet object storing the graphics onto the user interface itself.

6.2.2. Graphics Controls

The GUI needed to provide a number of functionalities to allow the user to manipulate and interact with the graphics display, to facilitate testing with the learning section of the system.

The **Pause** feature, allowing the user to stop and resume the operation of the graphics display was implemented by calling a method provided by the PApplet, freezing the refresh of the graphics display.

A **Reset** functionality, allowing the user to reset all objects in the display back to their original positions, was implemented by introducing a Boolean variable into the graphics class, which, when encountered to be true, restarts the graphical display.

The **Gravity** feature removes the gravity constraints set during the initialisation of the graphics object, hence allowing items in the world to be positioned freely, without falling.

Additionally, the **Delay** functionality was implemented by altering the frequency value at which the PApplet is updated. This removes any delay in updating the display, which is useful when testing needs to be carried out under specific time constraints.

As well as removing the delay from the graphics display, the **Graphics On/Off** feature was included to allow the removal of the graphical display itself from the GUI, for situations where processing power is more essential for the learning aspect of the system than updating and refreshing the graphics.

In order to allow the testing of specific scenarios a number of times, it was necessary to include a **Save/Load State** feature, which allows the user to save and load the positions of all objects in the world. This functionality is very important to the system, as ultimately it will allow the user developing the learning mechanism to run experiments from a specific state as many times as they require.

The **Save State** functionality was provided by writing the details of the world into an XML file using an Output Stream Writer object. The tags were as designed previously, and were printed out into an XML file specified by a File Chooser window.

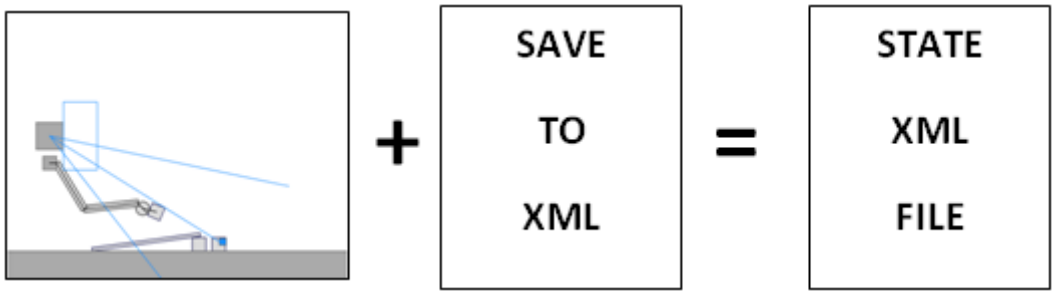

Figure 17 - Save State Functionality Summary

The **Load State** functionality parses the contents of an XML document, using DOM techniques, before applying the appropriate positions and rotations upon the objects in the world.

DOM was used to load the XML file data instead of SAX for a number of reasons, but mainly because DOM more easily accommodates accessing widely separated parts of a document at the same time. This feature was desirable for loading a Schema Library, for instance, where both the size and number of Schemas can cause the each section of Schema data in the document to be more separated.

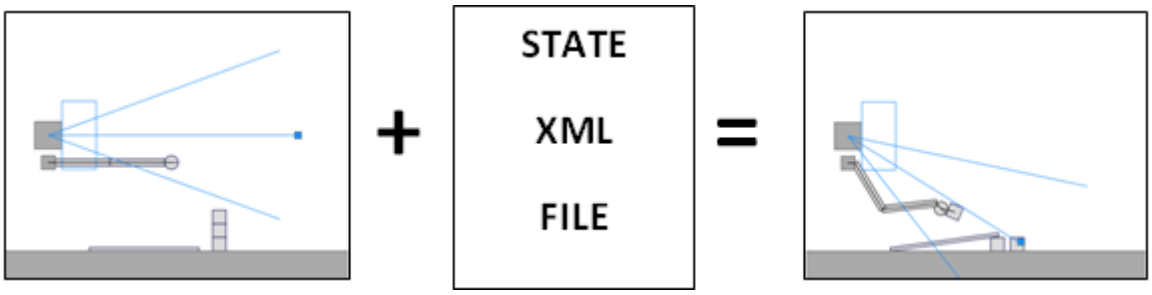

Figure 18 - Load State Functionality Summary

It is also worth noting that an **XML File Filter** class was created, to ensure only XML files could be saved to and loaded from, preventing any erroneous file types being used.

6.2.3. GUI Features

In addition to the graphics display, the user interface was required to offer a number of other functionalities to the user, including the facility to generate Transition Experience data, Save/Load Schema Libraries, and the ability to examine the current contents of the Schema Library, and inspect and execute individual Schemas.

Both the Transition Experience Generation and the Save/Load Schema Library facilities were implemented in a similar manner to the Save/Load State function, i.e. writing to and reading from XML documents using the DOM framework.

To examine the contents of the Schema Library, a secondary GUI was created, in the form of a pop-up window. The window contains a List object, which has an attached Listener, and hence the Schema display updates in correlation with the user's selection, which includes whether they have selected to examine Reflex Schemas or normal Schemas.

The Schema can then be executed by using a labelled button, which calls the Schema Manager object, running the Schema. A Text Area terminal is provided on the screen, and updates as the Schema is run, in order to inform the user of any issues arising from executing it.

"Play" functionality was included near the end of the project, which chooses a random Schema from the Schema Library, and runs it on the display. This was purely to demonstrate some of the infant's abilities.

Keyboard shortcuts were provided for all menu-toolbar operations available on the GUI, to ensure that they can be operated in a quick and easy manner.

6.3. Architecture

The system architecture was implemented as detailed in the Design chapter of this report, loosely following the traditional 3-tier architecture.

6.3.1. Architecture Details

The *Presentation Tier* is satisfied by the RobotGUI and SchemaGUI classes, which communicate with the classes in the *Logic Tier*, especially the RobotBaby class, which is required to update the graphical representation of the infant.

The *Storage Tier* is provided by the various Save/Load classes, which provide a layer between the data being written, and the storage location being written to; the XML files. However, as discussed previously, the *Storage Tier* does not follow the traditional concept of data storage, as it is not an essential part of the system.

All the packages that make up each tier contain only associated classes, providing a modular design, in an attempt to ensure that a specific class can be easily located by the user when required during future development of the project.

6.3.2. Startup

A **Startup** class was created to initialise all the required variables when the system is run. Additionally, when the system starts up, the Schema Library is prepared, and all handcoded Schemas are added.

The Startup class also initiates the Main Loop of the system, which runs the Forward Simulator and Schema Manager in a new thread, so they can be synchronised with the running of the physics engine.

6.3.3. Consts Class

After a number of classes had been created, it was visible that some variables were required to be available to the entire system, and not just the class they were instantiated in. Hence, a constants class, **Consts**, was created, storing all the common variables, with public and static properties.

The **Consts** class allows the variables it contains, like the Schema Library variable, **schemaLibrary**, for instance, to be called from anywhere in the system, without the need to instantiate the **Consts** class each time. As long as the variable has been assigned some value during the start-up process of the system, it can be retrieved by calling it in the format **Consts.schemaLibrary**.

Creating the **Consts** class was a key design decision early on in the development of the project, which made accessing important variables in the system a lot easier.

6.4. Cognitive Architecture

The Cognitive Architecture implemented in this project models infant behaviour by drawing inspiration from Piaget's *Theory of Cognitive Development*. The Cognitive Architecture is composed of the Schema objects as defined by Piaget, as well as a number of classes used to create them, and manage their execution.

6.4.1. Sensor Data

A **SensorData** object is a representation of the state of the world, and is defined as detailed in the Design section of the report; a variable is created for each of the objects listed in Table 1 in the class constructor, and the value is obtained from the graphics display. This object provides a Schema object with both the initial and final states of the world it requires by definition.

The list of objects seen by the eye is stored as an Array List of SeenObject objects, which are as discussed previously in section 6.1.4.1. Seen Objects.

The SensorData class also provides methods for state comparisons, which is used when determining if a Schema can be activated or not, based on the current state of the world.

An instance of a SensorData object is presented in Table 2, below, along with a corresponding image illustrating the state of the world at the point in time the data was recorded.

Attribute	Value
Eye angle	-32.5
Upper-Arm Angle	194.44934
Lower-Arm Angle	170.44048
Hand Angle	234.3257
Fovea Distance	29.69261
Grabbing Item	true
In Contact	false
Near View	false
Object In Fovea	2
Seen Objects	ID: HAND Angle:8.872747 Distance:11.48390 ID: BLOCK2 Angle: -0.1123466 Distance:0.0582226 ID: BLOCK4 Angle: 6.532818 Distance: 9.482213

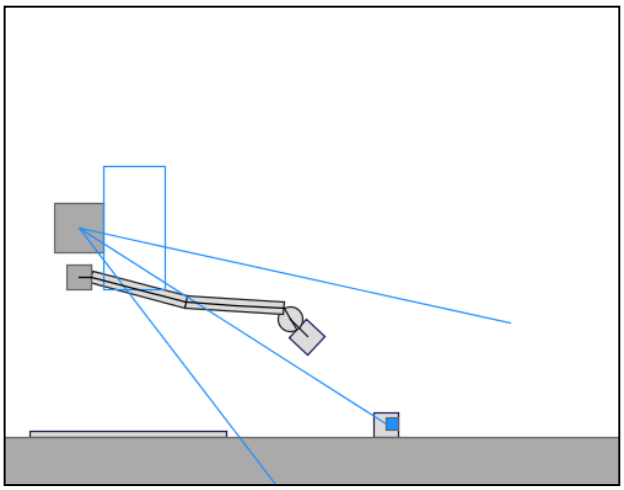

Table 2 - SensorData Object Example

As is visible from the image provided, the data held within the SensorData object provides an accurate representation of the state of the world at any point in time.

6.4.2. Response

The **Response** class is defined with three overloaded constructors, as discussed in the Design section in Chapter 5 of the report. Hence, a Schema can either have a ResponseType, Network (neural net object) or a list of Schemas as its Response.

A ResponseType is an enumerated value, with one existing for each basic action the infant can perform in the world, as illustrated in Figure 19, below.

```
public enum ResponseType{
    ARM_UP, ARM_DOWN, ARM_IN, ARM_OUT,
    HAND_UP, HAND_DOWN, HAND_GRAB, HAND_DROP,
    EYE_UP, EYE_DOWN,
    FOVEA_IN, FOVEA_OUT, FOVEA_FIXATE,
    LOOK_AT_OBJECT, PULL_HAND_IN,
    REACH_FOR_FOVEA,
    PULL_BLOCK_ALONG
}
```

Figure 19 - Response Type Enumeration

When a Response with a ResponseType is fired, a switch/case loop is called, which takes an appropriate action based on the ResponseType's value. This action may be simply moving a body part, like in the case of ARM_UP, for example, or performing a more complex action, like reaching for the fovea.

If a Response has a list of Schemas as its type, which is stored as an Array List, then, when the Response is fired, each sub-Schema is subsequently fired in order, in a "chain" effect. This allows for more complex Schemas to be created, which consist of multiple different moves; something required for learning more complex behaviours.

6.4.3. Schema

Like the Response object, the **Schema** class is overloaded with multiple constructors, allowing for different situations that a Schema may be executed in. A Schema can be of the "normal", 3-tuple form (Initial State, Response, Final State), a pair (Initial State, Response), or, for Reflex actions, just a single Response value.

When a Schema is fired, it returns a Boolean value, indicating if its execution was successful or not, based on the firing of its associated Response. This enables the Schema Manager to act accordingly, and display a suitable message to the user.

Near the end of the project, it was decided it would be beneficial to add extra features to the Schema object, in order to help the Forward Simulator determine the best Schema to execute. Hence, the Schema object was given multiple float values, indicating how interesting it was (based on the last time it was executed), its novelty (based on how long it had been present in the Schema Library) and its chances of success. These three values were then combined, adding them up to produce a single indicative value for the Schema.

Each Schema's value is altered as appropriate every time a Schema is run. For example, if a Schema is fired from the Schema Library, then its interest and novelty value decrease, and the interest value of all other Schemas in the library will increase, as they have not been executed recently.

A Schema uses its initial state to determine if it can be activated; if the current state of the world matches with the initial state of the Schema, then it can be fired. The example Schema presented in Table 3, below, illustrates such a situation.

Initial State		Response	Final State	
Eye angle:	1.4E-45	HAND_GRAB	Eye angle:	1.4E-45
Upper Arm angle:	1.4E-45		Upper Arm angle:	1.4E-45
Lower Arm angle:	1.4E-45		Lower Arm angle:	1.4E-45
Hand angle:	1.4E-45		Hand angle:	1.4E-45
Fovea distance:	1.4E-45		Fovea distance:	1.4E-45
Grabbing item:	false		Grabbing item:	true
In contact:	**true**		In contact:	true
Near View:	null		Near View:	null
Object in Fovea:	0		Object in Fovea:	0

Table 3 - Example Schema Object - Grab

In the scenario above, the Schema to grab an object can be activated on the condition that the hand is in contact with an object (displayed in **bold**). The angle and position values are all *null*, and hence are negligible when comparing to the current state of the world.

Therefore, the only condition required for this Schema to be activated is that, in the current state, the *In Contact* value is set to "true", i.e. the hand is touching an object. If this occurs in the current state, the *HAND_GRAB* Response will be fired, and the Schema's final state reached.

6.4.4. Schema Library

The Schema Library class contains two Array Lists, used to store "normal" and Reflex Schemas respectively. It provides methods for adding and removing Schemas, as well as completely emptying the Schema Library, to be used when loading a new Schema Library from a file.

Additionally, the Schema Library provides a method to retrieve all Schemas which can currently be activated, based on the current state of the world. When the method is called, given the current state, **s**, it runs through the Schema Library, obtaining all Schemas that have an initial state, $\mathbf{s}^i$, matching the current state (i.e. $s = s^i$). These Schemas are then returned as a list, making it easy to determine if a Schema can currently be executed.

6.4.5. Schema Manager

The Schema Manager class is used to manage the execution of Schemas when they are fired, either from the Schema GUI, or by the learning section of the system.

The main method in the Schema Manger class first determines how to run a Schema based on the type of its Response. If a Schema has a ResponseType, it will be activated only if it's initial state meets the current state of the world; in which case, the Response will be fired.

Similarly, if a Schema has a Network as its Response, the neural network action will be called if the initial state of the Schema matches the current state of the world.

If a Super-Schema is to be managed, i.e. its Response consists of a list of sub-Schemas, then each of the sub-Schemas will be called in order, controlled by a Java **Timer**, which manages each individually.

The method to manage Schemas returns a Boolean value to where it was called from, based on the success of executing the Schema; if the Response of the Schema failed to fire, false will be returned.

6.4.6. Schema Harvester

The Schema Harvester class will be called whenever a Schema is being executed, and will use machine learning techniques to generalise over past Schema patterns to add new Schemas to the Library.

However, this class will be fully implemented by the student creating the learning section of the system, and hence only a class framework is provided in this project.

6.4.7. Forward Simulator

The Forward Simulator class, when used in conjunction with the learning section of the system, will ultimately be used to determine the best Schema to activate at a particular point in the simulation.

At present, the Forward Simulator generates a Tree object, with Nodes of Activation objects, where each Activation is composed of a Schema activated, and the SensorData object of the state in which it was activated.

The Tree and Node classes were obtained from and are credited to the Internet blog of one Sujit Pal[16]. The classes used met the requirements of the solution perfectly, and hence were integrated into the system with only a few minor amendments.

The Tree structure is first given a root, consisting of a null Activation object with the current state of the world assigned to the final state of the Schema. The Schema Tree generation method then obtains a list of all Schemas which can be currently activated, based on the current state, from the Schema Library.

Using this list of Schemas, the Forward Simulator will select all those who can be activated given the final state of the parent node (which in the root case, is the current state of the world), and adds them as child nodes. This process is repeated a required number of times, as defined by a parameter; in this project, 4 levels of child nodes are produced.

Next, starting from the bottom child node, the Tree generation method then works its way up the Tree, adding the “interest” value from each child onto its parent node, until it reaches the first level of child nodes. From this point, using the total “interest” value of the branch of the Tree, it is possible to determine the best path to take; a method is provided to return the most “interesting” child node.

The following example indicates how a Forward Simulator generated Tree appears when the system has just been initialised. Because no Schema has yet been fired, all Schemas have a beginning value of 3.00 (see description in Schema section, above).

Hence, given the starting state of the world, and four basic Schemas (look, reach, grab, pull), the Tree is produced as in Figure 20, below.

```
1 → LOOK_AT_OBJECT     Interest: 18.00
    2 → REACH_FOR_FOVEA     Interest: 15.00
        3 → LOOK_AT_OBJECT     Interest: 6.00
            4 → REACH_FOR_FOVEA     Interest: 3.00
        3 → HAND_GRAB     Interest: 6.00
            4 → PULL_HAND_IN     Interest: 3.00
```

Figure 20 - Tree Generation Example Output

It is apparent that these results are feasible given the start state of the world; each of the parent Schemas can easily lead to its child, based on its final state. Additionally, it is clear to see that, because no Schema has yet been executed, the "interest" value of each node is equal, and totals up correctly.

In a more complex scenario, the Forward Simulator is able to pick the best path for the learner to chose, based on the value of the individual nodes.

6.5. Transition Experience Generation

As discussed in the Design section of this report, Transition Experiences were required to be generated in such a way that the maximum amount of area is covered by the arm, essentially "sweeping" the useful area.

After a number of attempts at Transition Experience generation, including entirely random movements of the arm, it became apparent that a more general solution was required; one which accounted for the fact that the arm could be in any number of positions when the generation begins.

The previous attempts produced reasonably scattered Transition Experiences, but on most occasions seemed to ignore some specific areas, either above or below the arm's reach. Additionally, random movements of the arm did not always guarantee that all forms of arm movement were being recorded in equal measures, something which is not ideal for the learner.

Hence, Transition Experiences are generated in the system according to the algorithm presented in Figure 21.

1. *for (number of required transition experiences)*
    - **1.1.** *Record starting State of the world*
    - **1.2.** *Move upper-arm up, record resulting State, add to list, reset to starting State.*
    - **1.3.** *Move upper-arm down, record resulting State, add to list, reset to starting State.*
    - **1.4.** *Move lower-arm up, record resulting State, add to list, reset to starting State.*
    - **1.5.** *Move lower-arm down, record resulting State, add to list, reset to starting State.*
    - **1.6.** *Move arm to entirely random position.*
    - **1.7.** *Repeat.*

Figure 21 - Transition Experience Generation Algorithm

Generating Transition Experiences in the above manner ensures that the Experience for each possible arm movement at each position is recorded, which is the ideal situation.

When carried out a large number of times, the above algorithm produces results similar to those displayed below, in Figure 22, which generated a total of 1,000 Transition Experiences.

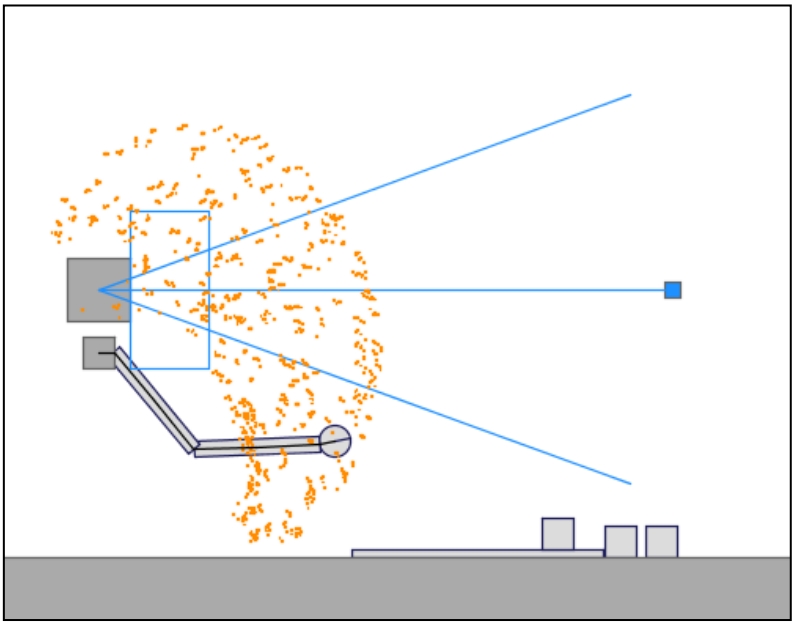

Figure 22 - Transition Experience Generation Example Screenshot

Each orange point on the display represents a Transition Experience generated. The points are gathered into clusters of 4, representing a Transition Experience for each arm movement from a default position.

Visibly, the algorithm used covers a large area of the available space in front of the infant. Although there are a number of white spaces between the points, the results produced are more than sufficient, as the learner will be able to estimate the results in the gaps, based on the rest of the generations. Additionally, the number of Transition Experiences generated can be increased, meaning it is possible to cover even more space, given more generations.

There are, however, a number of issues arising from the Transition Experience generation algorithm, which still need to be resolved.

First, as is visible from Figure 22, above, the hand occasionally strays into the head shape when being randomly repositioned, though only momentarily. This can be easily addressed by the learner, by disregarding any Transition Experiences generated within undesired areas.

Additionally, the Transition Experience Generation solution at the moment is not time effective, and can take up to 10 minutes when generating large amounts. However, there is no real solution to this issue, as speeding up the generation would lose a lot of accuracy, as the arm would be moving far too fast to record results correctly. Hence the speed of generation is as fast as is necessary.

Implementing the feature for generating Transition Experiences was a time-consuming exercise, but upon discussion with the student who will make full use of the feature, it is apparent it was a worthwhile requirement to fulfil.

6.6. Learning Schemas

As the final task of the project, it was necessary to implement some handcoded “cheat” Schemas, which would display some of the behaviours the infant will ultimately learn for itself using the learning section of the system.

A number of basic Schemas were created, displaying behaviours like reach, look, grab, and pull arm in, as well as a few master Schemas, chaining these together to display the full behaviour of obtaining an object.

The creation of these “cheat” Schemas made it visible that each part of the cognitive architecture was functioning correctly.

Details of the handcoded Schemas created are detailed in Appendix D.

6.7. Additional Tasks

At many points throughout the project, it was often necessary to expand on the original objectives, and work on implementing features which, while not actually included in the original requirements specification, were essential for the students working on the other two sections of the system.

Because the Developmental AI system had been divided into three individual sections, and a key aim of this project was to create a testbed for the system as a **whole**, extra requirements were frequently included an impromptu manner based on emerging requirements of the other two parts.

A lot of additional work was performed to ensure that the learning section of the system could slot into place with ease, often by adding extra features to accommodate its requirements.

For instance, generation of Transition Experience data, as discussed previously, was not an original requirement of this project, but was implemented about half-way through the development cycle because the learning neural network required the data it produces by necessity.

Additionally, a lot of work was carried out to expose variables and methods hidden within the graphics display for use by the learning mechanism. The **BodyMovements** class, containing all the methods to move the infant, was implemented with the key purpose of providing the learning section of the system with access to the movement methods.

As well as adding new requirements as the project unfolded, some of the initial requirements needed to be amended based on changes in the other sections of the system; for example, the Response class constructor being overloaded.

Because of the work entailed in accommodating these additional requirements and amendments throughout the development cycle, some of the original requirements of the project were omitted due to time constraints, most noticeably the more advanced hand-coded Schemas.

6.8. Summary

Overall, despite some initial setbacks in implementing the graphical representation, the system produced in the implementation phase of the project successfully manages to account for the majority of the original requirements.

However, due to time constraints caused by the delay at the beginning of the project, there are a few requirements which were not fully achieved, notably implementing handcoded Schema objects for displaying more complex behaviours.

A considerable amount of time was spent at the beginning of the implementation stage comprehending the usage of the physics engine and graphics package, due to the lack of any real documentation for the JBox2D API.

Additionally, as detailed previously, implementing additional features, and making amendments to the original requirements based on emerging needs of the other two sections of the system expended a lot of time as the project progressed.

However, it is clear that accommodating these additional requests was far more important to the system than producing example behaviours that would ultimately only be used for demonstration purposes. The additional features facilitate essential requirements of the other two sections of the system, making the future integration of the individual parts an easier process. Hence, the effects resulting from the requirements omitted have no real impact upon the potential successes of the system, and future work can continue unhindered without them.

Therefore, it is apparent that, despite not fulfilling all of the original requirements, the implementation phase of the project was successful, resulting in a system that is suitable for its intended purpose.

Chapter 7: Testing and Evaluation

This chapter details the testing carried out on the system during and after the implementation stage, and also evaluates the system performance. An evaluation of the system as a whole is also presented, alongside the bugs and problems known to exist in this version of the system.

7.1. Testing

This section details the testing strategies employed both during and after the implementation stage of the project.

7.1.1. Individual Component Testing

Extensive testing of the individual components of the system was carried out during the implementation stage of the project; this included the GUI functionalities, XML file saving/loading, and the graphics display. As each element of the system was created, it was tested thoroughly, to ensure it both met the requirements and was fit for use under any possible circumstances.

This enabled any problems or errors residing in each component to be discovered early, meaning there was less chance of the same errors occurring at a later stage in the development process.

Additionally, throughout its development, the system was constantly under operation, with a number of users, including the project supervisor and the two students requiring some parts of this project to supplement their own. Hence, the system was essentially team tested on a weekly basis, meaning any glitches or omissions were discovered in normal usage circumstances, and amended immediately.

7.1.2. Schema Execution Testing

Each element of the cognitive architecture, including the Schema object and its associated parts, needed to be tested thoroughly, as they provide the foundation for the core of the system, and so it is essential that they be error-free. Schema execution, for instance, will form the basis of the learning mechanism, hence comprehensive testing of this functionality was imperative.

The execution of Schemas in the system was tested by first creating a large, varied list of handcoded Schemas, which perform certain behaviours given specific states of the world. As mentioned previously, the handcoded Schemas which were created are detailed in Appendix D.

Each Schema was then tested extensively; it was first executed in a state of the world matching its initial state. This, by definition, should allow the Schema to execute successfully. The Schema was then executed in a state which differed from its initial state, which should prevent the Schema from being executed. These two tests were carried out for all of the sample Schema objects created, and the results are presented in Table 4, below.

ID	Schema Description	Initial State Matched	Activated	Final State Reached	Comments
A	Look at object	Y	Y	Y	
A	Look at object	N	N	-	Initial State not matched, so Schema not executed.
B	Reach for Fovea (Object focused on)	Y	Y	Y	
B	Reach for Fovea (Object focused on)	N	N	-	Initial State not matched, so Schema not executed.
C	Reach for Fovea (no object focussed on)	Y	Y	Y	
C	Reach for Fovea (no object focussed on)	N	N	-	Initial State not matched, so Schema not executed.
D	Reach for Fovea (hand focussed on)	Y	Y	Y	
D	Reach for Fovea (hand focussed on)	N	N	-	Initial State not matched, so Schema not executed.
E	Grab object	Y	Y	Y	
E	Grab object	N	N	-	Initial State not matched, so Schema not executed.
F	Drop object	Y	Y	Y	
F	Drop object	N	N	-	Initial State not matched, so Schema not executed.
G	Pull hand in (object grabbed)	Y	Y	Y	
G	Pull hand in (object grabbed)	N	N	-	Initial State not matched, so Schema not executed.
H	Pull hand in (no object grabbed)	Y	Y	Y	
H	Pull hand in (no object grabbed)	N	N	-	Initial State not matched, so Schema not executed.
I	Pull Object Along Ground	Y	Y	Y	
I	Pull Object Along Ground	N	N	-	Initial State not matched, so Schema not executed.
J	Look at object, reach for object (object focussed on), grab object, pull hand in (object grabbed)	Y	Y	Y	
J	Look at object, reach for object (object focussed on), grab object, pull hand in (object grabbed)	N	N	-	Initial State not matched, so Schema not executed.

Table 4 - Schema Execution Test Results

It is visible from these test results that in each circumstance, the Schema Manager functioned correctly, meaning the mechanism for executing Schemas worked successfully in all of the test cases.

7.1.3. User Testing

As discussed previously, the system was used on a regular basis by multiple users working on different parts of the project, so a number of problems were discovered by these users, and subsequently amended, during the implementation of the project.

Additionally, after completion, the system was tested extensively by another student who was not part of the project.

Provided with the User Manual for the system, the user tested each functionality of the testbed, and ensured that no problems occurred from extensive usage. No significant issues arose from this final testing stage, and the user was able to operate the system without any difficulty.

7.1.4. Scalability Testing

The system will often be required to store copious amounts of data in multiple Array Lists; specifically the Schema Library contents and Transition Experience data. Hence, it was necessary to ensure the system would be able to perform correctly with large amounts of data stored in these structures.

Both the Schema Library and Transition Experience Array Lists were loaded into the system with an unrealistically excessive number of elements, to test the systems performance under strenuous conditions.

After repeated testing, which included loading the Array Lists individually and together, there were no noticeable signs of strain from the system, and it functioned as normal.

It can therefore be concluded that the system provides scalability to the user, in terms of the amount of data that can be loaded. Because it can perform unaffected under extreme conditions, the system should therefore function under normal usage without any issues.

7.1.5. Speed and Efficiency

Whilst the speed and efficiency of the system were not a primary concern, it was very important that they be considered during the implementation of the project.

Because the system makes use of a powerful physics engine, as well as using Array Lists, like the Schema Library, which grow exponentially as it runs, it is understandable that the system's performance will alter over time.

After a significant amount of rigorous testing, and running the system many times throughout the implementation cycle, it is apparent that the speed and efficiency of the system are not noticeably affected in most cases. On nearly every occasion, the system never noticeably lagged or froze, unless working in situations requiring extreme amounts of processing power.

For instance, when actually generating Transition Experiences (i.e. not just loading the data into an Array List from an XML file as discussed in the previous section), if the amount to be generated is

significantly large, greater than 5000, the system lagged noticeably towards the end of the generation. This is due to a number of factors which consume both the allocated memory and processing power of the system.

When a Transition Experience is generated, both the Transition itself, and the coordinates of the Transition are stored in separate Array Lists; these lists will grow exponentially as the generation runs. Additionally, the graphical display is constantly updating, calculating a new position for, and moving, the arm, as well as displaying the location of each Transition Experience on the screen. Hence, the system does not have the full processing capabilities it would usually be allocated, and is therefore visibly slower until the generation sequence is complete.

However, such high performance operations will never be required under normal usage of the system, and so it is not necessary to account for the system lagging in such circumstances.

7.2. Evaluation

This section provides an evaluation of the system implemented, including some criticism of the current design and suggestions for improvements.

7.2.1. Overall System Evaluation

Overall, the system implemented provides a satisfactory solution to the requirements of a Developmental AI system.

After discussion with those who will ultimately make use of the system as a foundation for the learning mechanism, it is apparent that the facilities provided by the testbed fulfil the requirements, and will be used extensively in future development.

The cognitive architecture created also satisfies the required conditions, providing a valuable framework for supporting the learning section of the system. During the later stages of the project, the learning mechanism was successfully integrated with the cognitive architecture, with only a few minor amendments required, proving the cognitive architecture is suitable for its purpose.

Furthermore, the architecture implemented in this system is both modular and extensible; enabling straightforward future development in comparison to the previous version. Many of the objects contained within the architecture were implemented with flexibility in mind; the Schema and Response objects, for example, have overloaded constructors, allowing them to be customized for a variety of situations. The Schema Library data structure is also flexible, providing storage for any type of Schema object, regardless of its composition.

Ensuring flexibility within the objects and data structures of the system means the system can be more easily adapted in the future, without too much additional work; hence aiding extensibility.

However, like all Developmental AI systems, there are a number of aspects which could draw criticism, and could be improved upon in order to provide as accurate a solution to the problem as possible.

For instance, the system implemented only makes use of 2D graphics to represent the infant and its world; it could be argued that 3D graphics would provide a more realistic simulation, although this may not have been as viable an option given the timescale of this project. A 3D graphics display would visibly portray a more life-like representation of an infant than a 2D display, but consequently there would be a number of additional factors to consider in its creation.

Also, at present in the graphics display, grabbing an object simply involves creating a joint between the hand and the object to be grabbed. This again, it could be argued, is not realistic, and hence adding "finger" objects and the ability to grasp would provide a more accurate depiction. This, however, would require a far more complex representation of the hand, with significantly more objects and joints to consider, something which again would not have been a viable option, taking the time allocated to the graphics display into consideration.

Additionally, the infant's vision in the current system is based on what we know about the 2D world; i.e. the coordinates of objects relative to the eye. This clearly is not a realistic representation of an infant's vision, and is in effect a "cheating" way of reproducing it. However, as discussed previously, the implementation of a more realistic vision system for the infant is currently taking place as part of another project, with the aim to integrate it with this system in the future.

While the current system provides a Developmental AI system by using a graphical simulation, one could argue that this is not necessarily the best possible depiction, as it's not "real" enough; there are a number of systems which provide a far more realistic representation of an infant.

The RobotCub project as discussed previously, for example, studies infant cognition using a humanoid robot representing a 3.5 year old child. Visibly, this project provides a more realistic simulation than a 2D graphics display, and hence may result in more innovative findings.

However, this clearly was not a feasible option given the resources and time assigned to this project; creating a representation like the iCub would take a number of years of research, and would require a lot of advanced technology.

Although a number of criticisms can be drawn from the system implemented, given the timescale of the project, and the facilities provided, the system provided is more than satisfactory, and can be used for continued research for the foreseeable future.

Although it is clear the system could be improved upon in a number of respects, given the timescale of the project, and the facilities allocated, the solution implemented is more than satisfactory when the requirements of the project are taken into consideration.

7.3. Known Bugs

There are a number of known bugs in the system which still need to be addressed:

- When generating Transition Experiences, due to the arm repositioning techniques used, the hand will occasionally stray into the head object, or will get caught between the head and the neck.

These Transition Experiences can be easily ignored when the generated data is used, but a more ideal solution would be to limit the hand's movement in these areas when a Transition Experience generation is active. However, limiting the hand does not provide a true representation of the arm movements in the system, so this issue would need to be addressed carefully.

- Also when generating Transition Experiences, because the arm is repositioning at such high speed, occasionally the arm segments will detach momentarily; this however, does not affect the results produced, and is purely a cosmetic glitch.

- If the hand reaches for an object it wishes to grab too quickly, then on some occasions, it will bounce away from the object, losing contact.

  Preventative measures were taken to ensure this does not happen, decreasing the restitution ("bounciness") of the hand object, but can still occur occasionally.

Chapter 8: Summary and Conclusion

This chapter summarises and discusses the system implemented during this project, detailing future development, as well as providing a conclusion to the project as a whole.

8.1. Summary

Looking over the initial project requirements, it is visible that the system implemented manages to successfully fulfil almost all of them.

- **Replacing the current physics engine with a more advanced one.**

  This was the first major task of the project, and was carried out successfully; the JBox2D physics engine was used, replacing the existing engine, and providing more advanced features.

- **Implementing a graphical representation of an infant in a world, with user controls.**

  This task was achieved using the JBox2D physics engine, as discussed previously, and the Processing 1.0 graphics renderer, to produce the image.

  Both keyboard and mouse controls were added to the graphical display, providing the user the facility to manipulate the world as they require it, for running experiments with the learning mechanism.

- **Revamping the system architecture with an emphasis on extensibility and modularity.**

  The system architecture was re-designed, ensuring packages and classes were as extensible and modular as possible. This aids development of the system in the future, and prevents the system reverting to the state of the previous version.

- **Creating a cognitive architecture framework, allowing the learning and vision sections of the system to be integrated.**

  A cognitive architecture was created, based on Piaget's theorised Schema object, as detailed in his *Theory of Cognitive Development*. Both the Schema object itself, as well as the classes to create, store and manage Schemas were implemented during this system.

  The learning mechanism can easily be integrated into the system, making use of the cognitive architecture provided.

- **Creating a Graphical User Interface (GUI), which provides a number of functionalities to the user allowing them to stage experiments with, and debug, the learning mechanism.**

  The GUI implemented provides a variety of features which can be used to both stage experiments and develop and debug the learning mechanism.

Facilities to repeat specific states of the world, as well as inspect the inner data structures of the Schema Library are provided, allowing the user to determine exactly what has been learned by the system thus far, and act on this information accordingly.

- **Displaying basic, elementary behaviours using handcoded Schemas.**

  This requirement was achieved, although not to the extent initially hoped. At the beginning of the project, it was anticipated that, as well as some basic behaviours, it would be possible to enable the system to display more complex behaviours, like pulling objects closer with a stick.

  However, due to time constraints, these complex behaviours were not implemented. Nonetheless, handcoded Schemas for the more basic behaviours are included in the system, like looking, reaching, grabbing then pulling an object closer.

As well as the initial requirements, the project managed to accommodate a number of additional features requested by the project supervisor, and the students working on the other sections of the system. New requirements were added in a desultory fashion throughout the project, including illustrating the infant's field of vision, accommodating for neural network objects, and providing facilities for the generation of Transition Experience data.

These additional tasks, however, did interrupt the original project plans, resulting in some of the initial requirements being omitted. Also, a number of bugs remain present in this version of the system that have yet to be amended due to a lack of time in the closing stages of the project.

Nonetheless, the project meets almost all of the original requirements, along with accommodating a number of additional features that were not originally required, resulting in a system that will be used for Developmental AI research for the foreseeable future.

8.2. Future Development

A considerable amount of work will be undertaken both on, and using, this system in the future. The system will provide a foundation for the learning and vision sections of the Developmental AI system, and hence parts of it will be refined in the future as new requirements become apparent.

The following tasks will all need to be achieved at some point, in order to progress with the creation of a Developmental AI system:

- **Integrating the vision and learning sections of the system.**

  Integrating the vision and learning packages into the system will not be an arduous task; both sections already make use of this system as part of their individual projects, and hence they have already been integrated to an extent. All that is required is to combine all three parts together into a single system.

- **Upgrading the graphics display to use JBox2D 2.0.1.**

  The newest version of JBox2D was made available from the developer's website a few weeks after the implementation phase began. However, a lot of classes and features have been altered considerably, so updating to the newer version during this project was not a viable option given the time learning the previous version had already consumed.

  Updating to the new version is not entirely necessary; the current version sufficiently provides all the features required for this project. However, it is beneficial to the project in the long term to keep the physics engine up-to-date.

- **Facilitating the saving and loading of Schemas with Network objects as their Response.**

  Saving and loading Schema objects with a neural network as a response has not been implemented in the current system, due to the Network object, developed as part of the learning project, being of a format that was unsuitable for saving to an XML document.

  Hence, at the moment, Network instances are stored as synchronised objects, and loaded into the system when required. However, they cannot be saved or loaded as part of a Schema in a Schema Library, a facility which will need to be implemented in the future, for user convenience.

- **Implementing the Schema Harvester class functionality.**

  As discussed previously, the Schema Harvester class currently only provides a template in the system; it will need to be expanded and fully implemented, making use of machine learning techniques to determine which Schemas to store in the Library.

8.3. Conclusion

In conclusion, this project was evidently successful in its aims to implement a cognitive architecture and testbed for a Developmental AI system. The system developed fulfils the requirements of the project, as well as including additional functionalities for the user not originally requested.

The testbed provides the facilities for developing and debugging the learning section of the system, the importance of which cannot be understated. These functionalities will ultimately enable the learning mechanism to be further refined, and hence become more advanced; this will hopefully result in significant findings towards the research of Developmental AI systems.

The implementation of the system was, in itself, not an easy process; a range of challenges were encountered, and eventually overcome, resulting in both the completed software, and a wealth of knowledge gained from the experience.

A extensive number of new techniques, skills, and technologies were learned throughout the duration of the project, which will undoubtedly prove useful as the system is developed further; this knowledge will be passed on to the future developers of the system, ensuring the prospective work on it is an uncomplicated procedure.

The system implemented in this project will undoubtedly be both utilized and further developed in the future. With plans for further work already in place, it is sure to provide the foundation for a Developmental AI system that will be used to support research for years to come.

References/Bibliography

[1] Guerin, F., 2010. *Learning Like Baby: A Survey of AI approaches,* Knowledge Engineering Review, to appear.

[2] Jean Piaget Society. 2007. *About Piaget.* [Online] Available at: http://www.piaget.org/aboutPiaget.html [Accessed 28 October 2009].

[3] Piaget, Jean, 1936. *The Origin of Intelligence in the Child*, Routledge, 1998.

[4] Lehman, J. Laird, J. & Rosenbloom, P. 2006., *Soar: A Gentle Introduction.* [Online] Soar. Available at: http://ai.eecs.umich.edu/soar/sitemaker/docs/misc/GentleIntroduction-2006.pdf [Accessed 05 February 2010].

[5] Anderson, J.R. et al., 2004. *An Integrated Theory of the Mind*. [Online] ACT-R. Available at: http://act-r.psy.cmu.edu/papers/526/FSQUERY.pdf [Accessed 05 February 2010].

[6]Langley, P. & Choi, D., 2006. *A Unified Cognitive Architecture For Physical Agents*. [Online] ICARUS. Available at: http://cll.stanford.edu/~langley/papers/icarus.aaai06.pdf [Accessed 05 February 2010].

[7]Laird J.E. Rosenbloom, P.S. & Newell, A., 1986. *Chunking in Soar: The Anatomy of a General Learning Mechanism*. [Online] SOAR. Available at: http://www.springerlink.com/content/h568q43948512071/fulltext.pdf [Accessed 05 February 2010].

[8] Schoppek, W., 2003. *Emerging structure in ACT-R: Explorations in competitive chunking.* [Online] ACT-R. Available at: http://act-r.psy.cmu.edu/publications/pubinfo.php?id=471 [Accessed 05 February 2010].

[9] CogX. 2008. *CogX Home*. [Online] Available at: http://cogx.eu/ [Accessed 06 February 2010].

[10]Metta, G. et al., 2008. *The iCub humanoid robot: an open framework for research in embodied cognition*. [Online] RobotCub. Available at: http://www.robotcub.org/index.php/robotcub/about_us/robotcub_paper [Accessed 06 February 2010].

[11] JBox2D. 2008. *JBox2D Demos.* [Online] Available at: http://www.jbox2d.org/ [Accessed 02 February 2010].

[12] JBox2D SourceForge. n.d.. *Get JBox2D at SourceForge.net.* [Online] Available at: http://sourceforge.net/projects/jbox2d/ [Accessed 02 February 2010].

[13] Box2D. 2007. *Box2D v2.1.0.User Manual*. [Online] Available at: http://www.box2d.org/manual.html [Accessed 02 February 2010].

[14] Processing. 2009. *Processing.org.* [Online] Available at: http://www.processing.org/ [Accessed 02 February 2010].

[15] Subversion. n.d.. *Apache Subversion*. [Online] Available at: http://subversion.apache.org/ [Accessed 15 February 2010].

[16] Pal,S., (26 May 2006) *Java Data Structure: A Generic Tree*. [Online] Sujit Pal's Internet blog. Available at: http://sujitpal.blogspot.com/2006/05/java-data-structure-generic-tree.html [Accessed 28 March 2010].

Appendix A:
User Manual

Appendix A: User Manual

The User Manual details how to operate the Developmental AI system interface, including instructions for all of the functionalities it provides.

A.1. Installing the System

There are a number of ways the system can be installed and run on your computer. The action to take depends on the format of the copy of the system you posses. This will be either the Java project folder, **RobotBaby** or the jar file **RobotBaby.jar**.

A.1.1. Running System from JAR File

If you are in possession of the system in jar file format (i.e. RobotBaby.jar), simply copy the file to the location where you want to store it on your computer, for example, the Desktop.

When the file has been copied to a location on your computer, to run the system, simply double click the jar file icon, displayed below in Figure A.1.

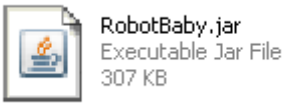

Figure A.1 - Jar File Icon

A.1.2. Running System in an IDE

If you are in possession of the system in Java project folder format (i.e. RobotBaby), you will need to run the project from a suitable IDE (Integrated Development Environment), like the latest version of NetBeans or Eclipse, for example.

When you have obtained a suitable IDE, simply copy the Java project folder **RobotBaby** into the directory where the IDE stores its associated projects, for example, the *NetBeansProjects* folder, if you are using NetBeans.

Next, open the IDE, and open the **RobotBaby** as a new project, using the controls provided by the IDE (Help and documentation will be provided with the IDE to assist you with this process.)

To run the project from the IDE, when it has been opened, either right-click and select “Run”, as illustrated in Figure A.2, below, or press the appropriate “Run” icon on the IDE’s interface.

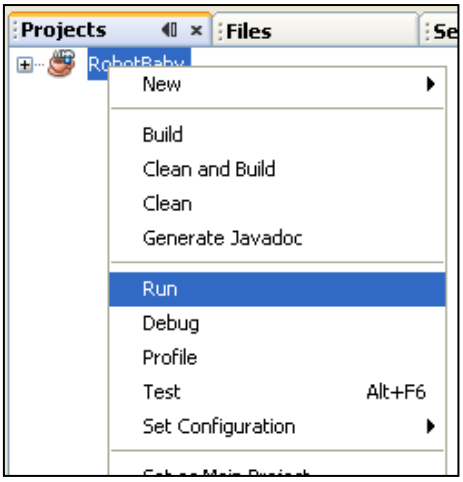

Figure A.2 – Running Project in IDE

A.2. Getting Started

Whichever method of running the project is used, you will then be presented with the user interface as displayed in Figure A.3.

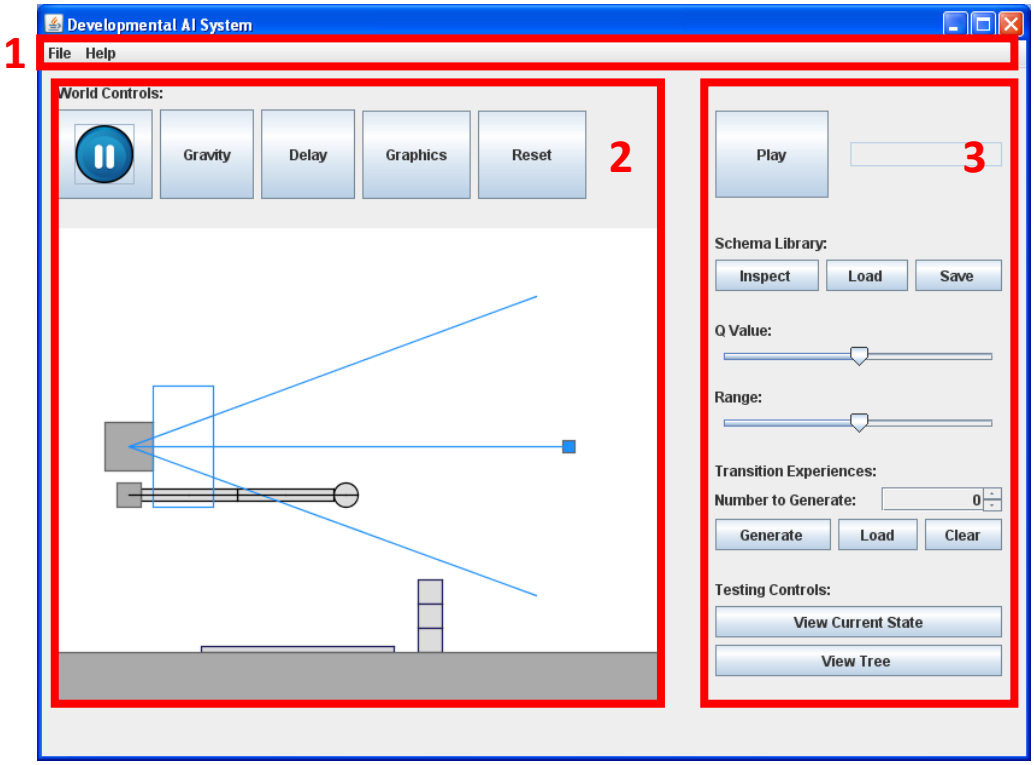

Figure A.3 – Main User Interface

This is the main user interface, from which the entire system can be operated. There are three main sections to the user interface, all of which will be mentioned throughout the rest of this manual:

1. The Menu Bar.
2. The Graphics Display and World Controls.
3. The Learning Tools and Testing Controls.

In the following sections, instructions for each part of the interface will be presented.

A.2.1. Exiting the System

If you wish to exit the system at any point, it can be closed either by clicking the red cross on the top right hand corner of the window, or by selecting the *Close* option from the *File* menu on the menu-bar, as illustrated in Figure A.4. The close option can also be selected by using the key combination *Ctrl + Q*.

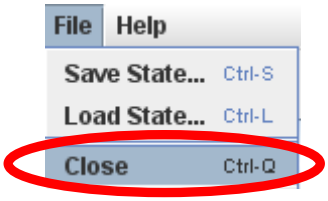

Figure A.4 – Close Option

A.3. Using the Graphics Display

The graphics display provides the representation of the infant and its world. It is possible to manipulate and interact with the display in a number of ways.

A.3.1. Mouse Interaction

To interact with the graphics display using the mouse, simply point and left-click at the object you wish to re-position, holding the left mouse button down for the duration of the interaction.

For instance, if you wish to move a block to a new location, left-click on it with the mouse, and hold the left button down. Now, move the block to its new location using the mouse, and let go of the button when you have finished making the movement.

All objects in the world can be manipulated using the mouse, except those displayed in dark grey, which are stationary objects, and hence cannot move.

A.3.2. Keyboard Controls

To interact with the graphics display using the keyboard, simply use the keys detailed in the table below to make the required movement.

Key	Action
1	Move Upper-arm up.
2	Move Upper-arm down.
3	Move Lower-arm up (in).
4	Move Lower-arm down (out).
5	Move Hand up.
6	Move Hand down.
7	Grab object.
8	Release object.
9	Move vision up.
0	Move vision down.
i or I	Move Fovea in.
o or O	Move Fovea out.
r or R	Reset the display.
c or C	Display contact points.

These keyboard controls can be displayed within the system, at any time whilst it is running, by selecting the *User Guide…* option from the Help menu on the Menu Bar, as illustrated in Figure A.5 below, or by using the key combination *Ctrl + H*.

Figure A.5 – User Guide Option

A.3.3. World Controls

The *World Controls* toolbar, presented in Figure A.6, is located above the graphics display, and provides a number of features for altering or interacting with the display itself. Each feature can be activated by pressing the appropriate button.

Figure A.6 – World Controls

The *Play/Pause* feature (**1**) can be used to pause the graphics display, holding all objects in position until it is pressed again.

To turn on/off the gravity in the world the infant is located in, simply click the *Gravity* button (**2**). This will allow objects to be placed freely across the display, without falling down, until gravity is turned back on, by pressing the button again.

To turn on/off the drawing delay in the graphics display, click the *Delay* button (**3**), which speeds up the drawing process until delay is turned back on, by pressing the button again.

To remove the graphics display from the GUI completely, click the *Graphics* button (**4**), which removes the display until the button is clicked again, at which point it will return.

To reset all objects in the infant's world back to their original locations, simply click the *Reset* button (**5**), which will restart the graphics display.

A.3.4. Saving & Loading States

The user interface provides the facilities to save and load states of the graphics display to and from XML files.

A.3.4.1. Saving States

To save a state of the graphics display, first position each object in the infant's world to how you require it to be, using either keyboard or mouse manipulation.

Next, click on the File menu on the Menu Bar, and select the *Save State...* option, as illustrated in Figure A.7. This can also be achieved by using the key combination *Ctrl + S*.

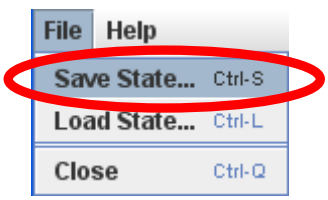

Figure A.7 – Save State Option

You will now be presented with the window illustrated in Figure A.8.

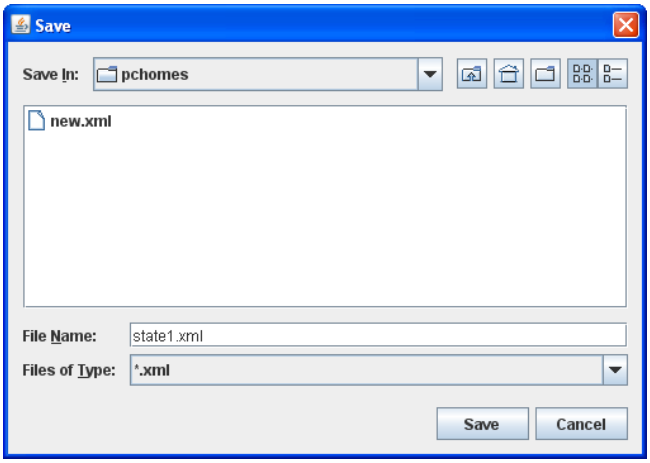

Figure A.8 – Save State Window

To save the state, select an existing XML file, or create a new file, giving it the extension ".xml", e.g. "state1.xml".

Now, click *Save*, and the state of the graphics will be saved to the file specified.

A.3.4.2. Loading States

NOTE: When you load a previously saved state onto the graphics display, you will lose the current state, unless it has already been saved.

To load a previously saved state onto the graphics display, click on the File menu on the main menu bar, and select the *Load State...* option, as illustrated in Figure A.9. This can also be achieved by using the key combination *Ctrl + L.*

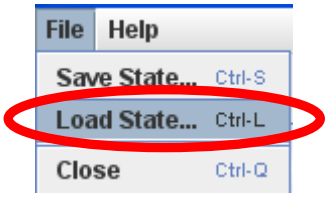

Figure A.9 – Load State Option

You will now be presented with the window illustrated in Figure A.10.

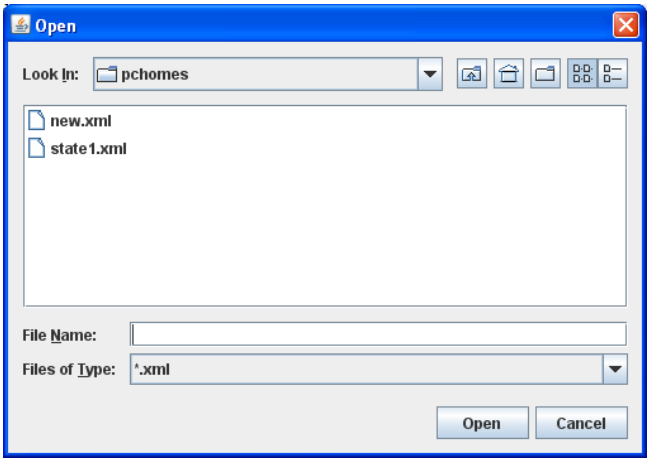

Figure A.10 – Load State Window

To load a state, first select the XML file you wish to load.

Now, click *Open*, and the state of the graphics will be updated to display the contents of the file.

A.4. Using the Learning Tools

The Learning Tools provide functionalities to display behaviours on the graphics display, as well as generating data for the learning mechanism. It also allows the user to load Schema Libraries into the system.

A.4.1. Saving & Loading a Schema Library

When the system starts, an example Schema Library will already be loaded into the system, to allow the user to test Schema execution. However, it is also possible to load new Schema Libraries into the system, as well as saving them for future use.

A.4.1.1. Saving a Schema Library

To save the current Schema Library to an XML file, simply click the *Save* button in the **Schema Library** section on the right-hand side of the screen, as illustrated by Figure A.11.

Figure A.11 – Save Button

When clicked, you will be presented with the window illustrated in Figure A.12.

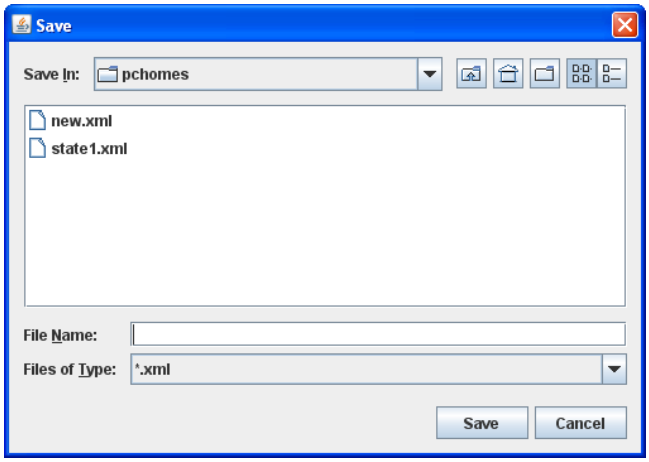

Figure A.12 – Save Library Window

To save the Schema Library, select an existing XML file, or create a new file, giving it the extension ".xml", e.g. "library.xml".

Now, click *Save*, and the contents of the Schema Library will be saved to the file specified.

A.4.1.2. Loading a Schema Library

NOTE: When you load a previously saved Schema Library into the system, the current Schema Library will be overwritten and lost, unless it has already been saved.

To load a previously saved Schema Library from an XML file, simply click the *Load* button in the **Schema Library** section on the right-hand side of the screen, as illustrated by Figure A.13.

Figure A.13 – Load Library Button

When clicked, you will be presented with the window illustrated in Figure A.14.

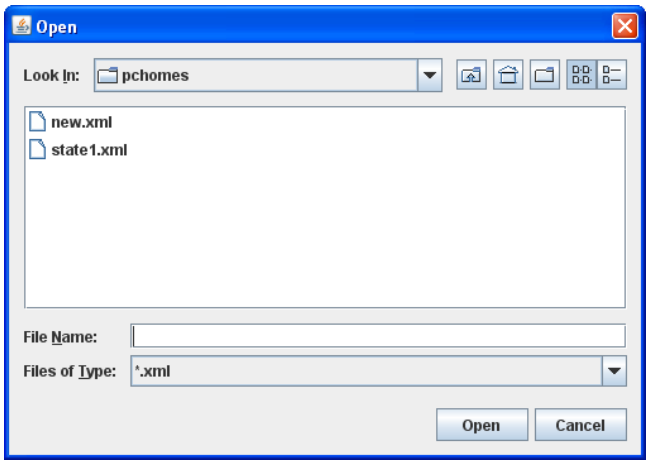

Figure A.14 – Load Library Window

To load a Schema Library, select the XML file you wish to load.

Now, click *Open*, and the Schema Library will be loaded into system memory.

A.4.2. Inspecting a Schema Library

To inspect the contents of a currently loaded Schema Library, simply click the *Inspect* button in the **Schema Library** section on the right-hand side of the interface, as illustrated in Figure A.15.

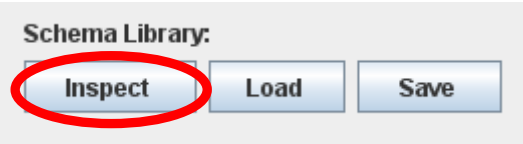

Figure A.15 – Inspect Library Window

When clicked, you will be presented with the window illustrated in Figure A.16.

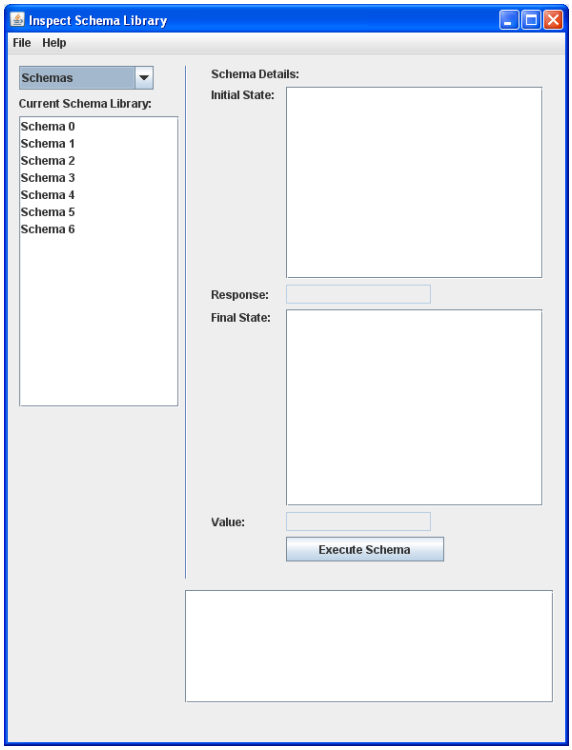

Figure A.16 – Inspect Library Window

From this interface, it is possible to inspect and execute the Schemas contained within the current Schema Library.

A.4.2.1. Inspecting "Normal" Schemas

To inspect "normal" Schemas within the Schema Library, select the *Schemas* option from the drop-down box as indicated below in Figure A.17.

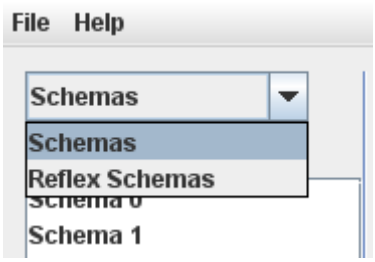

Figure A.17 – Selecting Schemas Option

Then, select the Schema you wish to inspect from the list presented. The contents of the Schema will now be displayed on screen, as illustrated by Figure A.18.

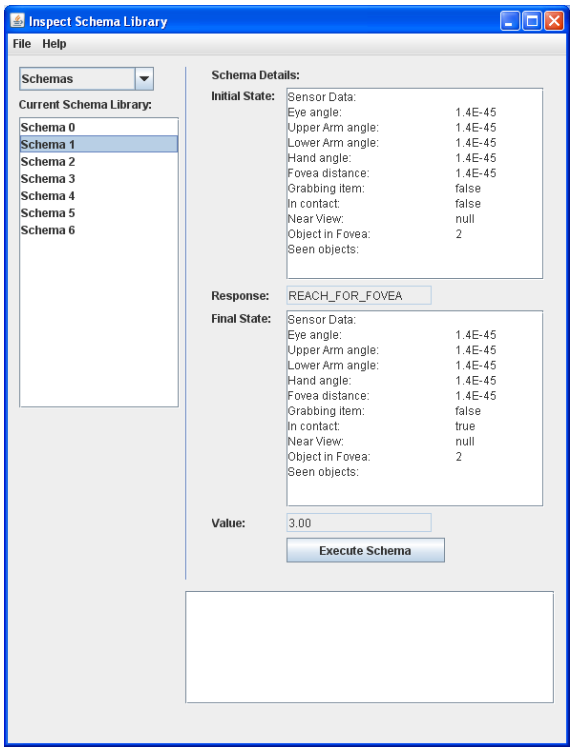

Figure A.18 – Viewing Schema Contents

A.4.2.2. Inspecting Reflex Schemas

To inspect Reflex Schemas within the Schema Library, select the *Reflex Schemas* option from the drop-down box as indicated below in Figure A.19.

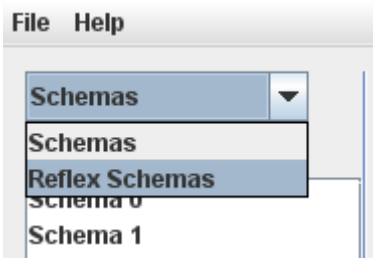

Figure A.19 – Selecting Reflex Schemas Option

Then, select the Reflex Schema you wish to inspect from the list presented. The contents of the Reflex Schema will now be displayed on screen, as illustrated by Figure A.20.

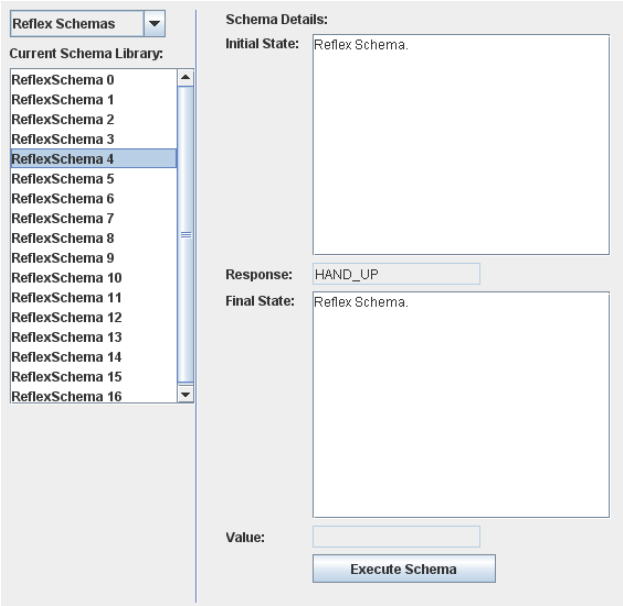

Figure A.20 – Viewing Reflex Schema Contents

A.4.2.3. Executing Schemas

To execute a Schema while inspecting the Schema Library, simply click the Schema you wish to execute, as above, and click the *Execute Schema* button, as displayed in Figure A.21, below.

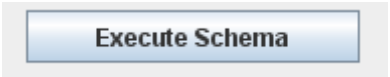

Figure A.21 – Execute Schema Button

If the Schema can be executed given the current state of the world, it will be fired on the graphical display. A suitable message will be displayed in the box at the bottom of the screen, depending on whether the Schema was successfully executed or not.

A.4.2.4. Exiting the Schema GUI

To exit this interface at any point, either click on the red cross at the top right-hand corner of the screen, or select the *Close...* option from this window's Menu Bar, as illustrated in Figure A.22, below.

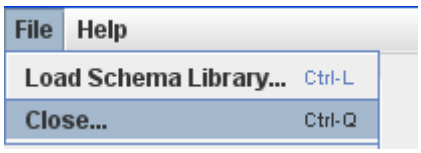

Figure A.22 – Close Schema Interface

A.4.3. Generating Transition Experiences

The system can also generate Transition Experience data, for use with the learning mechanism.

A.4.3.1. Saving Transition Experiences

To generate Transition Experiences in the system, first, specify the number of generations you wish to obtain, using the input box provided, as illustrated in Figure A.23. The amount of Transition Experiences to be generated must be a multiple of 4.

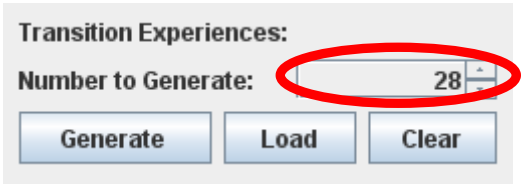

Figure A.23 – Assigning Number of Generations

When you have specified the number of Transition Experiences you wish to generate, click the *Generate* button in the **Transition Experiences** section on the right-hand side of the interface, as illustrated in Figure A.24.

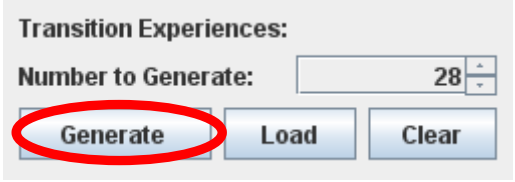

Figure A.24 – Generate Button

When clicked, you will be presented with the window illustrated in Figure A.25.

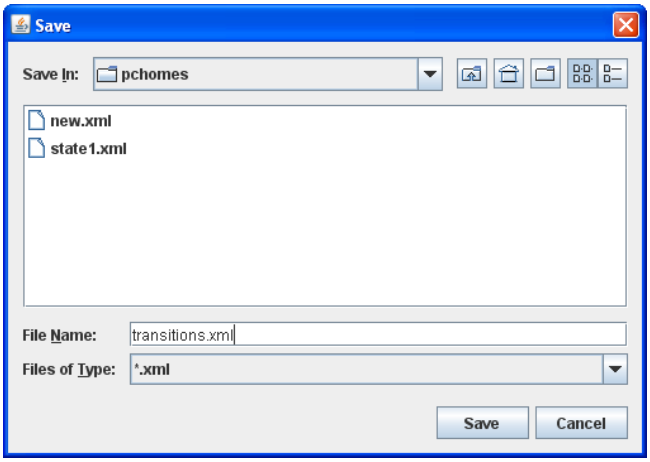

Figure A.25 – Save Generation Data Window

Now, select an existing XML file, or create a new file giving it the extension ".xml", e.g. "transitions.xml".

Then, click "Save", and wait until the Transition Experiences have all been generated.

When the generation has completed, a confirmation window will be displayed, like the one presented in Figure A.26.

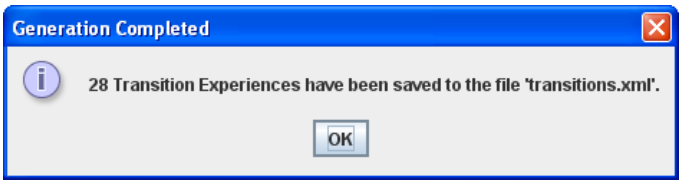

Figure A.26 – Generation Complete Confirmation

A.4.3.2. Loading Transition Experiences

To load Transition Experience Generation data that was previously saved to an XML document into the system, click on the *Load* button in the **Transition Experiences** section on the right-hand side of the interface, as illustrated in Figure A.27.

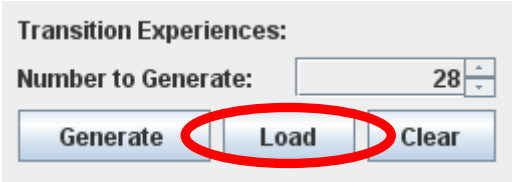

Figure A.27 – Load Generation Data

When clicked, you will be presented with window illustrated in Figure A.28.

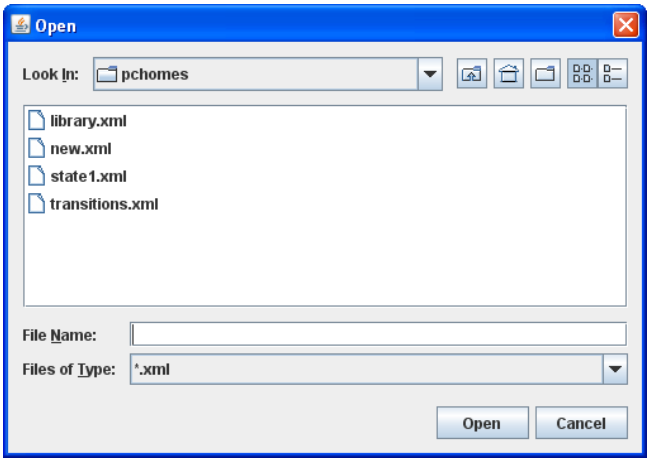

Figure A.28 – Load Generation Data Window

To load the required file into memory, click on an XML file containing Transition Experience data, and click the *Open* button.

The data will now have been loaded into memory.

A.4.3.3. Clearing the Display

If you have previously generated Transition Experience data and wish to remove the transition location indicators from the graphics display, click the *Clear* button in the **Transition Experiences** section on the right-hand side of the interface, as presented in Figure A.29.

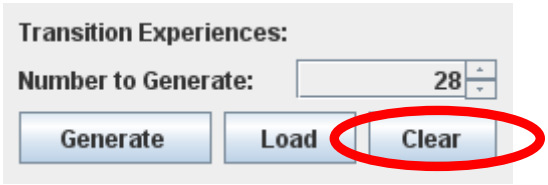

Figure A.29 – Clear Generation Data Button

A.4.4. Viewing Current World State Data

To view the data detailing the current state of the objects in the world, click on the *View Current State* button in the **Testing Controls** section on the right-hand side of the interface, as depicted in Figure A.30.

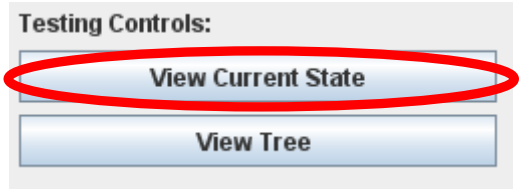

Figure A.30 – View Current State Button

When the button is clicked, a window similar to the one presented in Figure A.31 will appear, which displays data regarding the current state of the world at this point in the simulation.

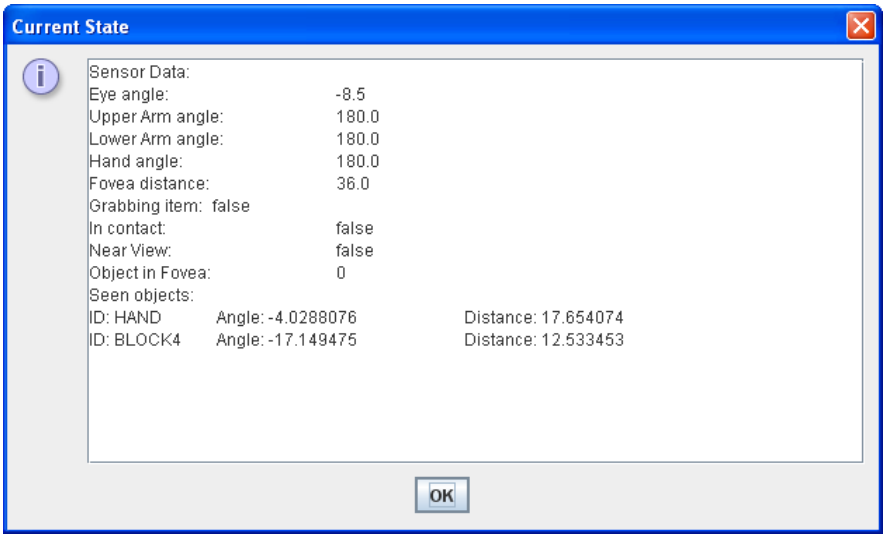

Figure A.31 – Displaying Current State Window

A.4.5. Viewing the Forward Simulator Tree

While the system is running, a Forward Simulator runs in the background, producing a Tree structure of Schemas, which, given the expected final state of each, should indicate chains which can be run.

To view the Forward Simulator tree, simply click on the *View Tree* button in the **Testing Controls** section at the bottom right-hand side of the interface, as presented in Figure A.32.

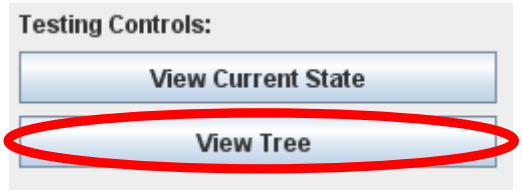

Figure A.32 – View Tree Button

When the button is clicked, a window similar to that presented in Figure A.33 will appear, which displays the Forward Simulator Tree at the current point in the simulation.

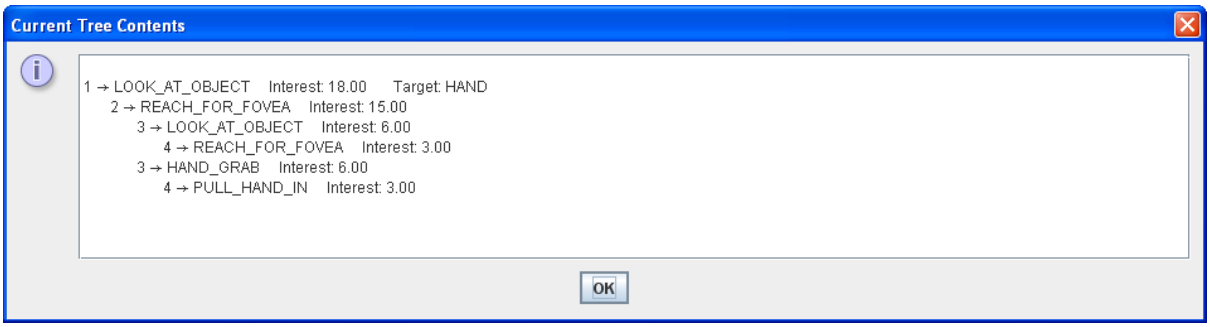

Figure A.33 – Display Current Tree Window

A.4.6. "Play" Functionality

To start a demonstration of the Schema execution functionality provided by the system, click on the *Play* button in the top right-hand corner of the screen, as illustrated by Figure A.34, below.

Figure A.34 – Play Button

The infant will now start executing random Schemas, displaying some of the more basic behaviours it possesses. The action the infant is taking will be indicated in the box of text on the right of the "Play" button.

Appendix B: Maintenance Manual

Appendix B: Maintenance Manual

The Maintenance Manual provides instructions for building and running the Developmental AI testbed, as well as detailing the source code classes provided.

B.1. Building and Running the System

The system can be run either from a Jar file, or from the Java source code folders in an IDE.

B.1.1. Running the System from a Jar File

To run the system from a Jar file, simply copy the Jar to the required location on your computer, and double click the file icon. The system will now run.

B.1.2. Running the System from Java Source Code Folders

To run the system from Java Source Code you will need an IDE, for example, NetBeans, or Eclipse. First, copy the source code folder into the location your IDE stores projects in, e.g. *NetBeansProjects* folder for NetBeans.

Next, open your IDE, and click *File*, and then *Open Project*. Locate the project folder where you stored it in the previous step, and click *Open*.

Now that the project is open in the IDE, right-click the folder, and click *Clean and Build* to build it, or, alternatively, use the *Build* icon as provided on the IDE interface.

To run the project, right-click the folder, and click *Run*, or, alternatively, use the *Run* icon as provided on the IDE interface.

B.2. Dependencies

The system has a number of hardware and software dependencies.

B.2.1. Hardware Dependencies

The system can be run on either a Windows or Linux computer, and there are no noticeable differences between versions.

The graphics display will require a monitor with a reasonable resolution, so it can be viewed correctly; most modern monitors should provide a sufficient resolution for this project.

B.2.2. Software Dependencies

The system must be run with at least Java Version 1.6, which can be obtained from Sun Microsystems' website:

- http://java.sun.com/

The system must be run from an up-to-date IDE. It is recommended that it is run from NetBeans, at least version 6.5, which can be obtained from the NetBeans developer website:

- http://netbeans.org/

All the required libraries, including *core.jar*, required for Processing 1.0, and *jbox2d-1.4.1b.jar*, required for JBox2D are packaged along with the project itself, and so do not need to be taken into consideration.

B.3. Memory Requirements

The entire project folder, including the libraries for the graphics package and the physics engine, requires around 11.8MB of space.

Running the system requires, on average, around 50MB of memory.

B.4. List of Classes

Detailed below is a list of all the source code classes contained within the project folder.

Developer Key: DB = David Bruce, JA = John Alexander.

Package	File Name	Description	Developer
robotbaby.gui	RobotGUI.java RobotGUI.form	Files required for creating the main User Interface provided by the system.	DB
	SchemaGUI.java SchemaGUI.form	Files required for creating the interface for inspecting the Schema Library, provided by the system.	DB
robotbaby.main	Consts.java	Stores all variables which are used across the entire system	DB
	Startup.java	Starts the system, initialising the Schema Library, and loading sample Schemas.	DB/JA
	Main.java	The main method of the system.	DB
robotbaby.robot	RobotBaby.java	The graphical representation of the infant, and associated methods.	DB
	GraphicsSettings.java	The settings for the graphical display.	DB
	BodyMovements.java	All methods for moving parts of the infant in the graphical display.	DB
	Fovea.java	The fovea object used by the infant, and all associated methods.	DB

	Eye.java	The eye object used by the infant, and all associated methods.	DB
robotbaby.storage	LoadRobotState.java	Provides method for loading a State of the world from an XML file onto the graphics display.	DB
	SaveRobotState.java	Provides method for saving the State of the world from the graphics display into an XML file.	DB
	XMLFilter.java	File Filter for saving/loading to XML files.	DB
schema.interaction	Response.java	Provides the action part of a Schema triple.	JA/DB
	SensorData.java	Representation of the state of the world on the graphics display, provides methods for state comparison.	DB/JA
	SeenObject.java	An object as seen by the infant's eye.	DB
schema.schema	Schema.java	Provides methods for the creation and execution of a Schema object.	DB/JA
	ReflexSchema.java	Provides methods for the creation and execution of a Reflex Schema object.	JA
schema.util	Util.java	Provides method for checking equality, given a delta value.	JA
schema.main	SchemaManager.java	Provides methods for managing the execution of Schema objects.	DB
	SchemaLibrary.java	Provides methods for storing and retrieving Schema objects	JA/DB
	SchemaHarvester.java	Provides a framework for the Schema Harvester class, not yet implemented.	DB
schema.storage	LoadSchemaLibrary.java	Provides a method to load the contents of a Schema Library from an XML file.	DB
	SaveSchemaLibrary.java	Provides a method to store the contents of a Schema Library to an XML file.	DB
icons	pause.png	Icon for GUI.	-
	play.png	Icon for GUI.	-
	Activation.java	Provides methods for the creation of Activation objects.	JA

cog.simulator	Bitmap.java	A bitmap snapshot of trajectory motion.	JA
	Frame.java	A bitmap frame and arm states of the trajectory of a movement.	JA
	ForwardSimulator.java	Provides methods for the creation of Tree, displaying chains of future Schema execution.	DB
cog.training	CrashFunction.java	Produces a neural network capable of recognizing the "crash" region.	JA
	NeuralFittedQ.java	Neural Fitted Q iteration algorithm.	JA
	PatternSet.java	Class that represents a set of training patterns.	JA
	SwingFunction.java	Functions for training a network for moving the arm up/down.	JA
	TransitionExperience.java	Represents a previously experienced transition.	JA/DB
cog.training.storage	TransitionExperienceGenerator.java	Generates Transition Experiences and stores the data.	DB
	LoadTransitionExperience.java	Loads Transition Experience data from an XML file into memory.	DB
	SaveTransitionExperience.java	Saves Transition Experience data into an XML file.	DB
	GenerateCrashExperience.java	Generates Crash Experiences.	JA
cog.util.tree	Tree.java	Tree structure.	-
	Node.java	Node of a tree.	-
cog.util.matrix	Matrix.java	Defines a matrix.	JA
	MatrixMath.java	Contains matrix operations.	JA
cog.rprop	ActivationFunction.java	Contains activation functions.	JA
	Layer.java	Represents a layer in a neural network.	JA
	Main.java	Used for testing.	JA
	Network.java	A feedforward neural network.	JA

Table B.1 – List of Source Code Files

B.5. Known Bugs

There are a number of bugs known to be present in the system, which have still to be addressed:

- When generating Transition Experiences, due to the arm repositioning techniques used, the hand will occasionally stray into the head object, or will get caught between the head and the neck.

  These Transition Experiences can be easily ignored when the generated data is used, but a more ideal solution would be to limit the hand's movement in these areas when a Transition Experience generation is active. However, limiting the hand does not provide a true representation of the arm movements in the system, so this issue would need to be addressed carefully.

- Also when generating Transition Experiences, because the arm is repositioning at such high speed, occasionally the arm segments will detach momentarily; this however, does not affect the results produced, and is purely a cosmetic glitch.

- If the hand reaches for an object it wishes to grab at high speed, then on some occasions, it will bounce away from the object, losing contact.

  Preventative measures were taken to ensure this does not happen, decreasing the restitution ("bounciness") of the hand object, but can still occur on occasion.

B.6. Future Development

A considerable amount of work will be undertaken both on, and using, this system in the future. The system implemented will provide a foundation for the learning and vision sections of the Developmental AI system, and hence parts of it will be refined in the future as new requirements become apparent.

The following tasks will all need to be achieved at some point, in order to progress with the creation of a Developmental AI system:

- **Integrating the vision and learning sections of the system.**

  Integrating the vision and learning packages into the system will not be an arduous task; both sections already make use of this system as part of their individual projects, and hence they have already been integrated to an extent. All that is required is to combine all three parts together into a single system.

- **Upgrading the graphics display to use JBox2D 2.0.1.**

  The newest version of JBox2D was made available from the developer's website a few weeks after the implementation phase began. However, a lot of classes and features have been altered considerably, so updating to the newer version during this project was not a viable option given the time learning the previous version had already consumed.

Updating to the new version is not entirely necessary; the current version sufficiently provides all the features required for this project. However, it is beneficial to the project in the long term to keep the physics engine up-to-date.

- **Facilitating the saving and loading of Schemas with Network objects as their Response.**

  Saving and loading Schema objects with a neural network as a response has not been implemented in the current system, due to the Network object, developed as part of the learning project, being of a format that was unsuitable for saving to an XML document.

  Hence, at the moment, Network instances are stored as synchronised objects, and loaded into the system when required. However, they cannot be saved or loaded as part of a Schema in a Schema Library, a facility which will need to be implemented in the future, for user convenience.

- **Implementing the Schema Harvester class functionality.**

  As discussed previously, the Schema Harvester class currently only provides a template in the system; it will need to be expanded and fully implemented, making use of machine learning techniques to determine which Schemas to store in the Library.

Appendix C: Example XML Files

Appendix C: Example XML Files

This Appendix presents a sample XML files for each of the three functionalities that produce them; World State, Schema Library contents and Transition Experience generation data.

C.1. World State XML Example

The following XML document contains data based on the state of the world at a specific moment.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<robotState>
	<object>
		<name>head</name>
		<xposition>-19.0</xposition>
		<yposition>2.0</yposition>
		<rotation>0.0</rotation>
	</object>
	<object>
		<name>shoulder</name>
		<xposition>-19.0</xposition>
		<yposition>-2.0</yposition>
		<rotation>0.0</rotation>
	</object>
	<object>
		<name>upperarm</name>
		<xposition>-15.933413</xposition>
		<yposition>-5.4247947</yposition>
		<rotation>-1.0278665</rotation>
	</object>
	<object>
		<name>lowerarm</name>
		<xposition>-9.892601</xposition>
		<yposition>-8.396215</yposition>
		<rotation>0.11358724</rotation>
	</object>
	<object>
		<name>hand</name>
		<xposition>-5.2480364</xposition>
		<yposition>-8.6848955</yposition>
		<rotation>-0.83612794</rotation>
	</object>
	<object>
		<name>block2</name>
		<xposition>5.994545</xposition>
		<yposition>-14.00496</yposition>
		<rotation>3.962186E-5</rotation>
	</object>
	<object>
		<name>block1</name>
		<xposition>-4.7457576</xposition>
		<yposition>-13.6789255</yposition>
		<rotation>0.13520686</rotation>
	</object>
	<object>
		<name>block3</name>
		<xposition>3.025613</xposition>
		<yposition>-14.005008</yposition>
		<rotation>1.5708002</rotation>
	</object>
	<object>
		<name>block4</name>
		<xposition>-3.2893538</xposition>
		<yposition>-9.178733</yposition>
```

```
            <rotation>4.3177137</rotation>
        </object>
        <grabbing>
            <value>true</value>
            <item>BLOCK4</item>
        </grabbing>
        <vision>
            <eyeAngle>-32.0</eyeAngle>
            <foveaXPosition>6.017419</foveaXPosition>
            <foveaYPosition>-13.632618</foveaYPosition>
            <foveaDistance>29.5</foveaDistance>
        </vision>
</robotState>
```

C.2. Schema Library XML Example

The following XML document contains data based on a Schema Library object with four basic Schemas loaded, along with the list of Reflex Schemas.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<schemaLibrary>
        <schema>
                <initialState>
                        <eye_angle>1.4E-45</eye_angle>
                        <fovea_distance>1.4E-45</fovea_distance>
                        <upperArm_angle>1.4E-45</upperArm_angle>
                        <lowerArm_angle>1.4E-45</lowerArm_angle>
                        <hand_angle>1.4E-45</hand_angle>
                        <in_contact>null</in_contact>
                        <grabbing_item>false</grabbing_item>
                        <near_view>null</near_view>
                        <in_fovea>-2147483648</in_fovea>
                        <seenObjects />
                </initialState>
                <response_type>LOOK_AT_OBJECT</response_type>
                <finalState>
                        <eye_angle>1.4E-45</eye_angle>
                        <fovea_distance>1.4E-45</fovea_distance>
                        <upperArm_angle>1.4E-45</upperArm_angle>
                        <lowerArm_angle>1.4E-45</lowerArm_angle>
                        <hand_angle>1.4E-45</hand_angle>
                        <in_contact>false</in_contact>
                        <grabbing_item>false</grabbing_item>
                        <near_view>null</near_view>
                        <in_fovea>2</in_fovea>
                        <seenObjects />
                </finalState>
        </schema>
        <schema>
                <initialState>
                        <eye_angle>1.4E-45</eye_angle>
                        <fovea_distance>1.4E-45</fovea_distance>
                         <upperArm_angle>1.4E-45</upperArm_angle>
                        <lowerArm_angle>1.4E-45</lowerArm_angle>
                        <hand_angle>1.4E-45</hand_angle>
                        <in_contact>false</in_contact>
                        <grabbing_item>false</grabbing_item>
                        <near_view>null</near_view>
                        <in_fovea>2</in_fovea>
                        <seenObjects />
                </initialState>
                <response_type>REACH_FOR_FOVEA</response_type>
                <finalState>
                        <eye_angle>1.4E-45</eye_angle>
                        <fovea_distance>1.4E-45</fovea_distance>
                        <upperArm_angle>1.4E-45</upperArm_angle>
                        <lowerArm_angle>1.4E-45</lowerArm_angle>
                        <hand_angle>1.4E-45</hand_angle>
                         <in_contact>true</in_contact>
                        <grabbing_item>false</grabbing_item>
                        <near_view>null</near_view>
                        <in_fovea>2</in_fovea>
```

```
                <seenObjects />
            </finalState>
        </schema>
        <schema>
            <initialState>
                <eye_angle>1.4E-45</eye_angle>
                <fovea_distance>1.4E-45</fovea_distance>
                <upperArm_angle>1.4E-45</upperArm_angle>
                <lowerArm_angle>1.4E-45</lowerArm_angle>
                <hand_angle>1.4E-45</hand_angle>
                <in_contact>true</in_contact>
                <grabbing_item>false</grabbing_item>
                <near_view>null</near_view>
                <in_fovea>-2147483648</in_fovea>
                <seenObjects />
                </initialState>
            <response_type>HAND_GRAB</response_type>
            <finalState>
                <eye_angle>1.4E-45</eye_angle>
                <fovea_distance>1.4E-45</fovea_distance>
                <upperArm_angle>1.4E-45</upperArm_angle>
                <lowerArm_angle>1.4E-45</lowerArm_angle>
                <hand_angle>1.4E-45</hand_angle>
                <in_contact>null</in_contact>
                <grabbing_item>true</grabbing_item>
                <near_view>false</near_view>
                <in_fovea>-2147483648</in_fovea>
                <seenObjects />
            </finalState>
        </schema>
        <schema>
            <initialState>
                <eye_angle>1.4E-45</eye_angle>
                <fovea_distance>1.4E-45</fovea_distance>
                <upperArm_angle>1.4E-45</upperArm_angle>
                <lowerArm_angle>1.4E-45</lowerArm_angle>
                <hand_angle>1.4E-45</hand_angle>
                <in_contact>null</in_contact>
                <grabbing_item>true</grabbing_item>
                <near_view>false</near_view>
                <in_fovea>-2147483648</in_fovea>
                <seenObjects />
            </initialState>
            <response_type>PULL_HAND_IN</response_type>
            <finalState>
                <eye_angle>1.4E-45</eye_angle>
                <fovea_distance>1.4E-45</fovea_distance>
                <upperArm_angle>1.4E-45</upperArm_angle>
                <lowerArm_angle>1.4E-45</lowerArm_angle>
                <hand_angle>1.4E-45</hand_angle>
                <in_contact>null</in_contact>
                <grabbing_item>true</grabbing_item>
                <near_view>true</near_view>
                <in_fovea>-2147483648</in_fovea>
                <seenObjects />
            </finalState>
        </schema>
        <reflexSchema>
            <response_type>ARM_UP</response_type>
```

```
        </reflexSchema>
        <reflexSchema>
                <response_type>ARM_DOWN</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>ARM_IN</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>ARM_OUT</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>HAND_UP</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>HAND_DOWN</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>HAND_GRAB</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>HAND_DROP</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>EYE_UP</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>EYE_DOWN</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>FOVEA_IN</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>FOVEA_OUT</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>FOVEA_FIXATE</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>LOOK_AT_OBJECT</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>PULL_HAND_IN</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>REACH_FOR_FOVEA</response_type>
        </reflexSchema>
        <reflexSchema>
                <response_type>PULL_BLOCK_ALONG</response_type>
        </reflexSchema>
</schemaLibrary>
```

C.3. Transition Experience XML Example

The following XML document contains data obtained from a Transition Experience generation in which 4 Transition Experiences were generated.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<transitions>
	<transitionExperience>
		<state>
			<eye_angle>0.0</eye_angle>
			<fovea_distance>36.0</fovea_distance>
			<upperArm_angle>180.0</upperArm_angle>
			<lowerArm_angle>180.0</lowerArm_angle>
			<hand_angle>180.0</hand_angle>
			<in_contact>false</in_contact>
			<grabbing_item>false</grabbing_item>
			<near_view>false</near_view>
			<in_fovea>0</in_fovea>
			<seenObjects>
				<seenObject>
					<object_id>HAND</object_id>
					<object_angle>-12.528808</object_angle>
					<object_distance>18.439089</object_distance>
				</seenObject>
			</seenObjects>
		</state>
		<action>ARM_UP</action>
		<statePrime>
			<eye_angle>0.0</eye_angle>
			<fovea_distance>36.0</fovea_distance>
			<upperArm_angle>180.0</upperArm_angle>
			<lowerArm_angle>180.0</lowerArm_angle>
			<hand_angle>180.0</hand_angle>
			<in_contact>false</in_contact>
			<grabbing_item>false</grabbing_item>
			<near_view>false</near_view>
			<in_fovea>0</in_fovea>
			<seenObjects>
				<seenObject>
					<object_id>HAND</object_id>
					<object_angle>-12.528808</object_angle>
					<object_distance>18.439089</object_distance>
				</seenObject>
			</seenObjects>
		</statePrime>
		<reward>0.0</reward>
	</transitionExperience>
	<transitionExperience>
		<state>
			<eye_angle>0.0</eye_angle>
			<fovea_distance>36.0</fovea_distance>
			<upperArm_angle>180.0</upperArm_angle>
			<lowerArm_angle>180.0</lowerArm_angle>
			<hand_angle>180.0</hand_angle>
			<in_contact>false</in_contact>
			<grabbing_item>false</grabbing_item>
			<near_view>false</near_view>
			<in_fovea>0</in_fovea>
```

```
                <seenObjects>
                        <seenObject>
                                <object_id>HAND</object_id>
                                <object_angle>-12.528808</object_angle>
                                <object_distance>18.439089</object_distance>
                        </seenObject>
                </seenObjects>
        </state>
        <action>ARM_DOWN</action>
        <statePrime>
                <eye_angle>0.0</eye_angle>
                <fovea_distance>36.0</fovea_distance>
                <upperArm_angle>183.251</upperArm_angle>
                <lowerArm_angle>170.70578</lowerArm_angle>
                <hand_angle>207.18231</hand_angle>
                <in_contact>false</in_contact>
                <grabbing_item>false</grabbing_item>
                <near_view>false</near_view>
                <in_fovea>0</in_fovea>
                <seenObjects>
                        <seenObject>
                                <object_id>HAND</object_id>
                                <object_angle>-12.527659</object_angle>
                                <object_distance>18.554304</object_distance>
                        </seenObject>
                </seenObjects>
        </statePrime>
        <reward>0.0</reward>
</transitionExperience>
<transitionExperience>
        <state>
                <eye_angle>0.0</eye_angle>
                <fovea_distance>36.0</fovea_distance>
                <upperArm_angle>181.88742</upperArm_angle>
                <lowerArm_angle>174.69485</lowerArm_angle>
                <hand_angle>183.41772</hand_angle>
                <in_contact>false</in_contact>
                <grabbing_item>false</grabbing_item>
                <near_view>false</near_view>
                <in_fovea>0</in_fovea>
                <seenObjects>
                        <seenObject>
                                <object_id>HAND</object_id>
                                <object_angle>-12.754098</object_angle>
                                <object_distance>18.51025</object_distance>
                        </seenObject>
                </seenObjects>
        </state>
        <action>ARM_IN</action>
        <statePrime>
                <eye_angle>0.0</eye_angle>
                <fovea_distance>36.0</fovea_distance>
                <upperArm_angle>185.70775</upperArm_angle>
                <lowerArm_angle>162.37778</lowerArm_angle>
                <hand_angle>234.53395</hand_angle>
                <in_contact>false</in_contact>
                 <grabbing_item>false</grabbing_item>
                <near_view>false</near_view>
                <in_fovea>0</in_fovea>
```

```
                        <seenObjects>
                                <seenObject>
                                        <object_id>HAND</object_id>
                                        <object_angle>-12.300963</object_angle>
                                        <object_distance>18.867094</object_distance>
                                </seenObject>
                        </seenObjects>
                </statePrime>
                <reward>0.0</reward>
        </transitionExperience>
        <transitionExperience>
                <state>
                        <eye_angle>0.0</eye_angle>
                        <fovea_distance>36.0</fovea_distance>
                        <upperArm_angle>184.06984</upperArm_angle>
                        <lowerArm_angle>167.04713</lowerArm_angle>
                        <hand_angle>188.88303</hand_angle>
                        <in_contact>false</in_contact>
                        <grabbing_item>false</grabbing_item>
                        <near_view>false</near_view>
                        <in_fovea>0</in_fovea>
                        <seenObjects>
                                <seenObject>
                                        <object_id>HAND</object_id>
                                        <object_angle>-12.680151</object_angle>
                                        <object_distance>18.772638</object_distance>
                                </seenObject>
                        </seenObjects>
                </state>
                <action>ARM_OUT</action>
                <statePrime>
                        <eye_angle>0.0</eye_angle>
                        <fovea_distance>36.0</fovea_distance>
                        <upperArm_angle>180.24261</upperArm_angle>
                        <lowerArm_angle>181.99556</lowerArm_angle>
                        <hand_angle>146.8706</hand_angle>
                        <in_contact>false</in_contact>
                        <grabbing_item>false</grabbing_item>
                        <near_view>false</near_view>
                        <in_fovea>0</in_fovea>
                        <seenObjects>
                                <seenObject>
                                        <object_id>HAND</object_id>
                                        <object_angle>-12.117595</object_angle>
                                        <object_distance>18.54825</object_distance>
                                </seenObject>
                        </seenObjects>
                </statePrime>
                <reward>0.0</reward>
        </transitionExperience>
</transitions>
```

Appendix D: Example Schemas

Appendix D: Example Schemas

Presented in the table below are a number of sample Schema objects which were created to ensure that the Schema Manager, and the Schema execution functionality worked correctly, as they were required to.

ID	Schema Description	Initial State	Response	Final State
A	Look at Object	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** null **Near View:** null **Object in Fovea:** -2147483648 **Seen objects:**	LOOK_AT_OBJECT	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** false **Near View:** null **Object in Fovea:** 2 **Seen objects:**
B	Reach for Fovea (object focussed on)	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** false **Near View:** null **Object in Fovea:** 2 **Seen objects:**	REACH_FOR_FOVEA	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** true **Near View:** null **Object in Fovea:** 2 **Seen objects:**
C	Reach for Fovea (no object focussed on)	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** false **Near View:** null **Object in Fovea:** 0 **Seen objects:**	REACH_FOR_FOVEA	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** false **Near View:** null **Object in Fovea:** 0 **Seen objects:**
D	Reach for Fovea (hand focussed on)	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** false **Near View:** null **Object in Fovea:** 1 **Seen objects:**	REACH_FOR_FOVEA	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** false **Near View:** null **Object in Fovea:** 1 **Seen objects:**
E	Grab object	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false	HAND_GRAB	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** true

		In contact: true **Near View:** null **Object in Fovea:** -2147483648 **Seen objects:**		**In contact:** null **Near View:** false **Object in Fovea:** -2147483648 **Seen objects:**
F	Drop object	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** true **In contact:** null **Near View:** null **Object in Fovea:** -2147483648 **Seen objects:**	HAND_DROP	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** null **Near View:** null **Object in Fovea:** -2147483648 **Seen objects:**
G	Pull hand in (object grabbed)	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** true **In contact:** null **Near View:** false **Object in Fovea:** -2147483648 **Seen objects:**	PULL_HAND_IN	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** true **In contact:** null **Near View:** true **Object in Fovea:** -2147483648 **Seen objects:**
H	Pull hand in (no object grabbed)	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** null **Near View:** false **Object in Fovea:** -2147483648 **Seen objects:**	PULL_HAND_IN	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** null **Near View:** true **Object in Fovea:** -2147483648 **Seen objects:**
I	Pull Object Along Ground	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** true **In contact:** null **Near View:** null **Object in Fovea:** -2147483648 **Seen objects:**	PULL_BLOCK_ALONG	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** null **Near View:** null **Object in Fovea:** -2147483648 **Seen objects:**
J	Look at object, reach for object (object focussed on), grab object, pull hand in (object grabbed)	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** false **In contact:** null **Near View:** null **Object in Fovea:** -2147483648 **Seen objects:**	List of sub-Schemas composed of Schemas A,B,E,G	**Eye angle:** 1.4E-45 **Upper Arm angle:** 1.4E-45 **Lower Arm angle:** 1.4E-45 **Hand angle:** 1.4E-45 **Fovea distance:** 1.4E-45 **Grabbing item:** true **In contact:** null **Near View:** true **Object in Fovea:** -2147483648 **Seen objects:**

Table D.1 – Sample Schema Objects

Glossary

The glossary provides definitions for terms commonly used throughout this report.

Term	Definition
Cognitive Architecture	An attempt to provide a plausible model of the main internal components of the mind, and how they are linked.
Developmental AI	Studies in AI attempting to overcome the shortcomings of "traditional" AI, in that there are attempts to get programs to learn "common sense" knowledge for themselves, and not just what they have been programmed to "know".
Developmental AI system	A system which can develop knowledge and skills from its environment autonomously, by experimenting within it, and learning from the results produced.
Extensibility	Taking into consideration future growth of a system.
Modularity	The degree to which the components of a system may be separated and re-combined.
Response	The action part of a Schema.
Schema	A snapshot of internal knowledge about a child's surroundings relating to a specific behaviour at a particular moment in time. A 3-tuple consisting of (Initial State, Response, Final State). Originally defined by Jean Piaget.
State	The representation of an environment at a specific time, forms the initial and final states of a Schema.
Testbed	A platform to allow for the experimentation and testing of software projects.